

Generation of TypeScript Declaration Files from JavaScript Code

Fernando Cristiani^a and Peter Thiemann^b

^a Hochschule Karlsruhe, Germany

^b Universität Freiburg, Germany

Abstract Developers are starting to write large and complex applications in TypeScript, a typed dialect of JavaScript. TypeScript applications integrate JavaScript libraries via typed descriptions of their APIs called declaration files. DefinitelyTyped is the standard public repository for these files. The repository is maintained manually by volunteers, which is error prone and time consuming. Discrepancies between a declaration file and the JavaScript implementation lead to incorrect feedback from the TypeScript IDE and thus to incorrect uses of the underlying JavaScript library.

This work presents dts-generate, a tool that generates TypeScript declaration files for JavaScript libraries uploaded to the NPM registry. It extracts code examples from the documentation written by the developer, executes the library driven by the examples, gathers run-time information, and generates a declaration file based on this information. To evaluate the tool, 244 declaration files were generated and 82 were compared with the corresponding declaration file provided on DefinitelyTyped. 52 files presented positive results, having no differences at all or differences that can be solved by modifying the code examples accordingly.

ACM CCS 2012

▪ *Software and its engineering* → *Software notations and tools*; *General programming languages*;

Keywords JavaScript, TypeScript, Dynamic Analysis, Declaration Files

The Art, Science, and Engineering of Programming

Perspective The Empirical Science of Programming

Area of Submission General-purpose programming, Web programming



© Fernando Cristiani and Peter Thiemann
This work is licensed under a “CC BY 4.0” license.
Submitted to *The Art, Science, and Engineering of Programming*.

1 Introduction

JavaScript is the most popular language for writing web applications [5]. It is also increasingly used for back-end applications running in NodeJS, a JavaScript-based server-side platform. JavaScript is appealing to developers because its forgiving dynamic typing enables them to create simple pieces of code very quickly and proceed on a trial-and-error basis.

JavaScript was never intended to be more than a scripting language and thus lacks features for maintaining and evolving large codebases. However, nowadays developers create large and complex applications in JavaScript. Mistakes such as mistyped property names and misunderstood or unexpected type coercions cause developers to spend a significant amount of time in debugging. There is ample evidence for such mishaps. For example, a JavaScript code blog¹ collects experiences from developers facing unexpected situations while programming in JavaScript. Listing 1 exposes some of these unintuitive JavaScript behaviors.

The cognitive load produced by these behaviors is mitigated in languages that use build tools based on type information. This insight motivated the creation of TypeScript, a superset of JavaScript with expressive type annotations [9]. It has become a widely used alternative among JavaScript developers, because it incorporates features that are helpful for developing and maintaining large applications [4]. TypeScript enables the early detection of several kinds of run-time errors and the integration of code intelligence tools like autocompletion in an IDE.

Despite the advantages, it is unrealistic to expect the world to switch to TypeScript in a day. Therefore, existing JavaScript libraries can be used in a TypeScript project by adding a declaration file that contains a description of the library's API in terms of types. The DefinitelyTyped repository [1] has been created as a community effort

¹ <https://wtfs.com>

```
"0" == false; // true
true == "1.00"; // true
false == " \n\r\t "; // true
false == []; // true
0 == []; // true
null == undefined; // true
[1] + 1; // '11'
[2] == "2"; // true
null + undefined + [1, 2, 3] // 'NaN1,2,3'
"hello world".lenth + 1 // NaN | note 'lenth' instead of 'length'
[1, 15, 20, 100].sort() // [ 1, 100, 15, 20 ]
typeof null; // object
null instanceof Object; // false
"01" < "00100"; // false
```

■ **Listing 1** Unintuitive JavaScript behavior Falsy values, **typeof**, **null** and **undefined** operators and type coercion

to collect declaration files for popular JavaScript libraries. At the time of writing it contains declaration files for more than 6000 libraries.

Unfortunately, creation and maintenance for most declaration files in DefinitelyTyped is conducted manually, which is error prone and time consuming. As TypeScript takes a declaration file at face value, discrepancies are not detected at compile time and the resulting inaccuracies in autocompletion, say, are confusing for the programmer. As TypeScript does not perform any run-time checking of types, either, discrepancies between the declaration file and its corresponding JavaScript library can lead to unexpected behavior and crashes. Such an experience can lead to developer frustration, longish debugging sessions, and decreasing confidence in the tool chain.

1.1 Approach

We aim to improve on this situation by providing a tool that generates sound TypeScript declaration files automatically. To do so, we rely on the examples provided as part of the documentation of a library. If these examples are sufficiently rich, then our toolchain can generate high-quality declaration files from them. This approach takes advantage of best practices in the dynamic languages community which favors examples and tests over writing out type signatures.

As an example, consider the NPM module `abs` that “computes the absolute path of an input”. Its documentation consists of the following three examples.²

```
const abs = require("abs");

console.log(abs("/foo"));
// => "/foo"

console.log(abs("foo"));
// => "/path/to/where/you/are/foo"

console.log(abs("~/foo"));
// => "/home/username/foo"
```

From the examples, our tool generates a TypeScript declaration file, which is equivalent to the declaration file provided in DefinitelyTyped³:

```
export = Abs;
declare function Abs(input: string): string;
```

Our tool `dts-generate` is a first step to explore the possibilities for generating useful declaration files from code examples. `dts-generate` comes with a framework that supports the generation of declaration files for an existing JavaScript library published to the NPM registry. The tool gathers data flow and type information at run time to generate a declaration file based on that information.

The novelty of our tool is twofold:

² <https://www.npmjs.com/package/abs>

³ <https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/abs/index.d.ts>

Generation of TypeScript Declaration Files from JavaScript Code

1. We do not rely on static analysis, which is hard to implement soundly and precisely and which is prone to maintenance problems when keeping up with JavaScript's frequent language updates.
2. Instead we extract example code from the programmer's library documentation and rely on dynamic analysis to extract typed usage patterns for the library from the example runs.

