Generation of TypeScript Declaration Files from JavaScript Code

Fernando Cristiani
Hochschule Karlsruhe
Karlsruhe, Germany
fernando.cristiani@hs-karlsruhe.de

Peter Thiemann
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany
thiemann@informatik.uni-freiburg.de

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 populated and 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 directly from an NPM module and 70 of these were compared with the corresponding declaration file provided on DefinitelyTyped. We obtained positive results for all of these files, which exhibited no differences at all or differences that can be resolved by modifying the developer-provided examples.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

JavaScript, TypeScript, Dynamic Analysis, Declaration Files

1 INTRODUCTION

JavaScript is the most popular language for writing web applications [6]. 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 blog[1] 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 [10]. It has become a widely used alternative among JavaScript developers, because it incorporates features that are helpful for developing and maintaining large applications [5]. 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 to collect declaration files for popular JavaScript libraries. At the time of writing it contains declaration files for more than 6000 libraries.

---

[1] https://wtfjs.com

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

Listing 1: Unintuitive JavaScript behavior: Falsy values, `null` vs. `undefined`, `typeof`, and type coercion

Unfortunately, creation and maintenance of most declaration files in DefinitelyTyped is conducted manually, which is time consuming and error prone. Errors are aggravated because TypeScript takes a declaration file at face value. Discrepancies between declaration and implementing JavaScript library are not detected at compile time and result in incorrect behavior (e.g., autocompletion hints) of the IDE. As TypeScript does not perform any run-time checking of types, these discrepancies 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 by the developer as part of the documentation of a library. If these examples are sufficiently rich, then our toolchain can generate high-quality declaration files for the library. 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[2] module abs that "computes the absolute path of an input". Its documentation consists of the following three examples[3].

```
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 DefinitelyTyped[4]:

```
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:

(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.

The implementation and the results are available under the following repositories:

- https://github.com/proglang/run-time-information-gathering/tree/v1.1
- https://github.com/proglang/ts-declaration-file-generator/tree/v1.2
- https://github.com/proglang/ts-declaration-file-generator-service/tree/v1.2
- https://github.com/proglang/dts-generate-method/tree/v1.1
- https://github.com/proglang/dts-generate-results/tree/v1.2

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[5]. 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 (Section 5.2) can compare the new file for incompatibilities with the previous declaration file.

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 object described by an interface.

---

[2]NPM is the Node package manager, an online repository for JavaScript modules.
[3]https://www.npmjs.com/package/abs
[4]https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/abs/index.d.ts

---

[5]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;
  }

}
```

```
import globToRegExp from 'glob-to-regexp';

globToRegExp("*.min.js", {});
                         ⊘ extended
                         ⊘ flags              (property) GlobToRe…
                         ⊘ globstar
```

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[6]

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

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

Listing 2: Example for module-function template

TypeScript namespaces organize types declared in a declaration file and avoid name clashes with other types [11]. For example, the interface `I__opts` is declared in namespace `globToRegexp` in Figure 1. Hence its uses must be qualified by the namespace as in `globToRegexp.I__opts`.

This file is an instance of one of the standard templates for writing declaration files[7]: **module**, **module-class**, and **module-function**. Each template corresponds to a different way of organizing the exports of a JavaScript library. The choice of the template depends on the structure of the underlying JavaScript library:

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

There are further templates, but most libraries fall in one of these three categories. Both libraries `glob-to-regexp` and `abs` are instances of the **module-function** template.

The TypeScript project provides a guide on how to write high-quality declaration files[8]. 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 usage of the module. If the imported entity is used

---

[7]https://www.typescriptlang.org/docs/handbook/declaration-files/templates.html
[8]https://www.typescriptlang.org/docs/handbook/declaration-files/by-example.html

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

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

greet({
greeting: "hello world",
color: "#00ff00"
});
```

(a) Example for module-function 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

```
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

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

(d) Declaration file 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 = 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

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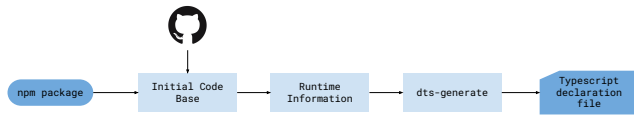
Figure 3: dts-generate - Architecture overview - The initial code base is retrieved from the NPM package's repository. Run-time information is gathered driven by example code. A TypeScript Declaration File is generated using run-time information.

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 [9] 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.

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 match 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 very simple and inspired by the `dts-gen` tool [2] (see Listing 2). The 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 execute the code, because the analysis modules provided by Jalangi can only gather information 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, we need to identify a dependent package, which is costly to download and instrument.

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 a scenario.

In the end, option 3 was the most viable option 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 showcase common use 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.
- Extract the code examples from the README file. To this end, observe that README files are written using Markdown[9], 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 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`.

---

[9]https://www.markdownguide.org

