Type Test Scripts for TypeScript Testing

ERIK KROGH KRISTENSEN, Aarhus University, Denmark
ANDERS MØLLER, Aarhus University, Denmark

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.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: feedback-directed random testing, JavaScript, types

ACM Reference Format:
Erik Krogh Kristensen and Anders Møller. 2017. Type Test Scripts for TypeScript Testing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 90 (October 2017), 25 pages.
https://doi.org/10.1145/3133914

1 INTRODUCTION

The TypeScript programming language [Microsoft 2015] 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 [Feldthaus and Møller 2014], and assisting in the creation of declaration files and in updating the declarations as the

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

Authors' email addresses: {erik,amoeller}@cs.au.dk.

2475-1421/2017/10-ART90
https://doi.org/10.1145/3133914

JavaScript implementations evolve [Kristensen and Møller 2017]. However, those techniques rely on unsound static analysis and consequently often overlook errors and report spurious warnings.

Gradual typing [Siek and Taha 2006] 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 [Rastogi et al. 2015]. This can be adapted to check TypeScript declaration files [Williams et al. 2017] by wrapping the JavaScript implementation in a higher-order contract [Keil and Thiemann 2015b], 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. [2017] 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 [2014] and Kristensen and Møller [2017] 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. [2017], our method does not require existing test cases, and it avoids the performance overhead and interference problems of higher-order contract systems for JavaScript.

Our method is based on the idea of *feedback-directed random testing* as pioneered by the Randoop tool by Pacheco et al. [2007]. 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).

```
1  declare var Path: {
2     root(path: string): void;
3     routes: {
4        root: IPathRoute,
5     }
6  };
7
8  interface IPathRoute {
9     run():void;
10 }
```

(a) A part of the TypeScript declaration.

```
11 var Path = {
12    root: function (path) {
13       Path.routes.root = path;
14    },
15    routes: {
16       root: null
17    }
18 };
```

(b) A part of the JavaScript implementation.

```
19 *** Type error
20   property access: Path.routes.root
21   expected: object
22   observed: string
```

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

Fig. 1. Motivating example from the *PathJS* library.

- 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 2 shows two examples that motivate TSTEST. Section 3 describes the basic structure of the type test scripts generated by TSTEST, and Section 4 explains how to handle the various challenging features of TypeScript. Section 5 describes the main theoretical properties, Section 6 presents our experimental results, Section 7 discusses related work, and Section 8 concludes.

2 MOTIVATING EXAMPLES

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

2.1 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 1a and 1b contain parts of the declaration file and the implementation, respectively. The `Path.root` method (line 12) 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 2), which does not match the type of the variable `Path.routes.root` (line 4). By inspecting how the variable is used elsewhere

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

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

```
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.

Fig. 2. Motivating example from the *Async* library.

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 [Facebook 2017] also cannot find the error, 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 1c 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.

2.2 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 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

[4] https://github.com/caolan/async
[5] https://www.npmjs.com/package/async

a special value that represents the error or the result value. In the declared type of `reflect` (see Figure 2b), 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 2a) 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 2c, 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 2a), 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.

3 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 [Pacheco et al. 2007]. 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 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 [Pacheco et al. 2007], 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 1). 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

```
60 declare module async {
61     function memoize(
62       fn: Function,
63       hasher?: Function
64     ): Function;
65     function unmemoize(fn: Function):
66       Function;
67 }
```

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

```
68 var async = {
69   memoize: function(fn, hasher) {
70     hasher = hasher ||
71       function (a) {
72         return JSON.stringify(a)};
73     var cache = {};
74     var result = function() {
75       var key = hasher(arguments);
76       return cache[key] ||
77        (cache[key] =
78          fn.apply(this, arguments));
79     };
80     result.unmemoized = fn;
81     return result
82   },
83   unmemoize: function(fn) {
84     return fn.unmemoized || fn;
85   }
86 };
```

(b) A simplified implementation.