1.2 Contributions

- A framework that extracts code examples from the documentation of an NPM package and collects run-time type information from running these examples (Sections 4.1 and 4.2).
- Design and implementation of the tool `dts-generate`, a command line application that generates a valid TypeScript declaration file for a specific NPM package using run-time information (Section 4.3).
- A comparator for TypeScript declaration files (Section 5.2). This tool is necessary for evaluating our framework and also useful to detect incompatibilities when evolving JavaScript modules.
- An evaluation of our framework (Sections 5 and 6). We examined all 6029 entries in the DefinitelyTyped repository and found 244 sufficiently well-documented NPM packages, on which we ran `dts-generate` and compared the outcome with the respective declaration file from the DefinitelyTyped repository.

2 Motivating Example

The NPM package `glob-to-regexp` is a simple JavaScript library, which turns a glob expression for matching filenames in the shell into a regular expression.⁴ It has about 9,900,000 weekly downloads and 188 NPM packages depend on it. If a developer creates or extends TypeScript code that depends on the `glob-to-regexp` library, the TypeScript compiler and IDE requires a declaration file for that library to perform static checking and code completion, respectively. With `dts-generate` we automatically generate a TypeScript declaration file for `glob-to-regexp`. The tool downloads the NPM package, runs the examples extracted from its documentation, gathers run-time information, and generates a TypeScript declaration file. The result is shown in Figure 1 and it is ready for use in a TypeScript project. For example, Visual Studio's code completion runs properly with the generated file (Figure 1). If the `glob-to-regexp` package gets modified in the future, a new declaration file can be generated automatically using `dts-generate`. Our comparator tool can compare the new file for incompatibilities with the previous declaration file.

⁴ <https://www.npmjs.com/package/glob-to-regexp>

```

export = GlobToRegexp;

declare function GlobToRegexp(glob: string, opts?: GlobToRegexp.I__opts): RegExp;
declare namespace GlobToRegexp {
  export interface I__opts {
    'extended'?: boolean;
    'globstar'?: boolean;
    'flags'?: string;
  }
}

```



Figure 1 Declaration file for **glob-to-regexp** generated with **dts-generate** - The interface is detected correctly. Optional parameters are detected. The declaration file is usable in Visual Studio Code.⁵

```

$ ./dts-generate abs
$ cat output/abs/index.d.ts
export = Abs;

declare function Abs(input: string): string;

```

Listing 2 Example for template **module-function**

3 TypeScript Declaration Files

The declaration file shown in Figure 1 describes a package with a single exported function. The first parameter is of type `string` and the second one is an optional interface. TypeScript namespaces organize types declared in a declaration file and avoid name clashes with other types [11]. For example, the interface `I__opts` declared in Figure 1 is used as `globToRegexp.I__opts` because it is declared within a namespace.

This file is an instance of one of the standard templates for writing declaration files⁶: **module**, **module-class**, and **module-function**. Each template corresponds to a different way of describing the exports of a JavaScript library. The choice of the template depends on the organization of the underlying JavaScript library:

- module** several exported functions,
- module-class** a class-like structure,
- module-function** exactly one exported function.

⁶ <https://www.typescriptlang.org/docs/handbook/declaration-files/templates.html>

Generation of TypeScript Declaration Files from JavaScript Code

```
var greet = require("./greet-settings-module");

greet({
  greeting: "hello world",
  duration: 4000
});

greet({
  greeting: "hello world",
  color: "#ooffoo"
});
```

(a) Example for module-function template

```
var myLib = require("./greet-module");

var result = myLib.makeGreeting("hello, world");
console.log("The computed greeting is: " + result);

var goodbye = myLib.makeGoodBye();
console.log("The computed goodbye is: " +
  goodbye);
```

(c) Example for module template

```
var Greeter = require("./greet-classes-module.js");

var myGreeter = new Greeter("hello, world");
myGreeter.greeting = "howdy";
myGreeter.showGreeting();
```

(e) Example for module-class template

```
export = GreetSettingsModule;

declare function GreetSettingsModule(settings:
  ↳ GreetSettingsModule.I__settings): void;
declare namespace GreetSettingsModule {
  export interface I__settings {
    'greeting': string;
    'duration'?: number;
    'color'?: string;
  }
}
```

(b) Declaration file for module-function template

```
export function makeGreeting(str: string): string;
export function makeGoodBye(): string;
```

(d) Declaration file for module template

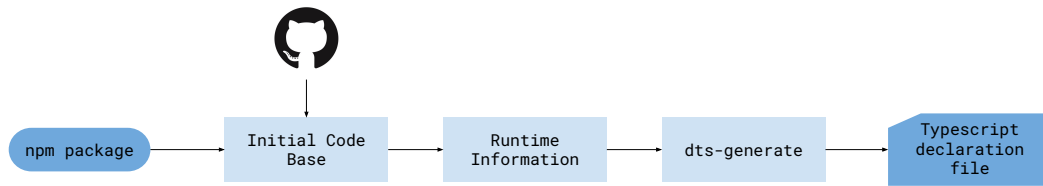
```
export = Greeter;

declare class Greeter {
  constructor(message: string);
  showGreeting(): void;
}

declare namespace Greeter {
}
```

(f) Declaration file for module-class template

■ **Figure 2** Example uses and declaration files for different templates



■ **Figure 3** **dts-generate - Architecture overview** - The initial code base is retrieved from the NPM package’s repository. Run-time information is gathered driven by example code. A TypeScript Declaration File is generated using run-time information.

The TypeScript project provides a guide on how to write high-quality declaration files.⁷ The guide explains the main concepts through examples. We selected one example for each template from the guide and used `dts-generate` to generate a corresponding declaration file, as shown in Figure 2.

Like `dts-gen`, a TypeScript package that generates a template for a declaration file, we determine the kind of declaration file by examining the provided example. If the imported entity is used solely as a function as in Figure 2a, the generated declaration file follows the `module-function` template as shown in Figure 2b. If the example only accesses properties of the imported entity as in Figure 2c, we generate according to the `module` template (Figure 2d). If the imported entity is used with `new` to create new instances as in Figure 2e, we generate a `module-class` template (Figure 2f).

The libraries `glob-to-regexp` as well as `abs` are both instances of the **module-function** template. Most libraries fall in one of these three categories. While it is possible to create a library of undetermined category, our selection is driven by the examples in the documentation and hence reflects the developers intent.

