Highly Illogical, Kirk: Spotting Type Mismatches in the Large Despite Broken Contracts, Unsound Types, and Too Many Linters

JOSHUA HOEFLICH, Northwestern University, USA
ROBERT BRUCE FINDLER, Northwestern University, USA
MANUEL SERRANO, Inria/UCA, France

The DefinitelyTyped repository hosts type declarations for thousands of JavaScript libraries. Given the lack of formal connection between the types and the corresponding code, a natural question is *are the types right?* An equally important question, as DefinitelyTyped and the libraries it supports change over time, is *how can we keep the types from becoming wrong?*

In this paper we offer Scotty, a tool that detects mismatches between the types and code in the DefinitelyTyped repository. More specifically, Scotty checks each package by converting its types into contracts and installing the contracts on the boundary between the library and its test suite. Running the test suite in this environment can reveal mismatches between the types and the JavaScript code. As automation and generality are both essential if such a tool is going to remain useful in the long term, we focus on techniques that sacrifice completeness, instead preferring to avoid false positives. Scotty currently handles about 26% of the 8006 packages on DefinitelyTyped (61% of the packages whose code is available and whose test suite passes).

Perhaps unsurprisingly, running the tests with these contracts in place revealed many errors in DefinitelyTyped. More surprisingly, despite the inherent limitations of the techniques we use, this exercise led to one hundred accepted pull requests that fix errors in DefinitelyTyped, demonstrating the value of this approach for the long-term maintenance of DefinitelyTyped. It also revealed a number of lessons about working in the JavaScript ecosystem and how details beyond the semantics of the language can be surprisingly important. Best of all, it also revealed a few places where programmers preferred incorrect types, suggesting some avenues of research to improve TypeScript.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Contracts, Gradual Typing, TypeScript, Definitely Typed, Buggy Types

ACM Reference Format:
Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. 2022. Highly Illogical, Kirk: Spotting Type Mismatches in the Large Despite Broken Contracts, Unsound Types, and Too Many Linters. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 142 (October 2022), 26 pages. https://doi.org/10.1145/3563305

1 INTRODUCTION

We built Scotty, a prototype infrastructure that scans every package in DefinitelyTyped, a large, Microsoft-maintained repository of TypeScript type declarations, in order to find inconsistencies between JavaScript implementations and their corresponding type declarations. Scotty is a mostly automatic system. It triggers errors when a mismatch is observed, but an absence of errors does not guarantee the correctness of a type declaration and, conversely, an error reported by Scotty is

Authors' addresses: Joshua Hoeflich, Northwestern University, USA, ; Robert Bruce Findler, Northwestern University, USA, ; Manuel Serrano, Inria/UCA, France, .

2475-1421/2022/10-ART142
https://doi.org/10.1145/3563305

not always the symptom of a true error in a declaration. However, Scotty lets us detect potential problems in 567 packages and it helped us submit just over 100 accepted pull requests. Despite its current weaknesses, some being inherent to the approach and some being possible to fix with additional engineering effort, we have found that Scotty is nonetheless already useful enough in practice to detect many meaningful bugs.

As of April 2022, on a reasonably fast server, Scotty takes less than 6 hours to analyze the whole DefinitelyTyped repository (containing 8006 packages). Scotty can also be used in an incremental mode where only modified packages are inspected. This feature makes Scotty a potentially valuable part of a continuous integration system.

A brief description of the languages and techniques at play is in order.

- TypeScript is a front-end to JavaScript. The TypeScript compiler verifies the syntactic compliance of the programs. It type checks them, which involves checking the compatibility of the declared types and inferring types for unannotated variables and functions. In order to execute TypeScript programs, it erases all types to produce JavaScript programs that can be executed by any unmodified JavaScript engine.
  TypeScript's design is governed by two essential rules: full compatibility with JavaScript, which means unrestricted interoperability between the two languages, and static error detection. JavaScript's high degree of dynamicity, however, may seem consubstentially incompatible with the idea of static enforcement. To solve that contradiction, TypeScript's type system is unsound. That is, there is no guarantee that the static types checked by the static type checker do indeed correspond to the values that appear during evaluation. Thus, there is no formal correspondence between the static types used by the TypeScript compiler and the dynamic types used by the JavaScript engine executing the program. Indeed, the desire for unrestricted interoperation with JavaScript seems to have removed other inhibitions and even a fully typed program in TypeScript might have values occupying its variables that do not match the variables' types.
  Using an unsound type system is a controversial choice. Here, we merely observe that according to others' data,[1] TypeScript popularity is substantial and its community is growing steadily. Also, in spite of the unsoundness of its type system, its static analysis helps to detect bugs (Gao et al. 2017).
- DefinitelyTyped is a public repository where contributors provide type declarations for existing JavaScript packages. It holds a central position in the TypeScript ecosystem. Providing an additional type declaration to a JavaScript package enables the TypeScript compiler to check the correctness of that package and its uses from within TypeScript code. As of 2022, DefinitelyTyped offers type declarations for about 8,000 JavaScript packages, and its official git repository shows about 80,000 commits by about 14,000 contributors. This is a huge database of types!

Many of the DefinitelyTyped types are crafted manually. Some others are partially generated by the TypeScript compiler that can be used to infer types from JavaScript programs. However, in both cases, there is no guarantee that these types are faithful to their corresponding JavaScript libraries. Our approach to finding these inconsistencies is to reuse techniques developed for natural-style gradual typing (Gronski et al. 2006; Tobin-Hochstadt and Felleisen 2006) and higher-order contracts (Findler and Felleisen 2002), following the pioneering work by Williams et al. (2017) with their tool, TPD.[2] Their work discovered significant problems due to the interference introduced

[1] https://madnight.github.io/githut/# and https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

[2] TPD stands for "The Prime Directive" from Star Trek and inspires our paper's title and tool name as well.

Table 1. Runing Scotty on all 8006 packages in Definitely Typed. **Yes** and **No** are the numbers of packages that respectively pass to the next phase or are eliminated. For *Can Fetch?* and *Can Test?*, the percentages are computed from the overall DefinitelyTyped package count. For *Can Inject?*, *Can Interpose?*, and *Contract Violation?* two percentages are computed. The first is from the overall DefinitelyTyped package count and the second from the number of packages that pass the *Can Test?* step. The Time column give the wall-clock time it takes to run the phases for all the packages at that step, on a decently modern server.

Step	Yes	No	Time
Can Fetch?	6446 (81%)	1560 (19%)	3.5 hours
Can Test?	3384 (42%)	3062 (38%)	
Can Inject?	2417 (30%, 71%)	967 (12%, 29%)	55 mins
Can Interpose?	2069 (26%, 61%)	348 (4%, 10%)	45 mins
Contract Violation?	567 (7%, 17%)	1502 (19%, 44%)	47 mins

by higher-order contracts, and, in the end, rejected the approach. Our experience confirms their observation that interference of higher-order contracts is problematic, but also differs from theirs as interference is not, in our experience, a significant impediment to discovering bugs in DefinitelyTyped. Instead, we found that the wide variety of build steps, testing frameworks, and linters that decorate the JavaScript ecosystem are more substantial challenges to overcome.

Despite the challenges, Scotty was able to lead us to hundreds of errors in DefinitelyTyped. After some work to try to debug the errors, we were able to open slightly more than 100 pull requests on DefinitelyTyped to fix the errors, of which all but two were merged. Our activity over about six months currently places the first author as the 40th contributor on DefinitelyTyped.

The rest of the paper is organized as follows. In section 2, we present the infrastructure we have developed for checking packages. Section 3 presents the higher-order contract system we have created for dynamically checking TypeScript types and the compiler that translates TypeScript types into contracts. The errors in DefinitelyTyped are themselves interesting and we present an analysis of them in section 4. Finally, section 5 presents the related work and section 6 concludes.

2 SCOTTY

Scotty tries to download and test all of the untyped packages listed on DefinitelyTyped, but the immense variation in the JavaScript ecosystem makes this task difficult. While the npm registry where these packages are hosted provides rudimentary structural guarantees, the command line interface offers commands like "`npm test`" that simply invoke one-line shell scripts. These scripts can behave in unexpected and surprising ways, from starting a web server that never exits to spinning up GUIs or playing songs. Nonetheless, they behave predictably often enough that our system can detect a meaningful number of bugs.

As of April 2022, DefinitelyTyped counts 8006 packages and Scotty attempts to process all of them. Packages are eliminated for a number of different reasons. Some packages simply do not provide source code or test suite. Some packages are intrinsically incompatible with Scotty, for instance because of its use of higher-order contracts. Some other packages cannot be checked because of current limitations of Scotty's implementation.

