Generation of TypeScript Declaration Files from JavaScript Code

John Q. Public
Dummy University Computing Laboratory, Country
My second affiliation, Country
`http://www.myhomepage.edu`
johnqpublic@dummyuni.org

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. 72 files presented positive results, having no differences at all or differences that can be solved by modifying the code examples accordingly.

2012 ACM Subject Classification Software and its engineering → Software notations and tools; Software and its engineering → General programming languages

Keywords and phrases JavaScript, TypeScript, Dynamic Analysis, Declaration Files

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *John Q. Public*: (Optional) author-specific funding acknowledgements

1 Introduction

JavaScript has become the most popular language for writing web applications [6]. It is also gaining popularity 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. Hence, it lacks features for maintaining and evolving large codebases. Presently, however, JavaScript is being used for creating large and complex applications. 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 such dynamic typing (which includes unchecked property names) is not present in other languages that use build tools based on type information. This insight motivated the creation of TypeScript, a superset of JavaScript with type annotations

[1] `https://wtfjs.com`

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Listing 1 Unintuitive JavaScript behavior - falsy values, **`typeof`**, **`null`** and **`undefined`** operators and type coercion

```
"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
```

[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 auto-completion in an IDE.

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. Lots of declaration files for popular JavaScript libraries have been created in a community effort in the DefinitelyTyped repository [1]. At the time of writing this repository contains declaration files for more than 6000 JavaScript libraries. Unfortunately, most declaration files in this libraries have been manually created and maintained, 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 inaccurate code-intelligence features are misleading the programmer. Moreover, TypeScript does not perform any run-time checking on types, either, so that a discrepancy between the declaration file and its corresponding JavaScript library can lead to unexpected behavior and crashes. Such behavior can lead to developer frustration, longish debugging sessions, and decreasing confidence in the tool chain.

Some previous work tackled the problem of improving the quality of declaration files. Feldthaus et al [4] search automatically for mismatches between a declaration file and implementation code. TSTest [8] adapted feedback-directed random testing to detect discrepancies between a declaration file and a JavaScript library. Tools like TSInfer and TSEvolve [7] are designed for assisting the creation of new declaration files and supporting the evolution the declaration file when the corresponding JavaScript library gets modified, respectively. They rely on an existing static analyzer for JavaScript [**?**]. TypeScript itself developed `dts-gen`, a tool that generates a declaration file that is meant to be used as a *starting point for writing a high-quality declaration file* [2].

In this work we present the tool `dts-generate` as a first step to explore the possibilities for generating useful declaration files without the heavy lifting of static analysis. `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 yearly 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 contributions of this paper are as follows.

- A framework that extracts code examples from the documentation of an NPM package and collects run-time type information from running these examples.
- The tool `dts-generate`, a command line application that generates a valid TypeScript declaration file for a specific NPM package using run-time information.
- A comparator for TypeScript declaration files. This tool is necessary for evaluating our framework and also useful to detect incompatibilities when evolving JavaScript modules.
- An evaluation of our framework. 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 into a regular expression[2]. It has about 6,500,000 weekly downloads and 181 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. We use `dts-generate` to 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 it (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.

After filling in some background information on declaration files, Section 4 examines each step in the generation process in detail and refer back to this example for concreteness.

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 are used for organizing types declared in a declaration file and avoiding name collisions with other types [11]. For example, the interface `I__opts` declared in Figure 1 is used as `globToRegexp.I__opts`, since it is declared within a namespace.

This file is an instance of one of the standard templates for writing declaration files[4]: **module**, **module-class** and **module-function**. Each template corresponds to a different way of describing the exports of a JavaScript library. Choosing the template depends on the particular organization of the underlying JavaScript library:

[2] https://www.npmjs.com/package/glob-to-regexp
[4] https://www.typescriptlang.org/docs/handbook/declaration-files/templates.html

```
export = GlobToRegexp;

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

}
```

Figure 1 Declaration file for `glob-to-regexp` generated with `dts-generate` - Interfaces is correctly detected. Optional parameters are explicitly declared as `undefined`. Declaration file can be correctly used in Visual Studio Code[3].