4 The Generation of TypeScript Declaration Files

This section gives an overview of our approach to generating TypeScript declaration files from a JavaScript library packaged in NPM. Figure 3 gives a rough picture of the inner working of our tool `dts-generate`. The input is an NPM package and the output is a TypeScript declaration file for the package if it is “sufficiently documented”, which we substantiate in the next subsection.

As `dts-generate` is based on run-time information, exemplary code fragments that execute the JavaScript library are needed to obtain significant run-time information from running the instrumented library code.

The examples and the code base of the library are instrumented with Jalangi [8] to gather data flow information and type information at run time. Jalangi is a configurable framework for dynamic analysis of JavaScript. It provides several analysis modules that we extended as needed to retrieve the required run-time information, which is then saved in a JSON file.

⁷ <https://www.typescriptlang.org/docs/handbook/declaration-files/by-example.html>

Generation of TypeScript Declaration Files from JavaScript Code

A second independent block uses the run-time information to generate a TypeScript declaration file. It infers the overall structure of the JavaScript library, its interfaces, and the types from the run-time information. The resulting declaration file is ready for use in the development process. Its contents mimic the usage of the library in the example code fragments and matches the structure of the JavaScript library under analysis, so that the JavaScript code generated after compiling the TypeScript code runs without interface-related errors.

The command-line interface is inspired by the `dts-gen` tool [2]. Listing 2 shows that invoking the package is very simple. Its only required argument is the name of the module published to the NPM registry.

4.1 Initial Code Base

To extract run-time information from a JavaScript library, it is necessary, by definition, to actually execute the code, because the analysis modules provided by Jalangi to gather information are only triggered if the instrumented code gets executed.

There are several options to obtain code fragments that drive the library code:

1. execute code that imports the library;
2. execute the test cases that come with the library;
3. execute code fragments extracted from the library documentation.

Option 1 does not solve our problem, it just delegates it to the importing library, which also needs code to drive it. Moreover, it is costly to download and instrument another package.

We considered option 2 under the assumption that most libraries would come with test cases. However, there is no standard for testing JavaScript code so that test cases were difficult to reap from the NPM packages: they use different directory structures, employ differing (or no) testing tools, or do not have tests at all. Moreover, developers try to cover edge cases or to trigger errors by using illegal input values in their tests. Such tests are not really useful for describing a library from the consumer's point of view. For example, a developer might write a test invoking a function with null just to validate that it throws an error in such scenario.

In the end, option 3 was the most viable even though there is no standard for documentation, either. However, almost all repositories contain README files where the library authors briefly describe in prose what the code does, which problem it solves, how to install the application, how to build the code, etc. It is very common that developers provide code examples in the README files to show how the library works and how to use it. These code fragments represent common usage cases rather than stress-testing the implementation (the problem of option 2) because they are meant to be instructive to users of the package.

Obtaining code examples for a specific NPM package is done in three steps.

- Obtain the repository's URL with the command `npm view <PACKAGE> repository.url`
- Retrieve the README file from the top-level directory of the repository.


```

var globToRegExp = require('glob-to-regexp');
var re = globToRegExp("p*uck");
re.test("pot luck"); // true
re.test("pluck"); // true
re.test("puck"); // true

re = globToRegExp("*.min.js");
re.test("http://example.com/jquery.min.js"); // true
re.test("http://example.com/jquery.min.js.map"); // false

re = globToRegExp("*/www/*.js");
re.test("http://example.com/www/app.js"); // true
re.test("http://example.com/www/lib/factory-proxy-model-observer.js"); // true

// Extended globs
re = globToRegExp("*/www/{*.js,*.html}", { extended: true });
re.test("http://example.com/www/app.js"); // true
re.test("http://example.com/www/index.html"); // true

```

■ **Listing 3** Extracted example code for the glob-to-regexp package

- Extract the code examples from the README file. To this end, observe that README files are written using Markdown,⁸ a popular markup language, where code examples are presented in code blocks labeled with the programming language, so that syntax highlighting is done correctly. Hence, we retrieve the code examples from code blocks labeled js or javascript, which both stand for JavaScript in Markdown.

Listing 3 shows the examples extracted from the README file of the glob-to-regexp package in that way.

Obtaining the code fragments from the examples provided in the README files of the repository proved to be an appropriate and pragmatic way of extracting the developer's intention and provides a useful initial code base with meaningful examples, thus avoiding possible cold start problems.

4.2 Run-time Information Gathering

The Runtime Information block described in Figure 3 gathers type-related information as well as usage-related information for each function parameter. For example,

- Function *f* was invoked where parameter *a* held a value of type *string* and *b* a value of type *number*.
- Property *foo* of parameter *a* of function *f* was accessed within the function.
- Parameter *a* of function *f* was used as operand for operator *==*.

The dynamic analysis framework Jalangi is used for gathering this kind of information. The configurable analysis modules enable programming custom callbacks that can get triggered with virtually any JavaScript event. Our instrumentation observes the following events:

⁸ <https://www.markdownguide.org>

Generation of TypeScript Declaration Files from JavaScript Code

- binary operations, like `==`, `+`, or `===`;
- variable declarations;
- function, method, or constructor invocations;
- access to an object's property;
- unary operations, like `!` or `typeof`.

The implementation stores these observations as entities called interactions. They are used for translating, modifying, and aggregating Jalangi's raw event information to get an application-specific data representation. Each function invocation is stored as a `FunctionContainer` which contains an `ArgumentContainer` for each argument. Each `ArgumentContainer` contains the name and index of the argument and a collection of interactions. The interaction `getField` gets triggered whenever a property of an object gets accessed and it tracks the name of the accessed field. The invocation of an object's method is tracked with the `methodCall` interaction. The `usedAsArgument` interaction gets triggered when an argument is used in the invocation of a function. The `followingInteractions` property is used for inferring nested interfaces. It is an array that recursively records interactions on the return value of `getField` or `methodCall`.

We wrap each function's argument in a wrapper object, which enables to store meta-information and greatly simplifies the mapping of an observation with the corresponding `ArgumentContainer` or interaction. We use the original values for critical operators such as `===` or `typeof`, for which wrapper objects would modify the behavior of the code.