In order to bring some structure to the chaos of the number of ways that packages might fail, Scotty is organized in a series of steps, as detailed in figure 1. After each step, some number

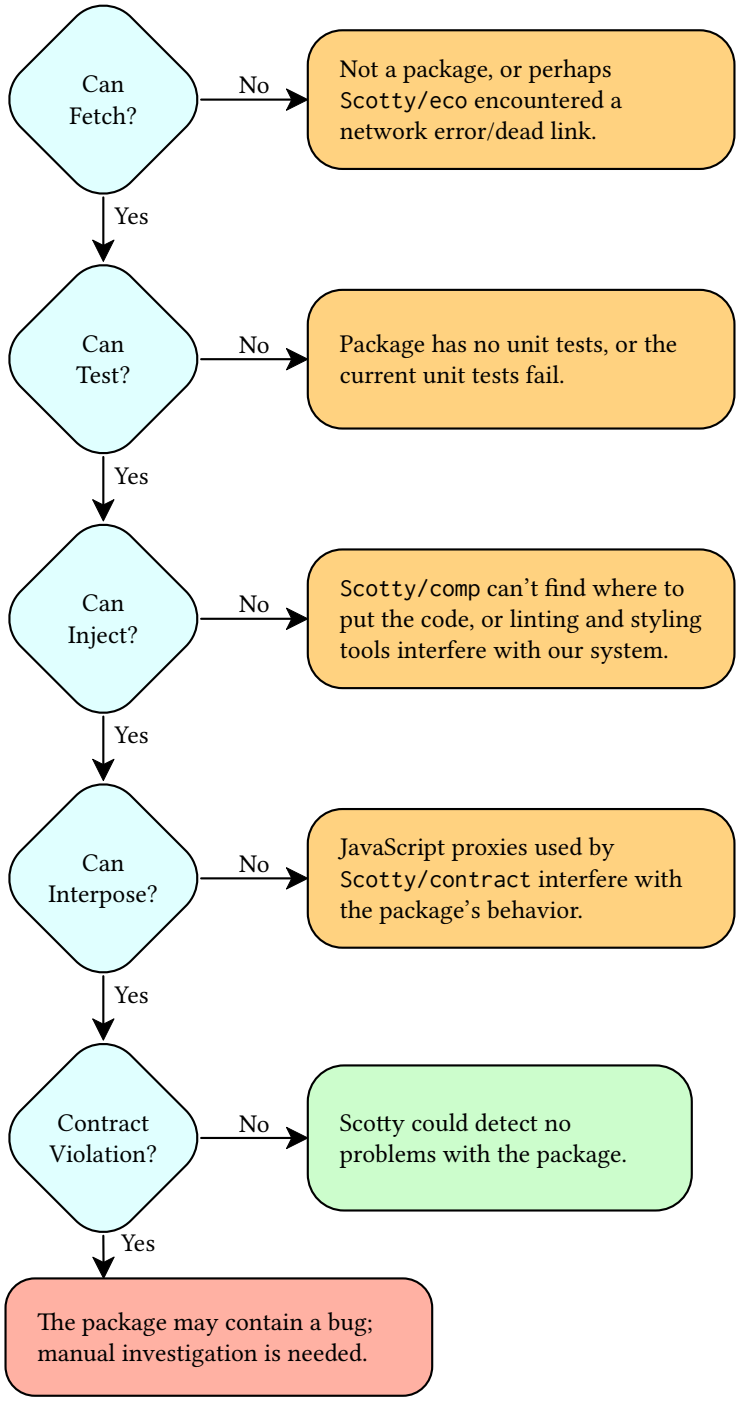

Fig. 1. The steps that Scotty takes.

```
### ECO:STEP 1/7: 2022-07-30T05:48:29.353Z (Download the Source Code)
Cloning into '$HOME/.eco/sandbox/bases'...

### ECO:STEP 2/7: 2022-07-30T05:48:30.905Z (Install the Dependencies)
up to date, audited 1 package in 98ms
found 0 vulnerabilities

### ECO:STEP 3/7: 2022-07-30T05:48:31.405Z (Run the Unit Tests)
> bases@0.2.1 test
> node test
All tests passed.

### ECO:STEP 4/7: 2022-07-30T05:48:31.900Z (Run the Unit Tests in Identity Mode)
> bases@0.2.1 test
> node test
All tests passed.

### ECO:STEP 5/7: 2022-07-30T05:48:33.305Z (Run the Unit Tests in Proxy Mode)
> bases@0.2.1 test
> node test
All tests passed.

### ECO:STEP 6/7: 2022-07-30T05:48:34.890Z (Run the Unit Tests With Full Contract Checking)
> bases@0.2.1 test
> node test
$HOME/.eco/sandbox/bases/contract-base.js:1389
  throw new ContractError(`${errorMessage}${nullMessage}${keyMessage}`);
  ^

ContractError [TypeError]: *** SCOTTY-CONTRACT-VIOLATION:
  Predicate `isNumber' not satisfied for value `2'
    blaming: neg
    value type: string
    keys: {0}
    at throw_contract_violation ($HOME/.eco/sandbox/bases/contract-base.js:1389:9)
```

Fig. 2. An example log file, from the bases package, lightly edited to fit on a single page.

of packages continue on to the next step of processing and some are eliminated because Scotty cannot handle them; these numbers are detailed in table 1. These steps and the processing of DefinitelyTyped are implemented in four major software components:

- `Scotty/contract`: a JavaScript library of higher-order contracts for mapping TypeScript types;
- `Scotty/comp`: a compiler that compiles TypeScript type declarations into `Scotty/contract` higher-order contracts;
- `Scotty/pkg`: a software application that compiles and tests DefinitelyTyped packages individually; and
- `Scotty/eco`: a software application that downloads, compiles, and tests batches of DefinitelyTyped packages.

As Scotty processes the packages, the steps provide a coarse grain categorization, but to better understand the problems it encountered, Scotty emits log files as it goes through all of the packages. As an example, see figure 2, showing the log file for the `bases` package. Scotty prints a line starting with `###` showing which step it is on. This package's logfile is a particularly simple and clear one, showing that the package behaved as expected until the contracts were turned on, at which point it tried to use the string `"2"` in a place expecting the type `number`, resulting in a contract violation.

Beyond diagnosis of errors in the types (or in the packages), the log files enable a post-mortem inspection that helps to identify the limitations of the current implementation that could be fixed in order to support more packages. For that, we cluster the log files using regular expressions. This reveals that many packages fail with similar error messages that we group and analyze together. This analysis gives us direction for the next Scotty improvements.

In the following sections we describe each Scotty phase individually, focusing on the reasons why packages fail at each stage and the jobs each of Scotty's components do.

2.1 Initialization

Scotty begins by cloning DefinitelyTyped from GitHub and mining it to find a list of types to examine. Helpfully, the available types all live within a single folder at the repository's root, which we read to proceed. Because DefinitelyTyped changes on a day-to-day basis, we check out a particular git commit of the project before proceeding to make repeated runs of the tool easier to analyze. We start with 8006 packages to consider.

Because running the tool over all the packages can become expensive, a user can also provide Scotty a list of packages to examine. This feature enables an incremental use of Scotty. The list of packages that have changed since the last inspection can be easily computed using git facilities. That restricted list of changed packages can be fed to Scotty and can then be used to recheck only the new package versions. Given the number of packages DefinitelyTyped publishes, the high pace at which packages are changed, the download times, and the checking times, providing this incremental package check is critical for the viability of the system in the long run.

2.2 Can Fetch?

Scotty next looks for the untyped JavaScript code that corresponds to the type declarations by following links to git repositories listed on the npm registry or in the `package.json` declaration files. One can also download the gzipped files that npm hosts for consumption in application code, but those files often lack metadata necessary for running the library's tests correctly. Accordingly, Scotty only considers packages where it can find the original source code.

Unfortunately, `Scotty/pkg`'s attempt to find the untyped package does not always succeed because some declarations do not correspond to any JavaScript packages. For example, the package `@types/elm` provides type declarations that allow TypeScript code to interact with the elm programming language; these types are outside of the scope of our project that focuses on JavaScript and TypeScript.[3] Scotty ignores such packages.

In other cases, the downloads fail due to dangling links, network timeouts, and other such problems. For instance, the git repository associated with the package `convert-string` no longer appears to exist.[4] Downloading packages may also fail due to rate limiting from the npm registry; in the future we may wish to implement a "wait and retry" policy. Overall, in 1560 cases, Scotty failed to download the JavaScript package associated with the type declarations, which is 19% of the DefinitelyTyped packages.