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

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

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

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

TypeScript provides a guide on how to write high quality declaration files[5]. They explain the main concepts through examples. We extracted one example for each template and generated used `dts-generate` to generate a corresponding declaration file, as presented in Figure 2. The `glob-to-regexp` library is an instance of a **module-function**. It is worth mentioning that templates are a way of describing how a JavaScript library is intended to be used. Although highly unlikely, it is possible to create a library that can be represented with a different template depending on how it is used, as shown in Figure 3. Run-time analysis proves useful in this case, since it is not sufficient to statically analyze the exported module.

4 The Generation of TypeScript Declaration Files

This section gives an overview of our approach to generating TypeScript declaration files from JavaScript libraries packaged in NPM. Figure 4 gives a rough picture of the inner working

[5] 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) TypeScript 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 | undefined
        ;
    'color': undefined | 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`

```
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`

```
export = Greeter;

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

declare namespace Greeter {
}
```

(f) Declaration file for `module-class` template

Figure 2 Results for **module-function** template

```
var multiPurposeModule = require('./multi-purpose-module');

// module-function
console.log(multiPurposeModule("John", "Doe"));
// John Doe


// module-class
var m = new multiPurposeModule("Jane", "Doe");
console.log(m.firstName + " - " + m.lastName);
// Jane - Doe

console.log(m.sayHello());
// Hello, my name is Jane Doe

// module
console.log(multiPurposeModule.anotherFunction());
// I am another function!
```

(a) index.js

```
var multiPurposeModule = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;

  return firstName + " " + lastName;
};

multiPurposeModule.prototype.sayHello = function() {
  return "Hello, my name is " + this.firstName + " " +
      this.lastName;
};

multiPurposeModule.anotherFunction = function() {
  return "I am another function!";
};

module.exports = multiPurposeModule;
```

(b) multi-purpose-module/index.js

Figure 3 Example module that implements all three templates simultaneously

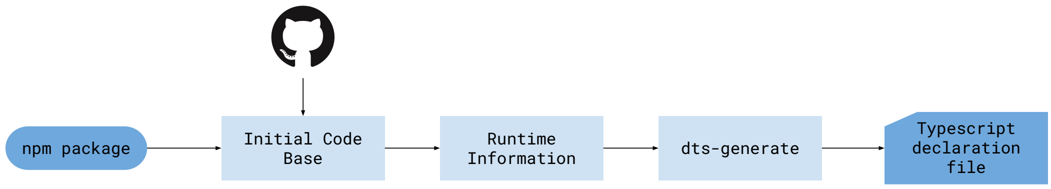

Figure 4 dts-generate - Architecture overview - Initial code base is retrieved from the NPM package's repository. A valid TypeScript Declaration File is generated using run-time information. A Symbolic Execution Engine creates test cases based on the generated Declaration File and via a feedback loop enriches the code base until the stopping criteria is reached. The final TypeScript Declaration File gets returned. Feedback loop through the Symbolic Execution Engine was not implemented. It can be added in a future to the existing architecture, without modifying the working blocks.

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 runtime. 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.

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 content mimics 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` package [2]. Listing 2 shows that invoking the package is very simple and 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 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.

Listing 3 Extracted example for `glob-to-regexp`

```
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
```

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 even not allowed 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. This observation holds in particular for NPM packages, which are generally created to solve a specific problem of JavaScript development. These code fragments do not present the problem of option 2 since they are meant to be used by developers using 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[6], a popular markup language. In such a file, it is customary to write code examples 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.

The extracted example for the case of `glob-to-regexp` is shown in Listing 3.

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 providing a useful initial code base with meaningful examples, thus avoiding possible cold start problems.

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

4.2 Run-time Information Gathering

The Runtime Information block described in Figure 4 gathers information such as:

- 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 get triggered with virtually any JavaScript event. Our instrumentation observes the following events:

- Binary operations, like `==`, `+` or `===`.
- Variable declaration.
- Function, method, or constructor invocation.
- 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 around a wrapper object, which enables to store meta-information and greatly simplifies the mapping of an observation with the corresponding `ArgumentContainer` or `interaction`. We then use the original values for critical operators such as `===` or `typeof`, for which wrapper objects will definitely 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 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 actual generation of the declaration file (cf. Figure 4). 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`