The property `requiredModule` stores the name of the module that declared the invoked function. If a function is explicitly exported by the module, the property `isExported` will be set to `true` when it is invoked. If a function of an exported object is invoked, `isExported` will be `false` and `requiredModule` will contain the name of the required module.

The run-time information is saved as a JSON file that can be used for later processing. The instrumentation tool is written in JavaScript and runs in NodeJS in a Docker container.

4.3 TypeScript Declaration File Generation

The next step in the pipeline after gathering the run-time information is the generation of the declaration file (cf. Figure 3). It is a lightweight, simple, and fast application, which solely relies on the run-time information gathered in the previous step.

Instead of analyzing the shape of the exported module, we inspect how the module is used to choose between the templates. We analyze the properties `isExported`, `requiredModule` and `isConstructor` from the run-time JSON to distinguish between module-class and module-function. If a function is invoked and it is imported from the module we are analyzing we infer the template module-class or module-function. We choose module-class if the function is used as a constructor. Otherwise, we choose module-function. If no function is invoked, we use the module template.

We implemented only basic TypeScript features and we focused our effort in building an end-to-end solution that generates valid and useful declaration files. Only the

```
export function v1(str: string): boolean;
export function v2(str: string): boolean;
export function v3(str: string): boolean;
export function v4(str: string): boolean;
export function v5(str: string): boolean;
```

(a) is-uuid/index.d.ts - Generated

```
export function v1(value: string): boolean;
export function v2(value: string): boolean;
export function v3(value: string): boolean;
export function v4(value: string): boolean;
export function v5(value: string): boolean;
export function nil(value: string): boolean;
export function anyNonNil(value: string): boolean;
```

(b) is-uuid/index.d.ts - DefinitelyTyped

■ **Figure 4** Results for `module` module is-uuid

basic types `string`, `number`, `boolean`, its union and the optional type were considered both for function parameters and interface properties. Common types such as `any`, arrays, function callbacks, tuples, intersections and features such as generics, index signatures were not implemented.

Each declaration file in the DT repository was tagged with `dts-parse`. We summarized the results in Table 1. We based our decisions of prioritizing particular features on these results.

Finally, interfaces are created by exploring `getField` and `methodCall` interactions from the run-time information. We gather the interactions for a specific argument and build the interface by incrementally adding new properties. Interactions within the `followingInteractions` field are recursively traversed, building a new interface at each level up to a predefined depth. Interfaces that turn out to be equivalent are merged.

5 Evaluation

After generating a declaration file for an NPM package, we need to evaluate its quality. To this end, we created `dts-compare`, a tool to compute the differences between two TypeScript declaration files.

For each module, we applied `dts-compare` to the generated declaration file and the one uploaded to the DefinitelyTyped repository for the same module, as shown in Figure 5. While this approach does not provide an absolute measure of quality, it gives us at least an indication of the pragmatics of `dts-generate`: The files on DefinitelyTyped are perceived to be useful by the community. If the accuracy of the generated files is comparable with the accuracy of the files on DefinitelyTyped, then `dts-generate` is a viable alternative to manually created definition files.

Question: check how many files on DT are just generated by `dts-gen` (e.g., the TypeScript tool)

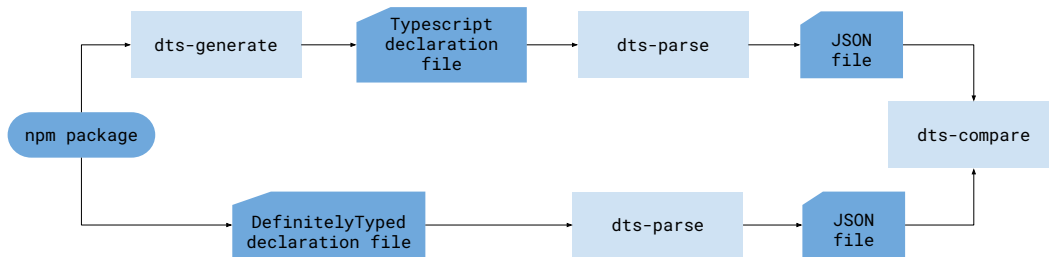
5.1 dts-parse

We need a means of comparing two declaration files. Of course, we are not interested in a textual comparison, but in a comparison of the structures described by the files.

Generation of TypeScript Declaration Files from JavaScript Code

TypeScript feature	Count Declaration Files	%
string	5086	76.50%
optional-parameter	4915	73.93%
boolean	3891	58.53%
number	3699	55.64%
void	3548	53.37%
type-union	3456	51.99%
type-function	3286	49.43%
type-array	3264	49.10%
type-any	3127	47.04%
literals	1925	28.96%
alias-type	1899	28.56%
index-signature	1271	19.12%
generics-function	1145	17.22%
dot-dot-dot-token	882	13.27%
call-signature	817	12.29%
generics-interface	718	10.80%
object	661	9.94%
undefined	577	8.68%
type-intersection	431	6.48%
readonly	373	5.61%
type-tuple	355	5.34%
generics-class	265	3.99%
static	251	3.78%
private	82	1.23%
public	43	0.65%
protected	19	0.29%

■ **Table 1** Distribution of TypeScript features in declaration files uploaded to Definitely-Typed. The 2nd column shows how many files in DT make use of a specific TypeScript feature. The 3rd column shows the corresponding percentage for a total of 6648 declaration files in DT. Numbers were obtained from commit da93d7b13094bacb170ead3f4a289f3b8687e4e5 (April 4, 2020).