[3] See https://www.npmjs.com/package/@types/elm
[4] See https://www.npmjs.com/package/convert-string

2.3 Can Test?

Ideally, with the source code at hand, one could run the unit tests for a package by invoking the officially supported "`npm test`" command and observing the output. In practice, some libraries do not have unit tests, and some have several dependencies which the tests require to execute correctly. One must therefore download all dependencies by invoking "`npm install`" first. Perhaps surprisingly, several libraries have compilation steps due to the need to support different JavaScript runtimes and module systems. Accordingly, Scotty executes "`npm run build`" to ensure these compilation steps get executed when necessary.

These testing commands happen to fail in many ways. Running "`npm test`" invokes a one-line shell script defined in the package `package.json` file that can effectively do anything, from deleting critical files on the disk to starting up a web server and blocking forever. The risk is similar to that of piping a script on the internet into a shell. Accordingly, we time out our system after two minutes as a simple mitigation mechanism. More fundamentally, the unit tests for the JavaScript might not pass; if so, Scotty cannot provide help, and simply fails.

Scotty detected 1026 cases in which "`npm test`" failed because the JavaScript had no unit tests. Of those, 282 packages had `package.json` files that contained the string "`echo "Error: no test specified" && exit 1`", which is the default message that npm creates when starting a project. Other failures were more subtle; for instance, the package `jexcel` emits the message "`No test files found`" instead, after invoking the "mocha" test runner.

In at least 312 cases, the unit tests failed because they appeared to depend on an external dependency not specified in the `package.json` or present on the host machine. For instance, the package `random-name` tries to invoke the popular `tap` test runner; however, because `tap` is not listed as a dependency on the project, the error message specifies that the file was not found.[5] In another instance, three tests in the package `webpack-plugin-serve` failed because the server on which we ran the tests does not have a browser.

Scotty found 1846 packages with tests that failed at runtime due to semantic or configuration issues. Semantic issues occur because of known, unpatched problems with the test suites; the package `timelinejs3` provides a well documented example of this failure mode.[6] We include the code from `timelinejs3` that causes problems:

```
// these tests fail because the timeline config instantiation is async, and the test
// proceeds before it's ready. I still haven't figured out how to make Jest wait
// until it's actually ready, or maybe there's a different problem?
test("Ensure options is optional", async() => {
    let timeline = await new Promise((resolve) => {
        let tl = new Timeline('timeline-embed', TEST_CONFIG)
        debugger
        tl.on('ready', () => resolve(tl))
    });
    // these tests will fail until we figure out how to deal with
    // the fact that the config creation/setting is async
    // tried some things waiting for
    expect(timeline.config).toBeDefined()
})
```

[5] See https://www.npmjs.com/package/random-name
[6] See https://github.com/NUKnightLab/TimelineJS3

With configuration issues, the tests fail because they require additional set up to run that our system does not provide. For example, the package `asciify` fails because the `README` specifies that before one can develop or test the project, one must clone two separate submodules first. Our system cannot cope with such idiosyncrasies; it assumes that "`git clone`", "`npm install`", and "`npm run build`" provide sufficient setup for testing.

Overall, we found 3384 packages (52% of the packages we could download) that we could test.

2.4 Can Inject?

`Scotty/pkg` next determines whether it can inject compiled code into the untyped JavaScript library without breaking its unit tests. Doing so involves running `Scotty/comp`, which translates TypeScript type declarations into a JavaScript module that wraps the exports of the original library with the contracts provided by `Scotty/contract`. To detect packages that are incompatible with the Scotty rewriting procedure, `Scotty/pkg` first performs a *dry* run. For the dry run, it uses a version of `Scotty/contract` that provides contract combinators that do no checking and do not create any proxy objects. Of course, this means we will not find any errors in such packages, but running the contracts in this mode helps us understand why failures occur.

One common reason for failures at this stage is that `Scotty/pkg` cannot determine where to put its compiled output. Consider, for instance, the package `facebook-locales`, whose `package.json` says the entry to the library exists in a file at the path `dist/index.js` from the root of the repository. This file does not exist because it is the output of a build step; to obtain it, one needs to run `npm run compile` before proceeding. The current implementation of `Scotty/pkg` does not know how to handle cases where the entry point is missing by default, so it gives up on the package. Scotty reports that analogous situations occurred in 478 other cases.

Another relatively minor but interesting source of error involves conflicts between the two predominant module systems in the node ecosystem. Most packages on npm as of now tend to use CommonJS modules instead of the ESModule system encoded in the official ECMAScript specification; accordingly, our compiler currently targets CommonJS. However, we detected 75 cases in which this decision prevented our system from interoperating with the package. Future versions of `Scotty/pkg` and `Scotty/comp` will have to handle ESModule as well.

Perhaps surprisingly, linting and code coverage tools are more frequently responsible for preventing Scotty from injecting its code into a package. Because of its dynamic and permissive design, JavaScript is considered an error-prone language. For instance, Frolin S. Ocariza Jr. et al. (2011) have shown that in JavaScript, even mundane syntax errors can cause many dynamic errors. To mitigate these problems developers frequently use linters (Beller et al. 2016; Tómasdóttir et al. 2017, 2020). We have observed that many packages run styling tools like "eslint", "jshint", and "standard" before or after invoking unit tests.[7] These different tools enforce different conventions around whitespace, semicolons, line lengths, and other such concerns, but any deviation from the package's norms can trigger a failure. Adding lines of code that the library's unit testing system detects are uncovered by the tests can also cause errors.

At one point, linting errors caused over 1000 different packages to fail at this step. As a way of mitigating the problem, Scotty preemptively stubs out several popular linters with a shell script that never fails. Unfortunately, linters integrated directly into testing tools and linters that we have not recognized in advance do not work with this strategy. Consider, for example, the package `@happi/crumb`, which relies on the `lab` testing library.[8] Running `lab` from an npm script causes `eslint` to report a number of style violations to which our new code does not conform. Our system

[7] See https://standardjs.com , https://eslint.org, and https://jshint.com/about/ for more details.
[8] See https://www.npmjs.com/package/@hapi/crumb

cannot detect that the test suite succeeded despite these problems, so Scotty reports a failure. Here is the precise report of the package, showing that the tests pass and even achieve 100% test coverage:

```
19 tests complete
Test duration: 311 ms
Assertions count: 91 (verbosity: 4.79)
Leaks: No issues
Coverage: 100.00%
```

but then the output shows that the style is incorrect:

```
Line 1: strict - Use the global form of 'use strict'.
Line 1: @hapi/no-var - Unexpected var, use let or const instead.
Line 1: quotes - Strings must use singlequote.
```

Even after disabling the most common linters, we found 41 packages that failed because Scotty's compiler uses double quotes and 20 packages that failed because it generates too many semicolons. These 61 failures were straightforward to detect (but not remediate) because they all shared similar error messages; however, our search misses linters with more idiosyncratic diagnostics. For example, 4 packages failed because some of the contracts our system generated used capital letters, which violates style conventions used to distinguish constructor functions from ordinary functions, from the era before JavaScript introduced the `class` keyword. Problems such as these can be difficult to spot among the sea of other errors, both with various other packages and Scotty itself.

After accounting for the errors above, we determined that our system can inject code into 2417 different packages, that is 71% of the packages for which Scotty found an operational test suite.

2.5 Can Interpose?

Our next step involves checking whether using proxies to perform contract checking causes problems. As in Williams et al. (2017)'s work, object equality is a problem for contract checking based on proxies. In a nutshell, the problem is that an object proxy and the original object are not the same under the `===` equality relation in JavaScript. Accordingly, if a value flows through a contract and gets a proxy, it may cause code downstream to behave differently if the proxied value flows into such an equality test where the unproxied value normally would have.

Proxies also cause problems with built-in classes like `Map` because calling their native methods on a proxy can cause an error at runtime. Consider what happens, for example, when creating a proxy named `MySet` of the native `Set` class that intercepts none of the set's handlers. Trying to add an element to `MySet` using `Set.add` triggers an exception because JavaScript does not support calling the `add` method on proxied Sets, only nonproxied ones.

```
const MySet = new Proxy(new Set(), {}); // a proxy with no interposition
MySet.add(3); // This causes a crash at runtime.
```

To automatically find libraries in which the proxy semantics causes problems, `Scotty/pkg` uses a *neutral* variant of `Scotty/contract`. In this version of the library, contracts are implemented by means of neutral proxies that do no checking (but may create other proxies for higher-order types). They are only used to detect situations where proxies interfere with the package's semantics. This

is an essential step as it let us evaluate how practical it is to use proxies for detecting type errors. The lower the number of failures, the more viable the approach. By contrast, the higher the number of failures, the less practical the approach.

We found that in only 348 cases, proxies did cause interference (14% of the packages for which Scotty can inject code). Of those cases, 19 were due to failures involving calling built-in methods. Another 18 errors were caused due to proxies causing the system to time out, possibly because of well studied performance problems (Greenman et al. 2019; Takikawa et al. 2016). These 348 are incompatible with the technique Scotty relies on to check type correctness.