```
var multiPurposeModule = require('./multi-purpose-module');

multiPurposeModule("John", "Doe");
// John Doe
```

(a) Example module `multi-purpose-module` used as `module-function`

```
export = MultiPurposeModule;

declare function MultiPurposeModule(firstName: string, lastName: string): string;
```

(b) Generated declaration file with `module-function` template

```
var multiPurposeModule = require('./multi-purpose-module');

var m = new multiPurposeModule("Jane", "Doe");
var fullName = m.firstName + " - " + m.lastName;
// Jane - Doe

m.sayHello();
// Hello, my name is Jane Doe
```

(c) Example module `multi-purpose-module` used as `module-class`

```
export = MultiPurposeModule;

declare class MultiPurposeModule {
  constructor(firstName: string, lastName: string);
  sayHello(): string;
}

declare namespace MultiPurposeModule {
}
```

(d) Generated declaration file with `module-class` template

```
var multiPurposeModule = require('./multi-purpose-module');

multiPurposeModule.anotherFunction();
// I am another function!
```

(e) Example module `multi-purpose-module` used as `module`

```
export function anotherFunction(): string;
```

(f) Generated declaration file with `module` template

Figure 5 Different ways of using example module presented in Figure 3. `dts-generate` generates infers the correct template for each case. The JavaScript implementation of the module remained unmodified for each example.

if the function is used as a constructor. If not, we choose `module-function`. Otherwise, we infer the `module` template. Figure 5 exposes this logic using the example provided in Figure 3.

We implemented only basic TypeScript features and we focused our effort in building and end-to-end solution that generates valid and useful declaration files. Only the basic types `string`, `number`, `boolean` and its union 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 was tagged with `dts-parse` and those containing unimplemented features were ignored. We add the `undefined` type for describing an optional parameter instead of adding the corresponding token `?`, which is also done by TypeScript in strict null checking mode[12].

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 in each recursion level.

```
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 6 Results for **module** module `is-uuid`

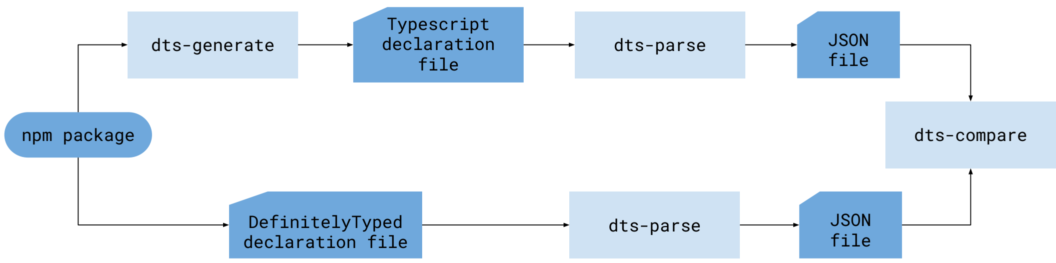

Figure 7 Evaluation of generated declaration files against DefinitelyTyped Repository - A parser transforms the generated declaration file and the equivalent file in the DefinitelyTyped repository into a JSON file using the TypeScript Compiler API [3]. Comparison is then performed on the JSON files, i.e. not on the declaration files.

5 Evaluation

After generating a declaration file for an NPM package, we need to evaluate its quality. To this end, we created two tools:

- `dts-parse`: A TypeScript declaration files parser, which transforms the file into a JSON file with a defined structure.
- `dts-compare`: A tool that uses `dts-parse` and compares two TypeScript declaration files computing differences by applying several criteria.