■ **Figure 5** Evaluation of generated declaration files against DefinitelyTyped Repository - The parser uses the TypeScript Compiler API [10] to transform declaration files into an abstract syntax which is serialized in JSON. The comparison is performed on the JSON format.

```
$ dts-parse -i glob-to-regexp/index.d.ts | jq '.parsing.functions[0].name'
"GlobToRegexp"
```

■ **Listing 4** Example for dts-parse executed on module glob-to-regexp and combined with jq to browse the returned JSON

```
$ dts-compare -e expected.d.ts -a actual.d.ts --module-name "my-module"
{
  "module": "my-module",
  "template": "module",
  "differences": [],
  "tags": []
}
```

■ **Listing 5** Example for dts-compare executed on declaration files without differences

The first step is to parse the declaration files, which can be achieved by using the TypeScript Compiler API, a library which traverse the Abstract Syntax Tree of a TypeScript program [10]. The step also performs a sanity check of the generated declaration files as it rejects files with syntactic or semantics errors. The output of dts-parse is a structure where declared interfaces, functions, classes and namespaces are stored separately. Function arguments are correctly described, identifying complex types like union types or callbacks. Optional parameters are also identified. For classes, a distinction between the constructor and methods is made. Declaration files are tagged according to their features, which is useful for identifying and filtering out declaration files containing unimplemented features.

dts-parse is itself written in TypeScript. It receives the file as input parameter and returns an AST serialized in JSON, as shown in Listing 4.

5.2 dts-compare

The tool is written in TypeScript. It takes two declaration files as input and returns a JSON file containing the result of the comparison, as shown in Listing 5. Following the naming convention of any standard Unit Testing framework, the input files are marked as expected or actual. We use expected for the DefinitelyTyped declaration files.

dts-compare applies dts-parse internally to both files to get normalized structures that are easy to compare. The result of the comparison is an array of an abstract entity Difference.

We extracted code examples written by developers to execute the code and gather run-time information. The quality of the generated declaration file is heavily dependent on the code examples. For example, if a property of an interface does not get accessed due to a code example not exploring a particular execution path, that property will not exist in the generated declaration file.

To measure the impact of incomplete code examples on the quality of generated declaration files we categorized each difference as follows:

Generation of TypeScript Declaration Files from JavaScript Code

- solvable: Differences that can be resolved by adding more code examples. For example, invoking a function with different parameters so that a missing interface property gets accessed.
- unsolvable: Differences that cannot be resolved with further examples.

The concrete evaluated differences are as follows:

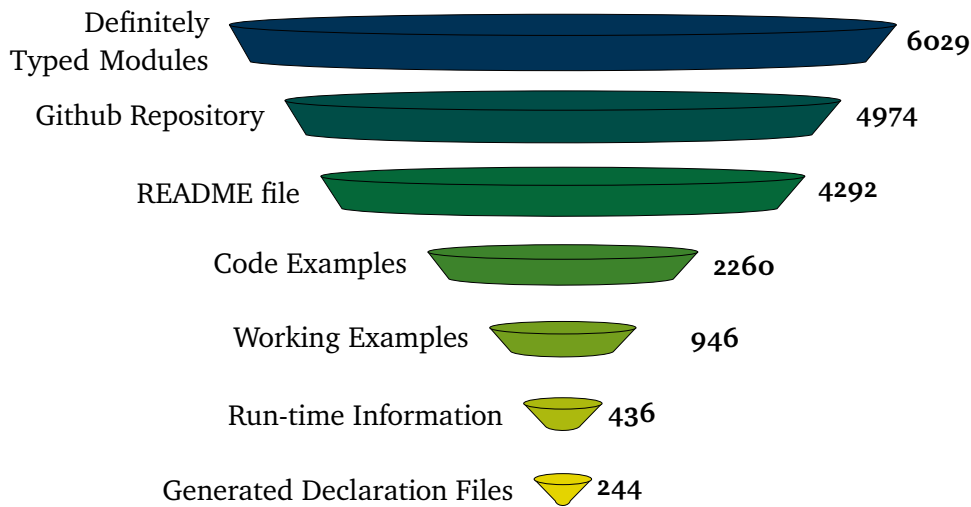
- TemplateDifference: A difference between the choice of TypeScript templates. For example, if the file in DefinitelyTyped is written as a module but we generated the file using the module-function template. The comparison stops if the templates differ.
- ExportAssignmentDifference: An equality check between the export assignments of both files, that is in the expression `export = XXX`.
- FunctionMissingDifference: A function declaration is present in the expected file but not in the generated one. This difference applies to functions as well as methods of classes and interfaces.
- FunctionExtraDifference: A function is present in the generated declaration file but not in DefinitelyTyped.
- FunctionOverloadingDifference: The number of declarations for the same function is different. This difference can be due to inexperience of the author of a declaration file, who uses, e.g., multiple declarations where a union typed argument would be appropriate.
- ParameterMissingDifference: A parameter of a function or a property of an interface is not present in the generated file.
- ParameterExtraDifference: A parameter of a function or a property of an interface is present in the generated file but not in the DefinitelyTyped file.
- ParameterTypeDifference: A parameter of a function or a property of an interface is generated with a different type than in the DefinitelyTyped file. Here we differentiate between SolvableDifference and UnsolvableDifference.
 - SolvableDifference: A type difference that can be solved by writing further code examples. For example, a basic type that is converted to a union type, a function overloading, or a parameter or property that is marked as optional.
 - UnsolvableDifference: Any difference not considered as SolvableDifference.

Type aliases are expanded before the comparison and thus invisible to dts-compare: the declaration `type T = string | number; declare function F(a: T);` is equivalent to `declare function F(a: string | number);`. The same approach applies to literal interfaces. dts-compare does not contemplate differences in function return types, because interface inference of function return types is not covered in the current version of dts-generate.

6 Results

We analyzed the obtained results focusing on the following aspects:

- The quality of the inferred types, interfaces, and module structure using run-time information.



■ **Figure 6** Number of analyzed modules for each stage of the experiment - A TypeScript Declaration File was generated for only 244 modules, out of 6029 modules in the DefinitelyTyped Repository. It was possible to gather valid run-time information for only 25% of the modules for which a Code Example was extracted.

- The benefits of using code examples provided by developers as a first approximation to execute the libraries and avoid a cold start problem.
- The usability of the generated declaration file and whether dts-generate can be used in a proper development environment.

The conducted experiments included tests that consisted of replacing a specific type definition from DefinitelyTyped [1] with the one generated in the experiments: TypeScript compilation was successful, the generated JavaScript code ran without errors and code intelligence features performed by IDEs like code completion worked as expected.