In total, we found that Scotty could interpose on 2069 packages, 61% of the packages for which Scotty found an operational test suite.

2.6 Contracts Pass?

In the last step, we use the real `Scotty/contract` implementation of the contract library that performs the actual checking in the proxies. If violations do not occur, Scotty reports that it could find no errors in the type declarations; if violations do occur, Scotty notes that there may be a problem with the types. Unfortunately, Scotty does not offer a fix, only some debugging information about which type the failing dynamic check comes from. At this point, a programmer must investigate further.

In our experience the amount of effort to write a pull request to offer a fix varies widely. Sometimes the contract error message leads us directly to the bug but, in general, it can be arbitrarily difficult to determine if there is a bug in the tests, the code, or the types. The worst one took many hours; it was from the `ical` package. The actual bug[9] is that a test case was checking to ensure an error happened while accessing the nonexistent field `fromURL`, but the types were written as if `fromURL` were a legitimate function. The difficulty finding the bug was that the error was being thrown in an asynchronous test, making it difficult to pin down exactly which code was signaling the error.

Worse, some kinds of unit tests create false positives with our system. A particularly perplexing occurrence is that some tests check that certain functions throw exceptions with specific error messages when passed arguments with the wrong types. In these cases, Scotty's messages interfere with the code. Consider, for example, the `string-similarity` package's `find-best-match` function.[10] It takes a string for its first argument and an array of strings for its second argument. The package tests that the function throws exceptions when provided illegal values, but `Scotty/contract` breaks these tests because it detects the illegal values before the function is invoked and it generates its own error message. The tests expect errors to match the string `"Bad arguments: First argument should be a string, second should be an array of strings"`, whereas `Scotty/contract` emits different messages like `"Wrong argument count 0/2"` and `"Predicate 'isString' not satisfied for value '2'"`. Currently, Scotty is unable to automatize the detection of false positives of this variety. Currently, they require one to go through all 567 packages for manual inspection.

2.7 Broken Contracts, Unsound Types, and Too Many Linters

JavaScript the language causes relatively few problems for our system. Combined, the interference problems known to occur with proxies and the presence of two incompatible module systems account for only 423 packages that Scotty cannot handle. While this number is nontrivial, compared to the hundreds of problems caused by malfunctioning test suites, fussy linters, code coverage tools, and other such auxiliary concerns, it is relatively small. The npm ecosystem poses us the most significant engineering challenges. Although we believe a production-quality version of Scotty is

[9] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54744
[10] See https://www.npmjs.com/package/string-similarity

feasible, especially given the value it provides, there is no question it would need to cope with the immense variety of all of the different packages.

3 DYNAMICALLY CHECKING TYPES

Following early work on natural gradual typing (Gronski et al. 2006; Tobin-Hochstadt and Felleisen 2006) and higher-order contracts (Findler and Felleisen 2002), Scotty compiles types to contracts using two separate components. `Scotty/comp` is our type to contract compiler; it uses Babel[11] to parse TypeScript declarations. `Scotty/contract` is our contract-checking runtime system; it uses JavaScript's proxies (Ecma International 2015; Van Cutsem and Miller 2013). `Scotty/comp` generates code that maps the information from the TypeScript type declarations onto combinators provided by `Scotty/contract`, the contract checking library. Together `Scotty/comp` and `Scotty/contract` try to ensure that mistakes in the types raise errors instead of passing silently.

The remainder of this section explains how the contract checking works and discusses shortcomings of the checking.

3.1 Checking Basic Types

As a first example, consider this JavaScript function:

```
function swap(result) {
    if (result === "pass")
        return "fail";
    return "pass";
}
```

Let us assume an associated TypeScript type declaration like the ones in DefinitelyTyped:

```
type Result = "pass" | "fail"
declare function swap(result :Result) :Result;
```

Scotty compiles the TypeScript declarations into contracts and then uses them to do dynamic checking on the JavaScript version of `swap`.

In general, Scotty creates one function for each different type. These functions all have the same signature: they accept arbitrary values and then either signal errors (if the input doesn't match the corresponding type) or the return the original value (if it does). For instance, for the TypeScript type `Result`, Scotty creates a function that behaves like this one:

```
function Result_ctc(x) {
    if (x == "fail") return x;
    if (x == "pass") return x;
    throw new ContractError("not a Result")
}
```

If the type is an object type or a function type (*i.e.,* the type has a behavior), then the function that Scotty creates uses a JavaScript `proxy` to delay the checking until an appropriate moment in

[11] See https://babeljs.io

the evaluation of the program. This function behaves like the one that Scotty creates for the type declaration of `swap`:

```
function Result_to_Result_ctc(f) {
  const handler = {
    apply: function(target, self, args) {
      if (args.length === 1)
        return Result_ctc(target.call(Result_ctc(args[0])))
      throw new ContractError("Incorrect number of arguments");
    }
  }
  return new Proxy(f, handler);
}
```

The actual functions that Scotty generates are not so neat and tidy as this one, but these capture the key ideas of the strategy that Scotty is using and they behave identically for these specific type declarations (except that Scotty's error messages have more information).

There are a number of improvements required to generalize these ideas to support the full language of TypeScript. Currently, Scotty can handle primitive types such as numbers, strings, arrays, symbols, null, undefined, and Nodejs primitives types such as files, streams, events, etc. All of the Nodejs primitives are handled using just first-order checks, as the `Result_ctc` does.

Scotty can also compile array types into array contracts, and interfaces into object contracts. These contracts are similar to function contracts, in that they create proxies to delay checking. Scotty also has an implementation of union and intersection contracts; and union types compile into union contracts and overloaded functions compile into intersection contracts; Scotty supports both variadic and optional arguments in this manner. The support for intersection and union is the most complex aspect of the contracts; our implementation follows ideas in Castagna and Lanvin (2017)'s, Keil and Thiemann (2015)'s, and Williams et al. (2018)'s work.

3.2 Checking Class Types

Classes are also a complex aspect of Scotty. First, let us recall their main characteristics. Classes were added to JavaScript in version 6 and, although they bring a class-based programming flavor to the language, they are merely syntactic extensions to functions. A JavaScript class is compiled down to a JavaScript function. Class instances are regular JavaScript objects. Instance properties are implemented as regular object properties and methods are implemented as properties of the prototype object of the function representing the class. Indeed, JavaScript classes and functions can be mixed without limitation. TypeScript enforces two important constraints. First, a TypeScript class cannot be used as a function, *i.e.,* a class identifier cannot be called via function application. Conversely, a TypeScript function cannot be used as a constructor, *i.e.,* with `new`. That said, it is possible to construct a type for a single value that behaves as both a function and a class.

When `Scotty/comp` compiles a class declaration, it generates a contract for the class instances, another one for the class constructor, and one contract for each of the class methods. Let us illustrates this process with the compilation of the `Point` class:

```
export class Point {
  x: number;
```

```
  y: number;
  constructor(x: number, y: number);
  toStr(): string;
}
```

The contract `Scotty/comp` generates for representing instances of `Point` is:

```
var Point$InstanceContract =
  CT.CTInstance(
    "Point",
    {
      x: CT.numberCT,
      y: CT.numberCT,
    },
    {
      toStr: CT.CTFunction(
        CT.CTRec(() => Point$InstanceContract),
        [],
        CT.stringCT
      ),
    },
    originalModule.Point,
    undefined
  );
```

There are a number of things to unpack in this code. First, the contract is generated by a call to `CT.CTInstance`, whose first argument is a string, naming the class. The second argument is a record giving the contracts of the fields. Instances of the `Point` class have two fields `x` and `y` whose values are numbers, enforced by the builtin `Scotty/contract` contract `CT.numberCT`, which uses a technique similar to `Result_ctc`, as above. The third argument of `CT.CTInstance` is an object whose keys are the methods and whose values are the method's contracts. TypeScript does not permit class methods to be invoked on arbitrary objects; it requires that they receive a proper instance of the class as their `this` argument. The contract enforces this constraint by using the `Point$ClassContract` passed as the first argument to the `CT.CTFunction` constructor. `CT.CTFunction`'s first argument is used to check the `this` argument during function invocation. The `Point$ClassContract` also ensures that `Point` instances have at least two properties `x` and `y`, that the method `toStr` is included in their prototype object, and that they are all genuine instances of the `Point` class.

Because class definitions use recursion to refer to the type of `this` in methods and constructors, and because class definitions are, in general, mutually referential, `Scotty/comp` uses `CT.CTRec` recursive contract constructor to tie the self-referential knot. `CT.CTRec` is wrapped around all references to identifiers that the compiler generates, ensuring that the references to the identifiers are evaluated only after all the contracts are defined, no matter what order they appear in the program. `CT.CTRec` accepts a thunk that returns a contract and internally it invokes that thunk and ties the recursive knot the first time the contract is checked, at which point all of the contracts have been defined.