```
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

```
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`

- 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:

- 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 choose the template based on how the module is used. We analyze the properties `isExported`, `requiredModule`, and `isConstructor` from the information gathered at run time 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 basic types `string`, `number`, `boolean`, unions thereof, and optional elements were considered both for function parameters and interface properties. We also considered overloaded functions. Common types such as `any`, arrays, function callbacks, tuples, intersections (that is, types formed using the intersection operator `&`) and features such as generics, index signatures were not implemented.

The selection of typing features covered in the tool is driven by examination of the files in the DefinitelyTyped repository. We parsed each declaration file with `dts-parse` and tagged it according to the features used in the file. Table 1 summarizes the results.

Table 1: Distribution of TypeScript features in declaration files uploaded to DefinitelyTyped (DT). 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).

TypeScript feature	Count	%
type-string	5086	76.50%
optional-parameter	4915	73.93%
type-boolean	3891	58.53%
type-number	3699	55.64%
type-void	3548	53.37%
type-union	3456	51.99%
type-function	3286	49.43%
type-array	3264	49.10%
type-any	3127	47.04%
type-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%
type-object	661	9.94%
type-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%

We decided to prioritize the features with the highest number of occurrences and support them first in our framework.

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. As we extracted code examples written by developers, the quality of the generated declaration file is heavily dependent on these examples. For example, if a property of an interface is not accessed because a code example does not explore a particular execution path, then that property will not appear in the generated declaration file.
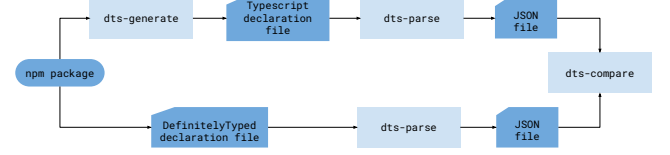

Figure 5: Evaluation of generated declaration files against DefinitelyTyped Repository - The parser uses the TypeScript Compiler API [3] to transform declaration files into an abstract syntax which is serialized in JSON. The comparison is performed on the JSON format.

As a measure of the quality of a generated file we use its distance to the declaration file provided on DefinitelyTyped for the same module. 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 hand-crafted definition files.

5.1 dts-parse

We are not interested in a textual comparison of declaration files, but in a comparison of the structures described by the files. Hence, the first step is to parse the declaration files using the TypeScript Compiler API to build and traverse the Abstract Syntax Tree of a TypeScript program [3]. This step also performs a sanity check of the generated declaration files as it rejects files with syntactic or semantic errors. The output of `dts-parse` is a structure similar to a symbol table 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 constructors and methods is made. Declaration files are tagged according to their features, which is useful for identifying and filtering out declaration files that contain unimplemented features.

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

5.2 dts-compare

The compare tool takes two declaration files as input and returns a JSON file containing the result of the comparison as shown in Listing 4. Following the naming convention of unit testing frameworks, 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 classify a difference as `solvable` if it can be resolved by adding more

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

Listing 4: Example for `dts-compare` executed on declaration files without differences

code examples. For example, we might add a function invocation with different parameters so that a missing interface property gets accessed. Otherwise it is classified as `unsolvable`.

We consider the following differences:

- `TemplateDifference`: A difference between the choice of TypeScript templates. For example, if the file in Definitely-Typed 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 properties of 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,
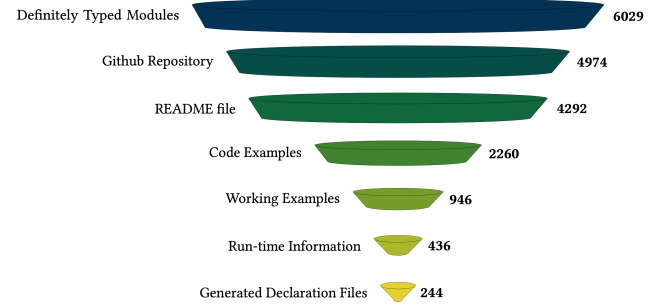

Figure 6: Number of analyzed modules for each stage of the experiment - A TypeScript Declaration File was generated for only 244 modules, out of 6029 modules in the Definite-lyTyped Repository. It was possible to gather valid run-time information for only 25% of the modules for which a Code Example was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

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.
- 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 70 modules that have only the features implemented by `dts-generate`. We obtained positive results for all of them. Section 6.2 provides a detailed explanation of the overall quality of the generated files. Section **??** 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:

- 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 2260 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, JalangiâĂŹs 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 JalangiâĂŹs 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 extracted code examples for these modules did not execute the library. Hence,

```
export = Abs;

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

(a) abs/index.d.ts - Generated

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

(b) abs/index.d.ts - DT

```
export = DirnameRegex;

declare function
    DirnameRegex(): RegExp
    ;
```

(c) dirname-regex/index.d.ts - Generated

```
export = dirnameRegex;

declare function
    dirnameRegex(): RegExp
    ;
```

(d) dirname-regex/index.d.ts - DT

```
export = EscapeHtml;

declare function EscapeHtml
    (string: string):
    string;
```

(e) escape-html/index.d.ts - Generated

```
declare function escapeHTML
    (text: string): string
    ;
declare namespace
    escapeHTML { }

export = escapeHTML;
```

(f) escape-html/index.d.ts - DT

Figure 7: Results for module-function template

the collected run-time information was not useful for generating a declaration file.