Declaration files were generated for existing modules uploaded to the NPM registry. The DefinitelyTyped repository was used as a benchmark. Each of the generated files was compared against the corresponding declaration file uploaded to the repository.

Figure 6 shows that a declaration file was generated for 244 modules out of 6029 modules and we identified 60 modules that have only the features implemented by dts-generate. We obtained positive results for 52 of them. Section 6.3 provides a detailed explanation of the overall quality of the generated files. Section 6.2 presents examples of the generated declaration files for templates **module**, **module-class**, and **module-function**.

6.1 Code Examples

Retrieving the code examples for the JavaScript libraries proved to be a pragmatic way of driving the type gathering at run time. However, as shown in Figure 6, it was only possible to obtain working code examples for 2260 packages. The process of getting a valid code example for a module is divided in four stages:

Generation of TypeScript Declaration Files from JavaScript Code

- extracting the repository URL;
- extracting a README file;
- extracting code examples from README files;
- executing code examples and discarding failing ones.

The results obtained for each step are described in the following sections.

Repository URL The URL of the source repository could be retrieved for only 4974 packages. More than 1000 packages on NPM do not have a repository entry in their corresponding package.json file. Therefore, the `npm view <module> repository.url` command returns no value. Even important modules like `ace` provide no repository URL.

README Files 700 packages do not have a README file in their repository, although the implementation checks for several naming conventions like `readme.md` or `README.md`.

Extraction of Code Examples In this step, we loose another 50% of modules! This loss is mainly explained because some developers do not wrap their code in a block using the `javascript` or `js` tags. However, as we are still left with code examples for 2200 modules, we did not further look into code extraction as this number was considered sufficient for evaluating the generation of declaration files.

Execution of Code Examples We executed the remaining 2260 extracted code examples by installing the required packages and running the code as a NodeJS application. Unfortunately, the code examples only worked for 946 modules. 1314 modules did not run correctly and had to be discarded. Some failing samples were analyzed and there were mainly two reasons for the failure:

- The code fragment had been properly extracted but the code was faulty. It invoked the library in an unsupported (obsolete?) way, which lead to a run-time error.
- The extracted code fragment was not intended to be executed and/or it was not even valid JavaScript code.

Run-time Information Run-time information was extracted for only 436 out of 946 modules with working code examples. As explained, to extract the run-time information, the behavior of the code under analysis was explicitly modified by wrapping the arguments. Furthermore, Jalangi's instrumentation itself caused some executions to fail, because the modules contained JavaScript features that are not supported by Jalangi. As a result, run-time information could not be extracted for 510 modules. An instrumentation without user-defined analysis modules was not applied, so it was not possible to determine which modules were failing only because of Jalangi's own limitations.

Generated Declaration Files A declaration file was generated for 244 out of 436 modules. Despite the correct execution of the instrumented code examples, 192 modules did not yield suitable run-time information for generating a declaration file. The

<pre>export = Abs; declare function Abs(input: string): string;</pre> <p>(a) abs/index.d.ts - Generated</p>	<pre>declare function Abs(input: string): string; export = Abs;</pre> <p>(b) abs/index.d.ts - DefinitelyTyped</p>
<pre>export = DirnameRegex; declare function DirnameRegex(): RegExp;</pre> <p>(c) dirname-regex/index.d.ts - Generated</p>	<pre>export = dirnameRegex; declare function dirnameRegex(): RegExp;</pre> <p>(d) dirname-regex/index.d.ts - DefinitelyTyped</p>
<pre>export = EscapeHtml; declare function EscapeHtml(string: string): string;</pre> <p>(e) escape-html/index.d.ts - Generated</p>	<pre>declare function escapeHTML(text: string): string; declare namespace escapeHTML { } export = escapeHTML;</pre> <p>(f) escape-html/index.d.ts - DefinitelyTyped</p>

■ **Figure 7** Results for **module-function** template

extracted code examples for these modules did not execute the library. Hence, the collected run-time information was not useful for generating a declaration file.

6.2 Declaration Files Generation

This section exhibits some examples of the 244 generated declaration files. It shows some results for each of the implemented templates: **module**, **module-function**, and **module-class**.

Figure 7 shows the generated declaration files for simple modules like abs, dirname-regex, and escape-html. All of them were generated using the **module-function** template. The left side of the figure shows the generated declaration file with dts-generate, the right side shows the corresponding file in the DefinitelyTyped repository. There are no relevant differences between the files. Functions are correctly detected; input and output types are accurately inferred.

The differences in the export assignment by modules dirname-regex and escape-html are not problematic because the module can receive an arbitrary name when it is imported. dts-generate automatically generates the export assignment by transforming the module name into camel case form, following TypeScript guidelines.

The parameter names are extracted from the variable names of the JavaScript code. A difference in the names of the function parameters does not affect the correctness of the declaration file. Furthermore, the module escape-html from DefinitelyTyped declares an empty namespace, which is not generated by our tool.

An example for the **module** template is shown for the is-uuid module in Figure 8. The generated file does not contain methods that were not executed by the extracted examples. The invoked function are correctly detected.

Generation of TypeScript Declaration Files from JavaScript Code

```
export function v1(str: string): boolean;  
export function v2(str: string): boolean;  
export function v3(str: string): boolean;  
export function v4(str: string): boolean;  
export function v5(str: string): boolean;
```

(a) is-uuid/index.d.ts - Generated

```
export function v1(value: string): boolean;  
export function v2(value: string): boolean;  
export function v3(value: string): boolean;  
export function v4(value: string): boolean;  
export function v5(value: string): boolean;  
export function nil(value: string): boolean;  
export function anyNonNil(value: string): boolean;
```

(b) is-uuid/index.d.ts - DefinitelyTyped

■ **Figure 8** Results for module is-uuid

```
var smartTruncate = require('smart-truncate');  
  
var string = 'To iterate is human, to recurse divine.';  
  
// Append an ellipsis at the end of the truncated string.  
var truncated = smartTruncate(string, 15);
```

■ **Listing 6** Code example for smart-truncate module exporting a function

Finally, we were unable to generate a declaration file based on the module-class template. All modules corresponding to this template exhibited at least one of the unimplemented TypeScript features.