Also, `Scotty/comp` generates a contract for the function that allocates and constructs `Point` instances. This one is generated from the prototype of the class constructor:

```
var Point$ClassContract = CT.CTClass(
  CT.CTRec(() => Point$InstanceContract),
  [CT.numberCT, CT.numberCT]
);
```

This contract enforces that the JavaScript generated `Point` function is used only in `new` expressions and that it is always invoked with two arguments that are both numbers. This is the purpose of the `CT.CTClass` contract constructor. It is very similar to `CT.CTFunction`, differing only by enforcing the type of `this` and by checking that no value is returned from the constructor.

Subtyping introduces a subtle point for the checking in these contracts and it is a place where `Scotty/contract` might miss some errors. To understand how, consider this pair of classes.

```
class C {
  cache;
  constructor() {
    this.cache = false;
  }
  getValue() {
    if (!this.cache) this.cache = this.computeValue();
    return this.cache;
  };
  computeValue() { return 1; };
}

class D extends C {
  aField = 1;
  computeValue() {
    return this.aField;
  }
}
```

The class `C` is intended to be illustrative of a caching protocol where the `getValue` method returns (and possibly computes) the cached value, with an extension point `computeCache` that is intended to be overridden to compute the cached value. In this code, `D` overrides `computeValue` and, as part of the computation, uses the field `aField`.

With types that simply declare that the cache is a number, we generate contracts similar to the ones described above, but specific to these fields and methods. The subtle point about these contracts is how the wrappers get created. In particular, when we create an instance of `D` and invoke the `getValue` method, the contract for `C` wraps the implicit `this` argument. Since there is no `aField` in `C`, the eventual reference to `aField` in `D` would signal an error.

The way we avoid this problem is to add a check: if the field we're trying to access actually exists in the object (but is not described the contract), we allow it in the `get` handler of the proxy created for the contract. In principle, this choice can cause the contract checking to miss some errors but since this problem creates false negatives and does not give up basic checking, we opted

to include it in `Scotty/contract`. In future work, we hope to devise a better strategy for checking class contracts that would catch errors that this strategy might miss.

3.3 Missing Checking

Some type constructs and features are still missing, which causes Scotty to be unable to handle some packages. Because of the incremental nature of Scotty, we add support for contracts and fix known problems with the existing support as it becomes important. One general, and simple, example of this approach is our catch-all contract that does no checking; it is implemented simply as the identity function and we use it for any types that Scotty does not handle. There are three notable shortcomings beyond the one described at the end of the previous subsection:

- `Scotty/contract` and `Scotty/comp` do not handle generic types; any use of a polymorphic variable is compiled into a contract that does no checking. We believe we could add support for polymorphic contracts along the lines of prior work (Ahmed et al. 2009; Ahmed et al. 2017; Guha et al. 2007; Matthews and Ahmed 2008), following the implementation in Racket's contract system.[12] It is difficult to know how many more errors we will uncover if we add support for generics until it is implemented, especially because contracts for generics are likely to introduce interference.
- `Scotty/contract`'s implementation of union and intersection contracts is incorrect in some situations. In many common situations, the intersection and union contracts are correct, but the implementation is simpler than the strategy presented in Keil and Thiemann (2015) and Williams et al. (2018).
  Roughly speaking, the problem is that our implementation does not delay the decision of which branch of the intersection or union long enough (in some situations). In general, this can lead to false positives, where the contracts signal a violation because they commit to a particular branch of the union or intersection too soon. We discovered this by using the test suite from Williams et al. (2018)'s work; the discrepancy has not yet come up with any TypeScript package.
- The current implementation of `Scotty/comp` requires all the TypeScript type declarations to be included in a single file. Supporting imports and exports in TypeScript declaration files requires additional engineering effort.

Unfortunately, in addition to these unsupported features, for various reasons, there are a number of situations where the contracts `Scotty/contract` generates are incorrect in the sense that they might generate false positives. Overall, false positives are rare but possible. The two following sections detail the situations where they occur.

3.4 Proxy Interference

In general, JavaScript proxy objects are not transparent. That is, in a few situations, introducing proxies changes the behavior of a program. Scotty supports a mode where it creates all of the proxy wrappers but the wrappers never signal any errors. We use this mode to determine if there is any interference between Scotty and the package we are testing, as discussed in section 2.5. This let us automatically separate packages that cannot be tested by Scotty and avoids false positives.

While we are not sure of the full set of causes of proxy interference in uses of Scotty, we observed three: the JavaScript equality relation `===` is not preserved by proxies, primitive methods accept only primitive objects (and not proxied primitive objects), and some packages time out only when contract checking is enabled in proxies. This last problem could probably be improved by adopting optimization techniques developed for higher-order contracts (Bauman et al. 2017; Feltey et al.

[12] https://docs.racket-lang.org/reference/parametric-contracts.html

2018) but because of the small number of packages actually hitting the timeout limit (18 packages), we have postponed this additional implementation effort.

3.5 TypeScript Unsoundness

TypeScript's deliberate unsoundness makes it difficult to generate contracts that faithfully mirror the types. TypeScript's function compatibility rules are the source of one problem that we encountered. To understand the specific unsoundness, consider a function named `runOneTwoThree`, a higher-order procedure that takes some function and invokes it with the numbers one, two, and three. Next, consider a function named `addTwo` that takes two numbers and sums them together. Invoking `runOneTwoThree` with `addTwo` does not raise a type error, even though calling `addTwo` with three arguments in other circumstances would cause compilation to fail because of the arity mismatch (The TypeScript Handbook 2019).

```
type TakesThreeArguments = (a: number, b: number, c:number) => number;

const runOneTwoThree = (input: TakesThreeArguments) => {
  return input(1, 2, 3);
}

const addTwo = (a: number, b: number) => a + b;

// This code passes TypeScript's type checker.
runOneTwoThree(addTwo);

// This line is a type error
addTwo(1, 2, 3);
```

These peculiar TypeScript typing rules are motivated by the ubiquitous JavaScript `map` operator. The function `map` receives is applied to three values: an array element, an array offset, and the array itself. However, most user codes passes a function of a single argument to `map`. If TypeScript were to use an ordinary typing rule, it would raise errors for all these situations. As `array` is extensively used, this would limit too severely the connection between the two languages.

Unfortunately, the contracts that our compiler uses cannot distinguish between the two different calls to `addTwo`. Our experience is that the higher-order use of a function like `addTwo` is far less common than the direct use, so the contracts always check that the number of arguments passed to a function with a contact matches exactly the number specified in the type (as the example code at the start of this section does). Accordingly, the higher-order use in the example signals an error when used in our system but does not in TypeScript.

We encountered this problem with the package `doge-seed` while experimenting with our system. The library exports a function with a type that optionally takes a single argument, but the test suite uses it in the point-free style with `Array.map`, which passes three arguments to its function. These caused our contracts to report an arity-related error.

4 ERROR ANALYSIS

Using Scotty enabled us to submit accepted pull requests to Definitely Typed efficiently. Over the summer of 2021, we maintained the rate of 8 new pull requests per week, based on the results of running Scotty. The pull requests we opened are not a random sampling of the ones that Scotty

finds today, however, for two reasons. First, we submitted pull requests as we were also developing the tool and, as time progressed, Scotty got better at handling packages, meaning that there were more contract violations to choose from. Second, some of the contract violations are easier than others to turn into suggestions for fixes. Indeed, different errors took wildly different amounts of time to understand, based on the complexity of the package and its types.

The remainder of this section analyzes all of the pull requests we submitted to understand the nature of the failures our system could uncover. We analyze each pull request twice, once *by semantic issue* and once *by external factor*.

When grouping packages by semantic issue, we examine the particular contract violation that created a bug. This categorization involved analyzing the blame message emitted by the contract system and categorizing the packages accordingly. Several of the type errors stemmed from the same kind of mistake related to one feature of the type system, which indicates that the semantics of the feature may be difficult to use correctly.

By contrast, when analyzing the external factors related to the bugs, we focus on how changes outside the type declarations themselves created problems. For these factors, we examined the git repositories of both the underlying JavaScript and the types to understand how differences in each may have created problems. These mistakes shed insight into how well the language's type system copes with the shifting requirements and outside world that software engineers face each day.

4.1 By Semantic Issue

We have organized the semantic failures of the 100 packages into the table below.

Reason	Count	Package examples
Record Mismatch	41	`checksum`, ...
Function Arity	27	`mouse-event-offset`, ...
Broaden Type	15	`natural-compare`, ...
Null and Undefined	9	`humanize-ms`, ...
Other	8	`ical`, `express-ejs-layouts`, ...