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. The different names in the export assignment by modules dirname-regex and escape-html do not matter because the module can receive an arbitrary name when it is imported: in `import Abs = require('abs');` the identifier Abs can be chosen by the programmer. We generate the name in the export assignment by transforming the module name into camel case form, following TypeScript guidelines.

Functions are correctly detected; input and output types are accurately inferred. We extract the parameter names 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, neither the order of declarations nor unused namespaces (as in module escape-html) matter.

Figure 8 contains an example for the **module** template, the declaration for the is-uuid module. The generated file only contain methods that were executed by the extracted examples. All invoked functions are correctly detected; the additional functions declared on DefinitelyTyped are not used in the example code. It would be easy to generate a perfectly matching declaration file by adding two examples using functions nil and anyNonNil.

```
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 - DT

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 5: Code example for `smart-truncate` module exporting a function

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`. 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 5 shows that the module is indeed exported as a function, as inferred by `dts-generate`.

6.2 Evaluation

We analyzed 70 of the 244 generated files. The remaining 174 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. All of the analyzed files contained only `solvable` differences. We manually completed the examples

```
'use strict';

// ...

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

// ...

}
```

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

```
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

```
export = GlobBase;

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

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

Figure 9: Incorrect optional parameter for module `glob-base`

for 10 of those modules. Applying `dts-generate` with the completed examples generates declaration files, which are equivalent to the files from DefinitelyTyped, as shown in Figure 10.

7 DISCUSSION

How can we improve further? Given that JavaScript developers are incentivized to develop good examples rather than writing "boring" type declarations, 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.

Literal types vs string. Many JavaScript libraries use literal strings to select options and configurations. To be concrete, 1925 DT packages rely on literal types (see Table 1). For example, the bonjour package[10] relies on strings to control events and indicate protocols in (different) interfaces:

```
    removeAllListeners(event?: 'up' | 'down'): this;
    protocol?: 'udp'|'tcp';
```

The carlo package[11] uses it to distinguish different types of events.

```
export type AppEvent = 'exit' | 'window';
```

---

[10]https://www.npmjs.com/package/@types/bonjour
[11]https://www.npmjs.com/package/@types/carlo

```
declare namespace gh {
  interface Options {
      enterprise?:
          boolean;
  }
  interface Result {
      ...
  }
}

declare function gh(url:
    string | {url: string
    }, options?: gh.
    Options): gh.Result |
    null;

export = gh;
```

(a) github-url-to-object/index.d.ts - DefinitelyTyped. Properties of interface `Result` were ignored for readability, since return types were not considered.

```
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://githuuub.com/monkey/business.
    git', {});
gh('git+https://githuuub.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. The literal interface in DefinitelyTyped is replaced by a declared interface. Return types are not considered.

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`. Both alternatives come with the problem that they create a local over-approximation, so that it would become harder to qualify the output of the generator. A further alternative that does not suffer from this drawback would be to communicate to the developer a set of typical example strings, *marker strings*, that the generator always generalizes 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 communicate 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

dts-gen. TypeScript comes with `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 the 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 most parameters and results being inferred as `any`.

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 of discrepancies between the declaration file and the implementation.

TSInfer & TSEvolve. TSInfer and TSEvolve are presented as part of TSTools [7]. Both tools are the continuation of TSCheck [4], a tool for detecting mismatches between a declaration file and the implementation of the module.

TSInfer 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 TSInfer 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 [8]. 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.

REFERENCES

[1] [n.d.]. DefinitelyTyped. http://definitelytyped.org/.
[2] [n.d.]. dts-gen: A TypeScript Definition File Generator. https://github.com/microsoft/dts-gen.
[3] [n.d.]. TypeScript Compiler API. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API.
[4] Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 1–16. https://doi.org/10.1145/2660193.2660215
[5] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. 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*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 758–769. https://doi.org/10.1109/ICSE.2017.75
[6] Github Statistics [n.d.]. Github Statistics. https://madnight.github.io/githut.
[7] Erik Krogh Kristensen and Anders Møller. 2017. Inference and Evolution of TypeScript Declaration Files. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10202)*, Marieke Huisman and Julia Rubin (Eds.). Springer, 99–115. https://doi.org/10.1007/978-3-662-54494-5_6
[8] Erik Krogh Kristensen and Anders Møller. 2017. Type Test Scripts for TypeScript Testing. *PACMPL* 1, OOPSLA (2017), 90:1–90:25. https://doi.org/10.1145/3133914
[9] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 488–498. https://doi.org/10.1145/2491411.2491447
[10] TypeScript [n.d.]. TypeScript Language Specification. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.
[11] TypeScript Namespaces [n.d.]. TypeScript Language Namespaces. https://www.typescriptlang.org/docs/handbook/namespaces.html.