It is worth mentioning that for some libraries the declaration file in DefinitelyTyped was not correct. For example, for the module glob-base, the parameter basePath is declared as optional in DefinitelyTyped. However, when invoking the function without the basePath parameter, an error is thrown at run time. The file generated by dts-generate does not mark this parameter as optional, as shown in Figure 9. Function return type interfaces are not inferred by dts-generate. The parameter name is pattern instead of basePath, because the JavaScript implementation declares the parameter as pattern, as shown in Figure 9d. We also discovered errors in the selected template in DefinitelyTyped. The module smart-truncate in DefinitelyTyped uses the module template, but dts-generate generates a file using the module-function template. Listing 6 shows that the module is indeed exported as a function, as inferred by dts-generate.

6.3 Evaluation

We analyzed 60 of the 244 generated files. The remaining 184 files contained at least one of the unimplemented TypeScript features so that they could not be considered. These features were identified by dts-parse as tags.

As shown in Figure 8, the code examples have a direct influence on the quality of the generated declaration file. We introduced the concept of solvable and unsolvable differences in Section 5.2 - dts-compare. Modules containing only solvable differences were considered as positive results. We identified 52 such modules. We manually completed the examples for 10 of those modules. Applying dts-generate with the

```
export = GlobBase;

declare function GlobBase(pattern: string): object;
```

(a) glob-base/index.d.ts - Generated

```
var globBase = require('glob-base');

globBase();

// Throws: TypeError: glob-base expects a string.
```

(c) Code example invoking the exported function without the first parameter, as specified in DefinitelyTyped

```
declare function globbase(basePath?: string):
    ↳ globbase.GlobBaseResult;

declare namespace globbase {
    interface GlobBaseResult {
        base: string;
        isGlob: boolean;
        glob: string;
    }
}

export = globbase;
```

(b) glob-base/index.d.ts - DefinitelyTyped

```
'use strict';

// ...

module.exports = function globBase(pattern) {
    if (typeof pattern !== 'string') {
        throw new TypeError('glob-base expects a
            ↳ string.');
```

(d) glob-base/index.js - Fraction of JS implementation that throws an Error when input parameter is not a string

■ **Figure 9** Incorrect optional parameter for module glob-base

completed examples generates declaration files, which are equivalent to the files from DefinitelyTyped, as shown in Figure 10.

We discovered only 8 modules with unsolvable differences. Most of them were caused by the same error: 7 modules incorrectly declared an interface for the string type. This interface contains standard properties and methods from the JavaScript String object, such as length or indexOf().

7 Discussion

How can we improve further? Given that JavaScript developers are more incentivized to develop good examples than writing “boring” type declaration, how can we capitalize on that attitude? The main issue is that any example-driven algorithm can only come up with an under-approximation of the intended meaning, so some form of help is required when generalization is desired.

Generation of TypeScript Declaration Files from JavaScript Code

```
declare namespace gh {  
  interface Options {  
    enterprise?: boolean;  
  }  
  interface Result {  
    user: string;  
    repo: string;  
    branch: string;  
    https_url: string;  
    tarball_url: string;  
    clone_url: string;  
    travis_url: string;  
    api_url: string;  
    zip_url: string;  
  }  
}  
  
declare function gh(url: string | {url: string}, options?: gh.Options): gh.Result | null;  
  
export = gh;
```

(a) github-url-to-object/index.d.ts - DefinitelyTyped

```
export = GithubUrlToObject;  
  
declare function GithubUrlToObject(repoUrl: string | GithubUrlToObject.I__repoUrl,  
  ↳ opts?: GithubUrlToObject.I__opts): object | null;  
declare namespace GithubUrlToObject {  
  export interface I__repoUrl {  
    'url?': string;  
  }  
  
  export interface I__opts {  
    'enterprise?': boolean;  
  }  
}
```

(b) github-url-to-object/index.d.ts - Generated with manually expanded code example

```
var gh = require('github-url-to-object')  
  
gh('github:monkey/business');  
gh('https://github.com/monkey/business');  
gh('https://github.com/monkey/business/tree/master');  
gh('https://github.com/monkey/business/tree/master/nested/file.js');  
gh('https://github.com/monkey/business.git');  
gh('http://github.com/monkey/business');  
gh('git://github.com/monkey/business.git');  
  
// Manually added:  
gh('git+https://github.com/monkey/business.git', {});  
gh('git+https://github.com/monkey/business.git', {enterprise: true});  
gh({url: 'git://github.com/monkey/business.git'});  
gh('this is not a proper url');  
gh({url: 'this is not a proper url'});
```

(c) Code example for module github-url-to-object.

- **Figure 10** Manually expanded code example for github-url-to-object to match DefinitelyTyped declaration file. Literal interface in DefinitelyTyped gets replaced by a declared interface. Return types are not considered.

Literal types vs string JavaScript libraries often (?) use literal strings to select options and configurations. For example, ...

For our framework, it is hard to distinguish a string that is meant literally from a string that just serves as an example. For an automatic solution, one might collect typical values of literal strings and generate the corresponding literal types (or unions thereof) when the arguments used in the example are all typical. Alternatively, one might always generate literal types unless the number of different examples provided is greater than some threshold, in which case the generator generalizes to type `string`. A further alternative would be to communicate to the developer a set of typical example strings, *marker strings*, that the generator always resolves to the `string` type.

Any Suppose the programmer supplies examples that use `number`, `bool`, and `string` at the same argument position. The obvious approximation to the type of this argument is the union type `number|bool|string`. But what if the programmer wants to advertise the type `any` for an argument? This intent is hard to advertise with a finite number of examples.

We propose that the programmer relies on marker strings like `"any"` to indicate the `any` type for an argument position.

Further types Similar approaches could be conceived for indexed types, polymorphic types, dependent types, etc. However, at present we believe incorporating these features in a tool to be overblown because they are not widely used in DefinitelyTyped (see Table 1).

8 Related Work

Microsoft's dts-gen Microsoft developed `dts-gen`, a tool that creates declaration files for JavaScript libraries [2]. Its documentation states that the result is intended to be used as a starting point for development of a declaration file. The outcome needs to be refined afterwards by the developer.