We applied `dts-compare` to compare the generated declaration file against the one uploaded to the DefinitelyTyped repository for the same module, as shown in Figure 7. 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 creating a definition file.

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.

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

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

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

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

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 [3]. 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 a JSON, as shown in Listing 4.

5.2 dts-compare

The tool is written in TypeScript and receives 2 declaration files as an input and returns a JSON 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 flagged as `expected` or `actual`. We assigned the `expected` flag to the DefinitelyTyped declaration file.

`dts-compare` applies `dts-parse` internally to both files to get a common structure eligible for comparison. The result of the comparison is an array of an abstract entity called `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. Consequently, in order to isolate the impact of incomplete code examples on our quality analysis we identified each difference as follows:

- `solvable`: Differences that can be solved by expanding the code examples. For example, invoking a function with different parameters so that a missing interface property gets accessed.
- `unsolvable`: Differences that can not be solved are marked as `unsolvable`. These differences are implementation errors of `dts-generate`.

The concrete evaluated differences are as follows:

- `TemplateDifference`: A difference between the formal TypeScript templates. For example, if the file in DefinitelyTyped is written with `module` and we generated a file

using `module-function`. 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 is present in the DefinitelyTyped file but not in the generated one. This applies to functions and methods of classes and interfaces.
- `FunctionMissingDifference`: A function is present in the generated declaration file but not in DefinitelyTyped.
- `FunctionOverloadingDifference`: The number of declarations for the same functions is different. This is particularly common when developers declare a function multiple times with different single types instead of using union types.
- `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 better code examples. This includes a basic type that is converted to a union type, a function overloading or a parameter or property that is marked as optional.
  - `UnsolvableDifference`: Anything not considered as `SolvableDifference`.

Type aliases are invisible to `dts-compare` as only final types were considered: the declaration `type T = string | number; declare function F(a: T);` is equivalent to `declare function F(a: string | number);`. The same concept applies for literal interfaces. `dts-compare` does not contemplate differences in function return types, since interface inference of function return types is not covered in this 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 runtime 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 one of the generated files was compared against the corresponding declaration file already uploaded to the repository.

Figure 8 shows that a declaration file was generated for 244 modules out of 6029 modules and we identified 80 modules that have only the features implemented by `dts-generate`.

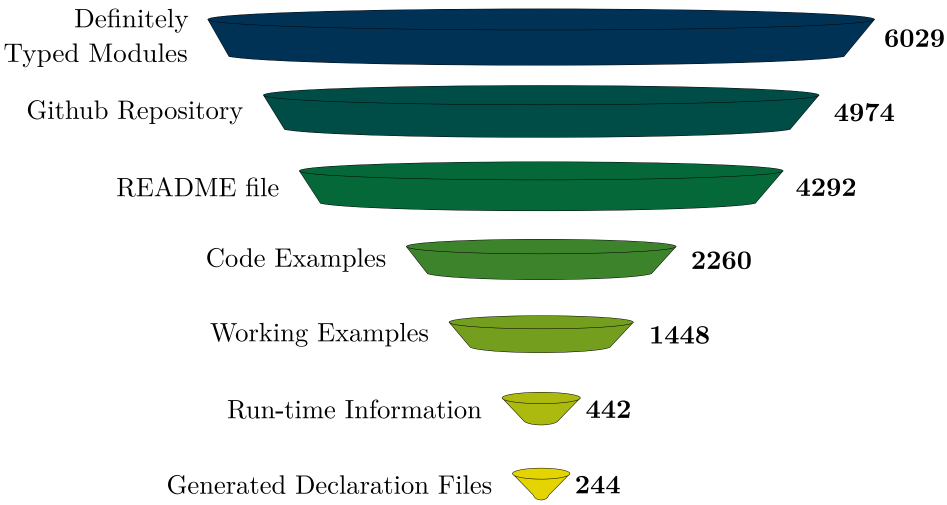

Figure 8 Number of analyzed modules for each stage of the experiment - A TypeScript Declaration File was generated for only 244 modules, out of 6029 modules in the DefinitelyTyped Repository. It was possible to gather valid run-time information for only 25% of the modules for which a Code Example was extracted.

