Inference and Evolution of TypeScript Declaration Files

Erik Krogh Kristensen and Anders Møller

Aarhus University, Denmark
{erik,amoeller}@cs.au.dk

Abstract. TypeScript is a typed extension of JavaScript that has become widely used. More than 2 000 JavaScript libraries now have publicly available TypeScript declaration files, which allows the libraries to be used when programming TypeScript applications. Such declaration files are written manually, however, and they are often lagging behind the continuous development of the libraries, thereby hindering their usability. The existing tool TSCHECK is capable of detecting mismatches between the libraries and their declaration files, but it is less suitable when creating and evolving declaration files.
In this work we present the tools TSINFER and TSEVOLVE that are designed to assist the construction of new TypeScript declaration files and support the co-evolution of the declaration files as the underlying JavaScript libraries evolve. Our experimental results involving major libraries demonstrate that TSINFER and TSEVOLVE are superior to TSCHECK regarding these tasks and that the tools are sufficiently fast and precise for practical use.

1 Introduction

The TypeScript [13] programming language has become a widely used alternative to JavaScript for developing web applications. TypeScript is a superset of JavaScript adding language features that are important when developing and maintaining larger applications. Most notably, TypeScript provides optional types, which not only allows many type errors to be detected statically, but also enables powerful IDE support for code navigation, auto-completion, and refactoring. To allow TypeScript applications to use existing JavaScript libraries, the typed APIs of such libraries can be described in separate *declaration files*. A public repository exists containing declaration files for more than 2 000 libraries, and they are a critical component of the TypeScript software ecosystem.[1]

Unfortunately, the declaration files are written and maintained manually, which is tedious and error prone. Mismatches between declaration files and the corresponding JavaScript implementations of libraries affect the TypeScript application programmers. The type checker produces incorrect type error messages, and code navigation and auto-completion are misguided, which may cause programming errors and increase development costs. The tool TSCHECK [8] has

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

been designed to detect such mismatches, but three central challenges remain. First, the process of constructing the initial version of a declaration file is still manual. Although TypeScript has become popular, many new libraries are still being written in JavaScript, so the need for constructing new declaration files is not diminishing. We need tool support not only for checking correctness of declaration files, but also for assisting the programmers creating them from the JavaScript implementations. Second, JavaScript libraries evolve, as other software, and when their APIs change, the declaration files must be updated. We observe that the evolution of many declaration files lag considerably behind the libraries, which causes the same problems with unreliable type checking and IDE support as with erroneous declaration files, and it may make application programmers reluctant or unable to use the newest versions of the libraries. With the increasing adaptation of TypeScript and the profusion of libraries, this problem will likely grow in the future. For these reasons, we need tools to support the programmers in this co-evolution of libraries and declaration files. Third, TSCHECK is not sufficiently scalable to handle modern JavaScript libraries, which are often significantly larger than a couple of years ago.

The contributions of this paper are as follows.

- To further motivate our work, we demonstrate why the state-of-the-art tool TSCHECK is inadequate for inference and evolution of declaration files, and we describe a small study that uncovers to what extent the evolution of TypeScript declaration files typically lag behind the evolution of the underlying JavaScript libraries (Section 2).
- We present the tool TSINFER, which is based on TSCHECK but specifically designed to address the challenge of supporting programmers when writing new TypeScript declaration files for JavaScript libraries, and to scale to even the largest libraries (Section 3).
- Next, we present the tool TSEVOLVE, which builds on top of TSINFER to support the task of co-evolving TypeScript declaration files as the underlying JavaScript libraries evolve (Section 4).
- We report on an experimental evaluation, which shows that TSINFER is better suited than TSCHECK for assisting the developer in creating the initial versions of declaration files, and that TSEVOLVE is superior to both TSCHECK and TSINFER for supporting the co-evolution of declaration files (Section 5).

2 Motivating Examples

The PixiJS library PixiJS[2] is a powerful JavaScript library for 2D rendering that has been under development since 2013. A TypeScript declaration file[3] was written manually for version 2.2 (after some incomplete attempts), and the authors have since then made numerous changes to try to keep up-to-date with the rapid evolution of the library. At the time of writing, the current version of PixiJS is 4.0, and the co-evolution of the declaration file continues to require

[2] `http://www.pixijs.com/`
[3] `https://github.com/pixijs/pixi-typescript`

```
export class Sprite extends PIXI.DisplayObjectContainer {
    constructor (texture: PIXI.Texture);
    static fromFrame: (frameId: string | number) => PIXI.Sprite;
    static fromImage: (imageId: string, crossorigin: any,
                       scaleMode: any) => PIXI.Sprite;
    _height: number;
    _width: number;
    anchor: PIXI.Point;
    blendMode: number;
    onTextureUpdate: () => void;
    setTexture: (texture: PIXI.Texture) => void;
    shader: any;
    texture: PIXI.Texture;
    tint: number;
}
```

Fig. 1. Example output from TSINFER, when run on PixiJS version 2.2.

substantial manual effort as testified by the numerous commits and issues in the repository. Hundreds of library developers face similar challenges with building TypeScript declaration files and updating them as the libraries evolve.