We detail each category in the following subsections and we present examples and accepted pull-requests we have submitted.

4.1.1 Record Mismatch. Incorrect interface types accounted for the most substantial proportion of problems we uncovered. An interface type in TypeScript describes the structure of a JavaScript dictionary by enumerating the names and types of its keys. Any mismatch in the type's shape results in a type error, although the language does offer facilities for marking some keys as optional or permitting arbitrary dynamic keys. We found several errors where the JavaScript code used dictionaries with too many, too few, misnamed, or incorrectly typed keys.

The library named `checksum` provides an example of an error that fits within this category. The library exports a function that a programmer can point at a file to get a string that represents a checksum. To configure the method, one can pass a dictionary with a key named `algorithm` that specifies the hashing technique to use and a key called `encoding` that specifies the encoding of the underlying file. While the interface in the type declarations had the `algorithm` key, it lacked the `encoding` key, which meant that a programmer could not use that feature safely from within TypeScript. Scotty reported that problem with a contract violation when invoking the function. Adding the missing key to the type declaration fixed the problem.[13]

4.1.2 Function Arity. Challenges with function arity accounted for another substantial portion of the errors we patched. TypeScript generally requires that programmers invoke functions with all of

[13] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54805

their declared parameters; leaving out or putting in too many arguments is generally a compile-time error. By contrast, JavaScript takes a more liberal approach. If a function gets called with more arguments than it was declared with, they are ignored, and if it gets called with less, it binds the missing parameters to the special `undefined` value. JavaScript programmers frequently overloaded their functions in subtle ways which the TypeScript declarations did not capture.

One library named `mouse-event-offset` typifies the kinds of mistakes within this category. The library exports a function for calculating the X and Y offsets of an event from the browser based on the target of the event. Optionally, one can pass the function a second argument that is another element in the DOM to instead compute the X and Y offsets from that element. However, the TypeScript type did not account for this second argument. By adding this parameter to the function type, TypeScript programmers became able to access this feature of the package.[14]

4.1.3 Broaden Type. On several occasions, the type declarations for a package proved overly restrictive. In these cases, the underlying JavaScript implementation invoked its code with types unsupported in the TypeScript declaration, which indicated the need for a union type to support the semantics of the package. The `natural-compare` library, for example, exports a comparison function meant for sorting that supports both numbers and strings. However, the TypeScript code only allowed programmers to call the method using strings. To help TypeScript programmers invoke the function safely from more contexts, we submitted a patch changing the input argument of the package to a union of strings and numbers.[15]

4.1.4 Null and Undefined. Unexpected occurrences of `null` and `undefined` also created problems. Functions often accepted or returned `null` or `undefined` in unusual situations; one can configure the TypeScript compiler to accept those values in place of any other, but DefinitelyTyped requires type declarations to adhere to a strict mode which makes the presence of those values bugs. The package `humanize-ms` serves as an example of this kind of problem. The library exports a function that tries to parse strings like `"1s"` to numbers. When the parser fails, instead of throwing an exception, it returns `undefined` to the caller. The type declarations did not account for this `undefined` situation, so making the return type the union of `number` and `undefined` solved the issue.[16]

4.1.5 Other. The remaining errors did not fit neatly into any of the above categories. For example, one package named `ical` had type declarations for a function that did not exist. Another library called `express-ejs-layouts` had types for a higher-order function when, in fact, the module exported only a first-order function.

In general, the semantic bugs found were easy to fix because the correction merely consisted of extending or refining a type declaration. The difficult step was mostly to realize that there was a bug somewhere! This is the job Scotty is designed for.

4.2 By External Factor

We have organized the external factors leading to contract violations below.

Reason	Count	Package examples
Documentation Challenges	47	`gaussian`, ...
Misimplementation	30	`envhandlebars`, ...
Platform Changes	23	`pwd-strength`, `http-codes`, ...

[14] See https://github.com/mattdesl/mouse-event-offset/tree/82805f285f8483f2373d2bf20f62f9eb8d79f74d
[15] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54185
[16] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54608

4.2.1 Documentation Challenges. Issues regarding the documentation of JavaScript libraries accounted for the most significant portion of errors our system discovered. In these cases, a library contained behavior recorded only in its source code, test suite, or git history that the type declarations should (but did not) capture. Without the help of a contract system and its error messages, a programmer would need to discover these cases by carefully reading through the library's source while writing the type declarations.

The library `gaussian` serves as an illustrative example of this problem. It exports a function named `random` which, according to the `README` file, takes a single numeric parameter. However, a JSDoc comment in the source code reveals that the function takes a callback as its second argument that one can use to configure the system of random number generation that the method invokes. The tests exercise this option, but the documentation does not mention it.

Note that before our pull request to fix this issue,[17] both the package and its type declarations were being maintained by the community. JavaScript support for the callback parameter got added in July of 2021, and the type declarations had received unrelated bug-fixes on October 1st of the same year. Developers were actively trying to keep the types in sync in this case, but the lack of documentation made doing so challenging.

4.2.2 Missimplementation. Mistakes in the initial implementation of the TypeScript declarations accounted for the second-largest proportion of errors. In these cases, the types for the package appear to have never matched the underlying JavaScript code. As the results of some mistakes in JavaScript are difficult to catch, these errors appear to have gone unnoticed until our patches.

One illustrative example of this error involved the `envhandlebars` package. The type declarations contained an interface type with a key named `arrayEnabled` meant to correspond to a boolean or `undefined`; however, the JavaScript code looked for a key named `arraysEnabled` instead. As the underlying JavaScript did not need the key to be present to work, this typo went unnoticed until Scotty revealed it.[18]

4.2.3 Platform Changes. The remaining errors we analyzed are most attributable to platform changes. In these cases, some external change rendered the type declarations invalid, like a modification to the underlying JavaScript code. The library `pwd-strength` contains an especially striking example of this type of problem. On March 26th, 2021, the library authors added a feature that let users add a `minNumberChars` key to a configuration dictionary. The day before, on March 25th, an interface describing the shape of that dictionary got merged into DefinitelyTyped; this type was only accurate for a single day. The incremental analysis of Scotty makes finding this sort of desynchronization between the implementation and the type declaration easy.[19]

Note that factors other than changing JavaScript code can cause the types to become inaccurate. For example, the package `http-codes` used to export a key named `UNORDERED_COLLECTION` because node's http package exported a dictionary that mapped the key `"Unordered Collection"` to the string `"425"`. However, the http library bundled with node changed, and the key's name became `"TOO_EARLY"` instead. This change rendered the previous type declaration invalid, even though neither the type nor the JavaScript changed.[20] This example shows the benefit of approach that consist of running actual code in an operational system. Any change to the underlying execution platform, in this case Nodejs, is handled by Scotty, without requiring sophisticated analysis or significant effort.

[17] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/58747
[18] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/55888
[19] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/55722
[20] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54281

4.3 Deliberate Unsoundness

In two situations Scotty revealed problems that the implementors did not consider to be errors or preferred not to fix. That is, the maintainers of DefinitelyTyped rejected our patches because they preferred the existing unsound type declarations. Their motivations for each differed and provided insight into the ways in which programmers use the type system.

The first case of deliberate unsoundness involved a package named `caseless`. This package exported an interface type with methods like `get` and `set` that represented actions one could perform on a specific dictionary. At runtime, that dictionary was stored on the object under the key `_dict`; a programmer could access the key in JavaScript directly instead of using the accessors provided by the package. Since the key `_dict` was not present on the interface type, the TypeScript type was not faithful to the JavaScript implementation, but it enforced encapsulation desired by the TypeScript maintainers of the package. Accordingly, the maintainers of the type declaration rejected our patch to add the key.[21]

The second case of deliberate unsoundness involved a callback in the `ffprobe` library. The package contained a function named `getInfo` whose last argument was a function of two arguments: `err` of type `Error` and `info` of an interface type called `FFProbeResult`. In the callback type, neither of those two values could be `null` or `undefined`; at runtime, however, one of them always was. One can most easily understand the correct type as a union of two function types. The first function type contains one argument, error, that is always present; the second type takes two arguments, the first of which is always the value `null`, and the second of which is always of type `FFProbeResult`.

Switching to the sound type would require users of the package to check whether the `err` parameter is `null` before using the data in the `info` parameter. However, such a change would cause existing TypeScript code that does not already perform this check to no longer compile. Because of this backward compatibility constraint and the fact the package had not received active development, the maintainers of the package opted not to accept our pull request.[22]

The cases reported in this section, although rare, show that to be effective, an approach needs to combine an automatic bug detection that isolates each potential error and a manual inspection of suspicious packages that confirms the error and that supports ad hoc treatment of subtle situations. Beyond the need for manual inspection, the tool also needs support so programmers can declare that specific type mismatches are false positives, either declaring that specific tests should not be run when checking the contracts, or by declaring that specific types should be ignored, and translated into contracts that do no checking. Overall, our experience suggests that these declarations, while necessary for a fully-featured tool, are needed only rarely.