We obtained positive results for 69 of them. A detailed explanation of the overall quality of the generated files is provided in Section 6.3 - Evaluation. Samples of the generated declaration files for templates **module**, **module-class** and **module-function** are presented in Section 6.2 - Declaration Files Generation.

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 8, 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 repository URL.
- Extracting README file.
- Extracting code examples from README files.
- Executing code examples and discarding failing ones.

The results obtained for each on of them 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 by installing the required packages and running the code as a NodeJS application. Unfortunately, the code examples only worked for 1448 modules. 812 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 not faulty. It was executing 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 442 out of 1448 modules with working code examples. As explained, in order to extract the run-time information, the behavior of the code under analysis was explicitly modified by wrapping the arguments around Wrapper Objects. Furthermore, Jalangi's instrumentation itself caused some executions to fail, since the modules contained JavaScript features that are not supported by Jalangi. As a result, run-time information could not be extracted for 1006 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 442 modules. Despite the correct execution of instrumented code examples, 198 modules did not contain suitable run-time information for generating a declaration file. The extracted code examples for these modules did not execute the library itself and therefore the collected run-time information was not useful for generating a declaration file.

6.2 Declaration Files Generation

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

Figure 9 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. 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.

Differences in the export assignment by modules `dirname-regex` and `escape-html` is not problematic since the exported module can receive a custom name given by the developer. `dts-generate` automatically generates the export assignment by transforming

```
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 - DefinitelyTyped

```
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 - DefinitelyTyped

```
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 - DefinitelyTyped

Figure 9 Results for **module-function** template

```
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 10 Results for **module** module `is-uuid`

the module name into camel case form, following TypeScript guidelines. The parameter names are extracted from the variable names of the JavaScript code. The JavaScript concrete implementation of module `escape-html` declares the first parameter of the exported function with the name `string` and not `text`, as specified in DefinitelyTyped. A difference in the names of the function parameters will anyway not affect the code compilation in any way. Furthermore, the omitted namespace in module `escape-html` is irrelevant. It is actually not required, since there is nothing declared within the namespace.

A sample for the **module** template for the `is-uuid` module is shown in Figure 10. Again, methods that were not executed by the extracted examples are not included in the declaration file.

Finally, we could not generate a correct declaration file with the template `module-class`. All the modules corresponding to this template had at least one of the unimplemented TypeScript features.

```
export = GlobBase;

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

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

```
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

```
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

```
'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 11 Incorrect optional parameter for module `glob-base`

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 that parameter an Error is thrown at runtime. The file generated by `dts-generate` will never mark the parameter as optional, as shown in Figure 11. Function return types interfaces are not inferred by `dts-generate`. Parameter name is also inferred as `pattern` instead of `basePath`, since the JavaScript implementation declares the parameter as `pattern`, as shown in Figure 11d. We even 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. We see in Listing 6 that the module is indeed exported as a function, as inferred by `dts-generate`.

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

```
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 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.