The tool analyzes the shape of the objects at runtime after initialization without executing the library. This results in many variables being inferred as `any`. Listing 7 shows an example for module `abs`.

In contrast, our tool `dts-generate` is intended to generate declaration files that are ready to be uploaded to DefinitelyTyped without further manual intervention. Any amount of manual work that a developer needs to do on a declaration file after updating JavaScript code increases the risk for having discrepancies between the declaration file and the implementation.

TSInfer & TSEvolve TSInfer and TSEvolve are presented as part of TSTools [6]. Both tools are the continuation of TSCheck [3], a tool for detecting mismatches between a declaration file and the implementation of the module.

Generation of TypeScript Declaration Files from JavaScript Code

```
$ npm i -g dts-gen
$ npm i -g abs
$ dts-gen -m abs
Wrote 5 lines to abs.d.ts.

$ cat abs.d.ts
/** Declaration file generated by dts-gen */

export = abs;

declare function abs(input: any): any;
```

■ **Listing 7** Microsoft’s dts-gen example - A declaration file for module abs is generated. Types are inferred as any. The correct module-**function** template is used.

TSTinfer proceeds in a similar way than TSCheck. It initializes the library in a browser and records a snapshot of the resulting state. Then it performs a lightweight static analysis on all the functions and objects stored in the snapshot.

The abstraction and the constraints introduced as part of the static analysis tools for inferring the types have room for improvement. A run-time based approach like the one presented in our work will provide more accurate information, thus generating more precise declaration files.

As TSTinfer analyzes the objects and functions stored in the snapshot, it faces the problem of including internal methods and private properties in the declaration file. Run-time information would have informed that the developer has no intention of exposing these methods.

Moreover, TSEvolve performs a differential analysis on the changes made to a JavaScript library to determine intentional discrepancies between the declaration files of two consecutive versions. However, a differential analysis may not be needed. If the developer’s intention were accurately represented by the extracted code, then the generated declaration file would already describe the newer version of a library without the need of a differential analysis.

TSTest TSTest is a tool that checks for mismatches between a declaration file and the JavaScript implementation of the module [7]. It applies feedback-directed random testing for generating type test scripts. These scripts execute the library with random arguments with the typings from the declarations file and check whether the output matches the prescribed type. TSTest thus provides concrete counterexamples if it detects mismatches.

TSTest could be used to extend our tool with a feedback loop. If TSTest detects a problem with a declaration file generated by dts-generate, then we would add the resulting counterexamples to the example code and restart the generation process.

9 Conclusions and Future Work

We have presented `dts-generate`, a tool for generating a TypeScript declaration file for a specific JavaScript library. The tool downloads code examples written by the developers from the library’s repository. It uses these examples to execute the library and gather data flow and type information. The tool finally generates a TypeScript declaration file based on the information gathered at run time.

We developed an architecture that supports the automatic generation of declaration files for specific JavaScript libraries without additional manual tasks. The architecture contemplates future incorporation of a Symbolic Execution Engine that refines the initial code base enabling the exploration of new execution paths. Although it is not implemented in this work, its incorporation would result in small incremental modifications to the presented architecture as it is considered to only expand the existing code base.

Building an end-to-end solution for the generation of TypeScript declaration files was prioritized over type inference accuracy. Consequently, types were taken over from the values at run time. Developers express their intent how a library should be used with example code in the documentation. Hence, obtaining the types from the code examples extracted from the repositories proved to be a pragmatic and effective approximation, enabling to work on specific aspects regarding the TypeScript declaration file generation itself.

We built a mechanism to automatically create declaration files for potentially every module uploaded to DefinitelyTyped. We managed to generate declaration files for 244 modules. We compared the results against the corresponding files uploaded to DefinitelyTyped by creating `dts-parse`, a TypeScript declaration files parser and `dts-compare`, a comparator.

We exposed the fundamental aspect of capturing the developer’s intention when inferring types in JavaScript. Instead of applying constraints and restrictions for operations with certain types, we presented a proposal that favors best practice. Uncommon usage is not forbidden but greatly discouraged. Accordingly, we collected evidence regarding the usage of JavaScript operators by analyzing 400 libraries.

References

- [1] *DefinitelyTyped*. <http://definitelytyped.org/>.
- [2] *dts-gen: A TypeScript Definition File Generator*. <https://github.com/microsoft/dts-gen>.
- [3] Asger Feldthaus and Anders Møller. “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*.

Generation of TypeScript Declaration Files from JavaScript Code

- Edited by Andrew P. Black and Todd D. Millstein. ACM, 2014, pages 1–16. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660215. URL: <https://doi.org/10.1145/2660193.2660215>.
- [4] Zheng Gao, Christian Bird, and Earl T. Barr. “To Type or not to Type: Quantifying Detectable Bugs in JavaScript”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017*. Edited by Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard. IEEE, 2017, pages 758–769. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.75. URL: <https://doi.org/10.1109/ICSE.2017.75>.
 - [5] *Github Statistics*. <https://madnight.github.io/github>.
 - [6] Erik Krogh Kristensen and Anders Møller. “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*. Edited by Marieke Huisman and Julia Rubin. Volume 10202. Lecture Notes in Computer Science. Springer, 2017, pages 99–115. ISBN: 978-3-662-54493-8. DOI: 10.1007/978-3-662-54494-5_6. URL: https://doi.org/10.1007/978-3-662-54494-5_6.
 - [7] Erik Krogh Kristensen and Anders Møller. “Type Test Scripts for TypeScript Testing”. In: *PACMPL 1.OOPSLA (2017)*, 90:1–90:25. DOI: 10.1145/3133914. URL: <https://doi.org/10.1145/3133914>.
 - [8] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. “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*. Edited by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pages 488–498. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491447. URL: <https://doi.org/10.1145/2491411.2491447>.
 - [9] *TypeScript Language Specification*. <https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>.
 - [10] *TypeScript Compiler API*. <https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API>.
 - [11] *TypeScript Language Namespaces*. <https://www.typescriptlang.org/docs/handbook/namespaces.html>.

About the authors

Fernando Cristiani contact fc@example.org

Peter Thiemann thiemann@informatik.uni-freiburg.de