5 RELATED WORK

Discrepancies between TypeScript type declarations and JavaScript implementations has long been identified as a source of bugs. The popularity of the DefinitelyTyped repository has exacerbated the problem. Improving the correctness of TypeScript types with respect to JavaScript package implementations is an active research field. In this section we discuss some of the most related threads of work.

Scotty fundamentally builds on Proxy objects (Van Cutsem and Miller 2013), introduced in ECMAScript 6 (Ecma International 2015), as they provide a low-level mechanism for portable higher-order contract checking, but JavaScript Proxy objects are not transparent and are thus subject to the *interference* problem (Keil and Thiemann 2013); see also section 3.4. Keil et al. (2015)

[21] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/55992
[22] See https://github.com/DefinitelyTyped/DefinitelyTyped/pull/54866

show how to mitigate that problem by introducing transparent and opaque proxy objects but as they have not been adopted by the JavaScript standard, Scotty does not use them.

Williams et al. (2017) studied the interference problems with JavaScript proxies when using contracts for checking DefinitelyTyped interfaces. They built TPD, a tool based on the polymorphic blame calculus, for monitoring JavaScript libraries and TypeScript definitions. They found that interference occurred "in an intolerable number of cases". Our own study concludes otherwise because we have found that only 348 packages (or 14% of the packages that could otherwise have been checked at that stage) cannot be verified because of proxy interference.

TPD's opposite conclusion may be because the landscape of types has changed since then (as reported by TPD's authors, there were only 500 packages on the DefinitelyTyped repository at that time) and it may also be because we do not check polymorphic types. Fundamentally, it is surprising that proxy interference and the unsoundness of TypeScript causes Scotty relatively few problems. Instead, we found that the wide variety of build steps, testing frameworks, and linters in the JavaScript ecosystem created significant engineering challenges for researchers and developers who seek to analyze arbitrary libraries from npm in the large. Happily, simply not checking polymorphic contracts and just ignoring the packages with proxy interference still enables us to find many bugs, which is our goal.

There has been a long line of research in using gradual typing (Gronski et al. 2006; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006) to combine statically and dynamically typed languages that reached a climax with Takikawa et al. (2016)'s famously provocative question, "Is Sound Gradual Typing Dead?", a question raised because of the intrinsic and severe impact on program performance. Many responses were published, all minimizing the slowdown incurred by gradual typing (Bauman et al. 2017; Feltey et al. 2018; Moy et al. 2021; Muehlboeck and Tate 2017). Scotty, in contrast, uses gradual typing only for checking type validity but not for running the production implementation. In this context, the performance of gradual typing is not crucial as long as the incurred performance penalty does not prevent running the tests. Our experience is that, at least for debugging and testing purposes, gradual typing is alive and well!

While Scotty is based on techniques behind one strategy for checking gradual types, there is a second strategy that is in active use in Reticulated Python (Siek and Taha 2006; Vitousek et al. 2014). We were not able to leverage this strategy, unfortunately, because it relies on adding checks into typed code. In our situation, we typically have only type declarations, an untyped JavaScript library, and an untyped test suite.

Static analysis and type inference have also been used to detect TypeScript declaration inconsistencies (Feldthaus and Møller (2014)'s TSCheck) but it was soon established that the static analyzers known at the time, alone, were not precise enough and a new direction based on dynamic checking was needed. This new approach proved to be more robust, that is yielding fewer false positives and more true positives (Kristensen and Møller (2017b)'s TSTest). Recent studies continue in this direction with improved static analysis and support for program evolution (Kristensen and Møller 2017a). This approach successfully finds mismatches between the TypeScript and JavaScript but, with only 50% coverage of the library code, it may leave many errors undiscovered. Recently, there has been a revival of static analyzers. Kristensen and Møller (2019) present the ReaGenT system, based on the JavaScript TAJS static analyzer (Jensen et al. 2009) adapted to TypeScript and per-module analysis. It aims to *guarantee that it finds all possible type mismatches* between type declarations and JavaScript implementations that realistic clients may encounter. The notion of *realistic clients* builds on the idea of *most general clients*, an intellectual tool that has been invented for the static analysis of dynamic programming languages (Rinetzky et al. 2007). Static analyzers are also facing another challenge: it is not enough for them to analyze the packages and their associated types; they must also model the execution platform and its libraries. Otherwise, some

bugs would not be found, such as the one reported in section 4.2 due to an incompatible evolution of Nodejs. Modeling the execution platform precisely is a daunting task. The approach promoted by Scotty that relies on actual execution by the platforms sidesteps this difficulty.

To support an unrestricted and unconstrained JavaScript interface, TypeScript is intentionally unsound (Bierman et al. 2014). It relies on an erasure semantics for a smoother integration with JavaScript, meaning that types are not checked during execution, and thus the JavaScript implementation might diverge from its type specification. Several projects attempt to build safe execution for TypeScript. Instead of checking or enforcing the consistency of TypeScript declarations, researchers have considered building a sound version of TypeScript. Safe TypeScript (Rastogi et al. 2015) is a TypeScript variant that enables better static error detection and checks types during execution. Type checks are implemented by a dedicated compiler. This technique enables safe and efficient execution of TypeScript programs, but it is unsuitable for our approach because it cannot cope with unannotated JavaScript packages.

More recent studies have explored the design and implementation of platforms going beyond individual package testing and instead handling bulk analysis of the whole DefinitelyTyped repository. Cristiani and Thiemann (2021)'s tool `dts-generate` is a pioneer in this category. It automatically generates type declarations whose precision can be compared to those provided by the package developers. Specifically, `dts-generate` extracts the examples from the package `README` files, and executes them against an instrumented version of the library implementation. The instrumentation is based on the Jalangi tool (Sen et al. 2013). It enables executions to gather dynamic data flow information about function calls, function results, binary operations, object accesses, etc. These pieces of information are used to generate type declarations. These declaration files are compared to the original ones using the `dts-compare` tool. The framework has been applied to the DefinitelyTyped repository and 249 packages were sufficiently documented for `dts-generate` to generate types. The approach embodied in `dts-generate` is complementary to that of Scotty: `dts-generate` uses the package examples to generate correct by construction type declarations. Scotty uses the package test suite to verify that the type declarations are correct. The smaller number of packages that `dts-generate` handles and the difficulty of extracting relevant examples from the `README` files suggests that generating types is a harder problem and, unsurprisingly, the whole process is less robust than simply testing declarations. Scotty is synergistic with `dts-generate`, as it can help verify the correctness of `dts-generate` types by re-testing the package with the newly generated type interface.

Tsinfer and Tsevolve (Kristensen and Møller 2017a) are two tools designed to assist the construction of new TypeScript declaration files and support co-evolution of the declarations and the implementations. Tsinfer accomplishes a similar task to that of `dts-generate` but it uses static analysis instead of instrumented executions. The tool Tsevolve compares two declaration files to detect situations where a change in an implementation might also change the type of the visible interface. The experimental results presented in the paper suggest that the analysis is fast enough to analyze all the DefinitelyTyped packages but there was no attempt to use Tsinfer on the whole repository and its authors have since declared Tsinfer obsolete (Kristensen and Møller 2019). NL2Type (Malik et al. 2019) is another tool akin to Tsinfo and `dts-generate`. It automatically generates type declarations for JavaScript packages but as it uses learning techniques applied to the textual information available with each package (function and variable names, comments, etc.), the validity of its production is not guaranteed and its output requires human verification. Consequently, NL2Type is not suitable for automatic handling of large package repositories such as DefinitelyTyped.

Tapir (Møller et al. 2020) is another tool that finds incompatibility between two versions of a JavaScript library. Like Tsevolve and `dts-compare`, it can be used to detect inconsistencies introduced

by a package's evolution, a source of many errors due to the constant and rapid evolution of npm packages (Mitropoulos 2019; Nielsen et al. 2021). Contrary to Scotty, its goal is not to discover implementation and type declaration inconsistencies.

Beyond TypeScript and DefinitelyTyped, the Flow (Chaudhuri 2016) type system also has the basic design choices that Scotty supports. As an experiment, we used the Babel Flow parser and adapted our tool to generate contracts for Flow packages. We were able to detect problems with the `http-codes` package (the same problem that we found with the TypeScript version of that package). Based on that experiment, we are hopeful that this tool might also generalize to supporting Flow, but we have not investigated the idea deeply.