```
$ 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;
```

6.3 Evaluation

We analyzed only 82 of the 244 generated files. The remaining 162 files contained at least one of the unimplemented TypeScript features and were therefore not considered. These features were identified by `dts-parse` as tags.

As shown in Figure 10, 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 72 such modules. We completed the examples manually for 10 of those modules, making them equivalent to the DefinitelyTyped files, as shown in Figure 12. Expected differences are present: optional parameters are declared both with the actual and `undefined` type and the union type is handled through multiple overloads.

We identified only 10 modules with `unsolvable` differences. Most of them were caused for the same error: 8 modules had an incorrectly declared interface for the `string` type. This interface contained standard properties or methods from the JavaScript `String` object, such as `length` or `indexOf()`. The remaining modules are `duplexer2` and `duplexer3`. Both of them used native NodeJS types within the declaration file and `dts-generate` could not identify those properly.

7 Related Work

Microsoft's dts-gen

Microsoft developed `dts-gen`, a tool that creates starter declaration files for JavaScript libraries [2]. Its documentation states that the result is however intended to be only used a starting point. The outcome needs to be refined afterwards by the developers.

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`.

The solution presented in this work, however, 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.

Formal aspects like applying the right template and using the correct syntax are perfectly covered by `dts-gen`.

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

  export interface I__opts {
    'enterprise': undefined | 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'});
```

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

Figure 12 Manually expanded code example for `github-url-to-object` to match DefinitelyTyped declaration file. Literal interface in DefinitelyTyped gets replace by a declared interface. The `undefined` type is used instead of the optional type. Return types are not considered.

TSInfer & TSEvolve

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

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

The abstraction and the constraints they 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.

Since they analyze the objects and functions stored in the snapshot, they faced the problem of including in the declaration file internal methods and properties that developers wanted to hide. Run-time information would have informed that the developer has no intention of exposing such methods.

Moreover, TSEvolve performs a differential analysis on the changes made to a JavaScript library in order to determine intentional discrepancies between declaration files of two consecutive versions. We consider that a differential analysis may not be needed. If the developer's intention is accurately extracted and the execution code clearly represents that intention 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 a JavaScript implementation [8]. It applies feedback-directed random testing for generating type test scripts. These scripts will execute the library in order to check if it behaves the way it is described in the declaration file. TSTest also provides concrete executions for mismatches.

We evaluated the generated declaration files comparing them to the declaration files uploaded to DefinitelyTyped. The disadvantage of doing this is that since the uploaded files are written manually, they could already contain mismatches with the JavaScript implementation. However, it is a suitable choice for a development stage since it is used as a baseline.

In a final stage, declaration files need to be checked against the proper JavaScript implementation and TSTest has to be definitely taken into account.

8 Conclusion

We have presented `dts-generate`, a tool for generating a TypeScript declaration file for a specific JavaScript library. The tool downloads code samples written by the developers from the library's repository. It uses these samples 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 a future incorporation of a Symbolic Execution Engine that refines the initial code base enabling the exploration of new execution paths. However 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. Since developers expose through code how a library should be used, 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 where common practices are favored. Uncommon usage is not forbidden but greatly disfavored. Accordingly, we collected evidence regarding the usage of JavaScript operators by analyzing 400 libraries.

Finally, the architecture is composed of different blocks that interact with each other. Each block is independent and has a well defined behavior as well as clear input and output values. As a result, each block can be independently and simultaneously improved.

References

1 Definitelytyped. `http://definitelytyped.org/`.

2 dts-gen: A typescript definition file generator. `https://github.com/microsoft/dts-gen`.

3 Typescript compiler api. `https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API`.

4 Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 1–16. ACM, 2014. `doi:10.1145/2660193.2660215`.

5 Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 758–769. IEEE / ACM, 2017. `doi:10.1109/ICSE.2017.75`.

6 Github statistics. `https://madnight.github.io/githut`.

7 Erik Krogh Kristensen and Anders Møller. Inference and evolution of TypeScript declaration files. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2017. `doi:10.1007/978-3-662-54494-5\_6`.

8 Erik Krogh Kristensen and Anders Møller. Type test scripts for TypeScript testing. *PACMPL*, 1(OOPSLA):90:1–90:25, 2017. `doi:10.1145/3133914`.

9 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,* *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. doi:10.1145/2491411.2491447.

10 Typescript language specification. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.

11 Typescript language namespaces. https://www.typescriptlang.org/docs/handbook/namespaces.html.

12 Typescript language optional parameters and properties. https://www.typescriptlang.org/docs/handbook/advanced-types.html#optional-parameters-and-properties.