```
 87 var vals = initializeVals();
 88 function makeValue1() {
 89   return selectVal(vals[1], vals[2],
 90     mkFunction());
 91 }
 92 var lib = require("./async.js");
 93 if (assertType(lib, "async"))
 94   vals[0] = lib;
 95 while (!timeout()) {
 96   try {
 97   switch (selectTest()) {
 98     case 0: // testing async.unmemoize
 99       var result =
100         vals[0].unmemoize(makeValue1());
101       if (assertType(result, "Function"))
102         vals[1] = result;
103       break;
104     case 1: // testing async.memoize
105       var result =
106         vals[0].memoize(makeValue1(),
107           makeValue1());
108       if (assertType(result, "Function"))
109         vals[2] = result;
110       break;
111   } } catch(e) {}
112 }
```

(c) The type test script for the declaration in Figure 3a.

Fig. 3. The type test script generated by TSTEST for a subset of the *Async* library.

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 4 and present results from an empirical evaluation in Section 6.

Figure 3 contains a small example of a type test script generated by TSTEST for a simplified version of the *Async* library that we also discussed in Section 2.2. Figures 3a and 3b show the declaration file and the implementation, respectively, and Figure 3c 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 69–82) uses JavaScript's meta-programming capabilities to implement function memoization, as its name suggests. The `unmemoize` function (lines 83–85) is overloaded, such 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 type system only specifies that unmemoize takes a function as argument (line 65). 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 87) 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 92–94). The main loop proceeds until a timeout is reached (line 95), which is determined either by the time spent or the number of iterations, according to a user provided configuration. The selectTest helper function (line 97) picks a test to run, based on which entries of the vals array have been filled in. The assertType helper function (lines 93, 101 and 108) checks if the first argument has the type specified by the second argument according to the declaration file. For example, line 93 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 88–91). 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 98–103). 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.

4 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 3c 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 116 in Figure 4), 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 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.

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

```
declare class Chart {
  constructor(
      context: string | JQuery | ...,
      options: Chart.ChartConfiguration
   );
  ...
}

export interface ChartConfiguration {
  type?: ChartType | string;
  data?: ChartData;
  options?: ChartOptions;
}
```

```
export interface ChartOptions {
  events?: string[];
  onClick?: (any?: any) => any;
  title?: ChartTitleOptions;
  legend?: ChartLegendOptions;
  ...
}

export interface ChartTitleOptions {
  fontSize?: number;
  fontFamily?: string;
  fontColor?: ChartColor;
  ...
}
```

Fig. 4. An example of structural typing in *Chart.js*.

4.1 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 3c) picks randomly (50/50) between these two sources of values. In TypeScript, types can be declared with `interface` or `class` (see examples in Figure 4), 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 100 and 107), 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 [Sen et al. 2005]. Creation of values for function types is explained in Section 4.2.

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.

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

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

```
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.

Fig. 5. Testing higher-order functions.

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 [Feldthaus and Møller 2014; Kristensen and Møller 2017], 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.

4.2 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 [Siek and Taha 2006; Siek et al. 2015]. 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

[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.

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 [Wadler and Findler 2009] provides a powerful foundation for tracking function contracts (e.g. types) dynamically and deciding 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 6.1 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 3), 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 78 in Figure 3).

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 5a and 5b, 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 147). The type declaration of the callback function, however, requires its argument to be a string (line 153). This mismatch is detected by the test shown in Figure 5c, which is a part of the type test script generated by TSTEST. A function is constructed (lines 157–165) according to the type declared on line 153. This function first checks if the argument matches the declared type, just like when receiving a value from the library. On line 160 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 [Wadler and Findler 2009]. When running the type test script, the report in Figure 5d is produced. Similar to the report from Section 3 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 signatures, 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 signatures.