6 CONCLUSION

Our work demonstrates that packages in the JavaScript ecosystem are sufficiently similar that one can build tools to detect bugs in their type declarations with gradual typing. This feasibility is surprising given the hostile-to-contract-checking semantics of proxies, conflicting module systems, and dozens of different linting and testing tools with slightly different behaviors. The extent to which the npm ecosystem caused more problems than JavaScript itself is also surprising: we did not expect nor anticipate that, when left unchecked, semicolon placement would cause more problems than the actual behavior of the language.

We are excited by how useful we have found the tool for assisting us in finding and fixing bugs, despite an approach that comes with no guarantees. Our tool is not complete, as the unit tests that come with the package may be insufficient to find errors in the types and we do not handle all of the types in TypeScript. It is also not sound, as generating dynamic checks based on an unsound type system inevitably leads to incorrect checks. In short, some packages will always be inherently incompatible with Scotty. From a more practical perspective, a fully automatic tool will remain out of reach because finding the correct fix is still a fully-manual process. Scotty can only identify mismatches between the types and the tests; it cannot offer fixes.

Nevertheless, we expect that relatively small amounts of further engineering effort into the tool could result in detecting many more mismatches between TypeScript types and the behavior of JavaScript packages. Indeed, we have followed a practical, error-based discipline as we developed Scotty, keeping our focus on the largest obstacle at each stage, instead of the one our programming-language or programmer sensibilities might encourage us to focus on. This discipline has led to a large number of closed pull requests fixing bugs on DefinitelyTyped.

Going beyond mere bug-finding, we are optimistic that further analysis of type mismatches could even suggest future changes to TypeScript's design, especially for the cases in which programmers prefer unsound types.

ACKNOWLEDGMENTS

Thanks to the PL group at Northwestern University for feedback on the work.

This material is based upon work supported by the National Science Foundation under Grant Number CNS-1823244 and the French ANR ANR17-CE25-0014-01 CISC project and the Inria SPAI project.

BIBLIOGRAPHY

Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Proc. Workshop on Script to Program Evolution*, 2009. https://doi.org/10.1145/1570506.1570507

Amal Ahmed, Dustin Jamner, Jeremy Siek, and Philip Wadler. Theorems for Free for Free: Parametricity, With and Without Types. In *Proc. ACM International Conference on Functional Programming*, 2017. https://doi.org/10.1145/3110283

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017. https://doi.org/10.1145/3133878

Moritz Beller, Radjino Bholanath, Shane Mcintosh, and Andy Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proc. IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 470–481, 2016. https://doi.org/10.1109/SANER.2016.105

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. European Conference on Object-Oriented Programming*, 2014. https://doi.org/10.1007/978-3-662-44202-9_11

Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. In *Proc. ACM International Conference on Functional Programming*, 2017. https://doi.org/10.1145/3110285

Avik Chaudhuri. Flow: Abstract Interpretation of JavaScript for Type Checking and Beyond (Invited Talk). In *Proc. Workshop on Programming Languages and Analysis for Security*, 2016. https://doi.org/10.1145/2993600.2996280

Fernando Cristiani and Peter Thiemann. Generation of TypeScript Declaration Files from JavaScript Code. In *Proc. International Conference on Managed Programming Languages and Runtimes*, 2021. https://doi.org/10.1145/3475738.3480941

Ecma International. Standard ECMA-262 - ECMAScript Language Specification. 2015. http://www.ecma-international.org/ecma-262/6.0/

Asger Feldthaus and Anders Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2014. https://doi.org/10.1145/2660193.2660215

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible Contracts: Fixing a Pathology of Gradual Typing. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2018. https://doi.org/10.1145/3276503

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM International Conference on Functional Programming*, 2002. https://doi.org/10.1145/2502508.2502521

Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. JavaScript Errors in the Wild: An Empirical Study. In *Proc. IEEE 22nd International Symposium on Software Reliability Engineering*, 2011. https://doi.org/10.1109/ISSRE.2011.28

Zheng Gao, Christian Bird, and Earl T. Barr. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *Proc. International Conference on Software Engineering*, 2017. https://doi.org/10.1109/ICSE.2017.75

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 2019. https://doi.org/10.1017/S0956796818000217

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid Checking for Flexible Specifications. In *Proc. Workshop on Scheme and Functional Programming*, pp. 93–104, 2006. http://www.schemeworkshop.org/2006/06-freund.pdf

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-Parametric Polymorphic Contracts. In *Proc. Dynamic Languages Symposium*, 2007. https://doi.org/10.1145/1297081.1297089

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proc. International Conference on Static Analysis*, 2009. https://doi.org/10.1007/978-3-642-03237-0_17

Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent Object Proxies in JavaScript. In *Proc. European Conference on Object-Oriented Programming*, 2015. https://doi.org/10.4230/LIPIcs.ECOOP.2015.149

Matthias Keil and Peter Thiemann. On the Proxy Identity Crisis. 2013. https://arxiv.org/abs/1312.5429

Matthias Keil and Peter Thiemann. Blame Assignment for Higher-Order Contracts with Intersection and Union. In *Proc. ACM International Conference on Functional Programming*, 2015. https://doi.org/10.1145/2858949.2784737

Erik Krogh Kristensen and Anders Møller. Inference and Evolution of TypeScript Declaration Files. In *Proc. International Conference on the Fundamental Approaches to Software Engineering*, 2017a. https://doi.org/10.1007/978-3-662-54494-5_6

Erik Krogh Kristensen and Anders Møller. Type Test Scripts for TypeScript Testing. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017b. https://doi.org/10.1145/3133914

Erik Krogh Kristensen and Anders Møller. Reasonably-Most-General Clients for JavaScript Library Analysis. In *Proc. International Conference on Software Engineering*, 2019. https://doi.org/10.1109/ICSE.2019.00026

Rabee Malik, Jibesh Patra, and Michael Pradel. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *Proc. International Conference on Software Engineering*, 2019. https://dl.acm.org/doi/10.1109/ICSE.2019.00045

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, Theorems for low, low prices! In *Proc. European Symposium on Programming*, 2008. https://doi.org/10.1007/978-3-540-78739-6_2

Charalambos Mitropoulos. Employing Different Program Analysis Methods to Study Bug Evolution. In *Proc. International Symposium on the Foundations of Software Engineering*, 2019. https://doi.org/10.1145/3338906.3342489

Cameron Moy, PhC. Nguyundefinedn, Sam Tobin-Hochstadt, and David Van Horn. Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. In *Proc. ACM Symposium on Principles of Programming Languages*, 2021. https://doi.org/10.1145/3434334

Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017. https://doi.org/10.1145/3133880

Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2020. https://doi.org/10.1145/3428255

Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries. In *Proc. International Conference on Software Engineering*, 2021. https://doi.org/10.1109/ICSE43902.2021.00020

Aseem Rastogi, Nikhil Swamy, Cand Bierman, Gavin Fournet, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. ACM Symposium on Principles of Programming Languages*, 2015. https://doi.org/10.1145/2676726.2676971

N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular Shape Analysis for Dynamically Encapsulated Programs. In *Proc. European Symposium on Programming*, 2007. https://dl.acm.org/doi/10.5555/1762174.1762197

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proc. International Symposium on the Foundations of Software Engineering*, 2013. https://doi.org/10.1145/2491411.2491447

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proc. Workshop on Scheme and Functional Programming*, 2006. http://scheme2006.cs.uchicago.edu/13-siek.pdf

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Symposium on Principles of Programming Languages*, 2016. https://doi.org/10.1145/2837614.2837630

The TypeScript Handbook. 2019. https://www.typescriptlang.org/docs/handbook/type-compatibility.html#comparing-two-functions

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2006. https://doi.org/10.1145/1176617.1176755

Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie Van Deursen. Why and How JavaScript Developers Use Linters. In *Proc. ACM/IEEE International Conference on Automated Software Engineering*, pp. 578–589, 2017. https://dl.acm.org/doi/10.5555/3155562.3155634

Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie Van Deursen. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* 46(8), pp. 863–891, 2020. https://doi.org/10.1109/TSE.2018.2871058

Tom Van Cutsem and Mark Miller. Trustworthy Proxies - Virtualizing Objects with Invariants. In *Proc. European Conference on Object-Oriented Programming*, 2013. https://research.google/pubs/pub40736/

Michael M. Vitousek, Andrew M. Kent, Jeremy Siek, and Jin G. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. Dynamic Languages Symposium*, 2014. https://doi.org/10.1145/2661088.2661101

Jack Williams, Garrett J. Morris, and Philip Wadler. The Root Cause of Blame: Contracts for Intersection and Union Types. *ACM Transactions on Programming Languages and Systems*, 2018. https://doi.org/10.1145/3276504

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *Proc. European Conference on Object-Oriented Programming*, 2017. https://www.pure.ed.ac.uk/ws/portalfiles/portal/38670500/ecoop17_mixed_messages_2.pdf