Overloaded callback functions are a bit more involved. The callback generated by TSTEST first checks the argument types with respect to each of the signatures in turn. If exactly one of the signatures match, a value of the corresponding return type is produced, as for a non-overloaded callback. If none of the signatures 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 signatures 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

```
170 declare var c1: Cell<string>;
171 declare var c2: Cell<number>;
172 interface Cell<T> {
173   value: T;
174 }
```

(a) A simple declaration with a generic type.

```
175 interface _Chain<T> {
176     partition(...): _Chain<T[]>;
177     values(): T[];
178     ...
179 }
180 declare var foo: _Chain<boolean>;
```

(b) A recursively defined generic type.

Fig. 6. Examples for explaining how generic types are handled by TSTEST.

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 signatures match, we let the constructed callback throw an exception (similar to the case where none of the signatures 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 [Keil and Thiemann 2015a].

4.3 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 2.2. Figure 6a 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 6b from the *Underscore.js*[9] library. We test that `foo` yields a value of type `_Chain<boolean>` and that `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 176), we treat the type parameter `T` as TypeScript's built-in type `any` that matches any value (see also Section 4.4). 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 5.

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

TypeScript also supports generic functions. In recent work on TypeScript, Williams et al. [2017] have chosen to enforce *parametricity* [Wadler 1989] 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.

```
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:

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

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. [2017] 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.

```
class Foo<T> {...}
function foo<T1>(t: T1): Foo<T1>;
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 2c), 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 6.3). Previous work on testing for Java [Fraser and Arcuri 2014] 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.

[10] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/lodash/index.d.ts#L14903
[11] Object literals like these are also valid types since TypeScript 2.0.

4.4 Other Design Choices Involving Types

TypeScript has, as other languages with optional types, the special type `any` that effectively disables static type checking [Bierman et al. 2014]. 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 5.

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 4.1 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 4.2 for overloaded function signatures.

5 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 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 4.3, 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

[12] Cf. the well-known quote by Dijkstra [1970].

[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.

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 6.4.

The discussion about soundness of the testing is complicated by the fact that TypeScript's type system is intentionally unsound, which is well documented [Bierman et al. 2014]. 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 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 [Godefroid et al. 2005], 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 4.4, 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 4.4, 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 4.1. 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 EXPERIMENTAL EVALUATION

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

6.1 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 analyzers, 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

[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/

be constructed by the DOM API—having the right properties is not enough. Instead of using the normal approach described in Section 4.1 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 5) 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.

6.2 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 4 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 [Feldthaus and Møller 2014] and TSINFER [Kristensen and Møller 2017]?

[18] The record/replay feature of the Jalangi tool [Sen et al. 2013] was abandoned for exactly this reason.

Table 1. Total number of type mismatches found by TSTEST, for different time budgets and repeated runs.

Timeout	Mismatches found			
	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	-	-

As benchmarks for experiments we use all of the libraries used by Feldthaus and Møller [2014] and Kristensen and Møller [2017], 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 Appendix A.

6.3 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()` 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 6.1. A summary of the results can be found in Table 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 [2014], Kristensen and Møller [2017], and Williams et al. [2017] that errors are common, even in declaration files for highly popular libraries. The numbers in Table 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 6.4) 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 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 6.1 that periodically resetting the library state is beneficial.

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

Table 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)

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

We evaluate the four most interesting design choices discussed in Section 4: (1) In Section 4.1 we discussed the use of a shallow type check for determining whether to use a value for feedback. What if a deep type check is used instead? (2) Another design choice in Section 4.1 involved the treatment of class types. What if we also generate random values for class types? (3) In Section 4.3 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 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 mismatches 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

Table 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%

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 3c) in the type test script were executed. (Recall from Section 3 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 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 3). The dominant cause of uncovered test cases is the feedback mechanism being unable to provide the required base objects

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

for testing method calls (see Section 4.1). 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 [Godefroid et al. 2005] to increase the coverage.

6.4 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 4.1).
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 5). 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:

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:

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

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

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 4.1, 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 [Feldthaus and Møller 2014; Kristensen and Møller 2017]. 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 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.

6.5 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 [Feldthaus and Møller 2014]. 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 [Kristensen and Møller 2017] 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 6.4, 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.

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

Some of the mismatches reported in Section 6.4 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.

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 [Feldthaus and Møller 2014], 23% of the found mismatches were false positives (and other 16% were benign). While the evaluation of tsinfer [Kristensen and Møller 2017] did not test for its efficiency in finding bugs, the quality of inferring method signatures was tested, and here tsinfer was able to infer the correct method signature for 23% of the signatures, 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 6.4, we observe no false positives with TSTest.

7 RELATED WORK

Detecting type errors in dynamically typed programs. The most closely related work is tscheck [Feldthaus and Møller 2014], 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 6.5. Recently, Kristensen and Møller [2017] 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 [Williams et al. 2017] 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 [2009]. Unlike TSTest, it does not perform automated exploration of the library code but relies on existing test suites. Also, as discussed in Section 4.3, that approach suffers from interference caused by the use of proxies. Safe TypeScript [Rastogi et al. 2015] 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 [Heidegger and Thiemann 2010] 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 [Pradel et al. 2015] is a dynamic analysis that warns about inconsistent types in JavaScript programs, but it does not use TypeScript types nor automated testing. TAJS [Jensen et al. 2009] 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.

Flow [Facebook 2017] 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 [Tobin-Hochstadt and Felleisen 2008] and Python [Lehtosalo et al. 2016; Vitousek et al. 2014]. 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 3, our approach builds on the idea of feedback-directed random testing pioneered by the Randoop tool [Pacheco et al. 2007]. 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 [Fraser and Arcuri 2014], which has support for testing generic classes in Java, similar to the challenge we address in Section 4.3. Property-based testing, or quickchecking [Claessen and Hughes 2000], 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 [Artzi et al. 2011] 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 A), 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 [Godefroid et al. 2005] to boost coverage. More specifically, the techniques used in CUTE [Sen et al. 2005] 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 possible to leverage previous work on symbolic execution for JavaScript from Kudzu [Saxena et al. 2010], Jalangi [Sen et al. 2013], or SymJS [Li et al. 2014].

8 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.

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).

REFERENCES

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* ACM, 571–580.

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science)*, Vol. 8586. Springer, 257–281.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* ACM, 268–279.

Edsger W. Dijkstra. 1970. *Notes on Structured Programming.* Technical Report EWD249. Technological University Eindhoven.

Facebook. 2017. Flow. (2017). http://flowtype.org/.

Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *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.* ACM, 1–16.

Gordon Fraser and Andrea Arcuri. 2014. Automated Test Generation for Java Generics. In *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering - 6th International Conference, SWQD 2014, Vienna, Austria, January 14-16, 2014. Proceedings (Lecture Notes in Business Information Processing)*, Vol. 166. Springer, 185–198.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005.* ACM, 213–223.

Phillip Heidegger and Peter Thiemann. 2010. Contract-Driven Testing of JavaScript Code. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6141. Springer, 154–172.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5673. Springer, 238–255.

Matthias Keil and Peter Thiemann. 2015a. Blame assignment for higher-order contracts with intersection and union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* ACM, 375–386.

Matthias Keil and Peter Thiemann. 2015b. TreatJS: Higher-Order Contracts for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 28–51.

Erik Krogh Kristensen and Anders Møller. 2017. Inference and Evolution of TypeScript Declaration Files. In *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 (Lecture Notes in Computer Science)*, Vol. 10202. Springer, 99–115.

Jukka Lehtosalo et al. 2016. Mypy. (2016). http://www.mypy-lang.org/.

Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* ACM, 449–459.

Microsoft. 2015. TypeScript Language Specification. (February 2015). https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007.* IEEE

Computer Society, 75–84.

Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1.* IEEE Computer Society, 314–324.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* ACM, 167–180.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. 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*. IEEE Computer Society, 513–528.

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *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.* ACM, 488–498.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *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.* ACM, 263–272.

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 274–293.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* ACM, 395–406.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014.* ACM, 45–56.

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK, September 11-13, 1989.* ACM, 347–359.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *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 (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 1–16.

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 28:1–28:29.

A 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.