From checking to inferring declaration files To our knowledge, only one tool exists that may alleviate the manual effort required: TSCHECK [8]. This tool detects mismatches between a JavaScript library and a TypeScript declaration file. It works in three phases: (1) it executes the library's initialization code and takes a snapshot of the resulting runtime state; (2) it then type checks the objects in the snapshot, which represent the structure of the library API, with respect to the TypeScript type declarations; (3) it finally performs a light-weight static analysis of each library function to type check the return value of each function signature. This works well for detecting errors, but not for inferring and evolving the declaration files. For example, running TSCHECK on PixiJS version 2.2 and a declaration file with an empty `PIXI` module (mimicking the situation where the module is known to exist but its API has not yet been declared) reports nothing but the missing properties of the `PIXI` module, which is practically useless. In comparison, our new tool TSINFER is able to infer a declaration file that is quite close to the manually written one. Figure 1 shows the automatically inferred declaration for one of the classes in PixiJS version 2.2. The declaration is not perfect (the types of `frameId`, `crossorigin`, `scaleMode`, and `shader` could be more precise), but evidently such output is a better starting point when creating the initial version of a declaration file than starting completely from scratch.

Evolving declaration files The PixiJS library has recently been updated from version 3 to version 4. Using TSCHECK as a help to update the declaration file would not be particularly helpful. For example, running TSCHECK on version 4 of the JavaScript file and the existing version 3 of the declaration file reports that 38 properties are missing on the `PIXI` object, without any information about their types. Moreover, 15 of these properties are also reported if running TSCHECK on

```
Property PrimitiveShader removed from object on window.PIXI
Property FXAAFilter removed from object on window.PIXI
Property TransformManual added to object on window.PIXI
  Type: typeof PIXI.TransformBase
Property TransformBase added to object on window.PIXI
  Type:  class TransformBase { ... }
Property Transform added to object on window.PIXI
  Type: class Transform extends PIXI.TransformBase { ... }
```

(a) Some of the added or removed properties.

```
Type changed on
  window.PIXI.RenderTarget.[constructor].[return].stencilMaskStack
from StencilMaskStack to PIXI.Graphics[]
```

(b) A modified property.

Fig. 2. Example output from TSEVOLVE, when run on PixiJS versions 3 and 4.

version 3 of the JavaScript file, since they are due to the developers intentionally leaving some properties undocumented. Our experiments presented in Section 5 show that many libraries have such intentionally undocumented features, and some also have properties that intentionally exist in the declaration file but not in the library.[4] While TSINFER does suggest a type for each of the new properties, it does not have any way to handle the intentional discrepancies. Our other tool TSEVOLVE attempts to solve that problem by looking only at differences between two versions of the JavaScript implementation and is thereby better at only reporting actual changes. When running TSEVOLVE on PixiJS version 3 and 4, it reports (see Figure 2(a)) that 8 properties have been removed and 24 properties have been added on the `PIXI` object. All of these correctly reflect an actual change in the library implementation, and the declaration file should therefore be updated accordingly. This update inevitably requires manual intervention, though; in this specific case, `PrimitiveShader` has been removed from the `PIXI` object but the developers want to keep it in the declarations as an internal class, and `TransformManual`, although it is new to version 4, is a deprecated alias for the also added `TransformBase`.

Changes in a library API from one version to the next often consist of extensions, but features are also sometimes removed, or types are changed. As an example of the latter, one of the changes from version 3 to 4 for PixiJS was changing the type of the field `stencilMaskStack` in the class `RenderTarget` from type `PIXI.StencilMaskStack` to type `PIXI.Graphics[]`. The developer updating the declaration file noticed that the field was now an array, but not that the elements were changed to type `PIXI.Graphics`, so the type was erroneously updated to `PIXI.StencilMaskStack[]`. In comparison, TSINFER reports the change correctly as shown in Figure 2(b).

[4] This situation is rare, but can happen if, for example, documentation is needed for a class that is not exported, see e.g. `https://github.com/pixijs/pixi.js/issues/2312/#issuecomment-174608951`.

A study of evolution of type declarations To further motivate the need for new tools to support the co-evolution of declaration files as the libraries evolve, we have measured to what extent existing declaration files lag behind the libraries.[5] We collected every JavaScript library that satisfies the following conditions: it is being actively developed and has a declaration file in the the DefinitelyTyped repository, the declaration file contains a recognizable version number, and the library uses git tags for marking new versions, where we study the commits from January 2014 to August 2016. This resulted in 49 libraries. By then comparing the timestamps of the version changes for each library and its declaration file, respectively (where we ignore patch releases and only consider major.minor versioning), we find that for more than half of the libraries, the declaration file is lagging behind by at least a couple of months, and for some more than a year. This is notable, given that all the libraries are widely used according to the github ratings, and it seriously affects the usefulness of the declaration files in TypeScript application development.

Interestingly, we also find many cases where the version number found in the declaration file has not been updated correctly along with the contents of the file.[6] Not being able to trust version numbers of course also affects the usability of the declaration files. For some high-profile libraries, such as jQuery and AngularJS, the declaration files are kept up-to-date, which demonstrates that the developers find it necessary to invest the effort required, despite the lack of tool support. We hope our new tools can help not only those developers but also ones who do not have the same level of manual resources available.

Scalability In addition to the limitations of TSCHECK described above, we find that its static analysis component, which we use as a foundation also for TSINFER and TSEVOLVE, is not sufficiently scalable to handle the sizes and complexity of contemporary JavaScript libraries. In Section 3.2 we explain how we replace the unification-based analysis technique used by TSCHECK with a more precise subset-based one, and in Section 5 we demonstrate that this modification, perhaps counterintuitively, leads to a significant improvement in scalability. As an example, the time required to analyze *Moment.js* is improved from 873 seconds to 12 seconds, while other libraries simply are not analyzable in reasonable time with the unification-based approach.

3 TSINFER: Inference of Initial Type Declarations

Our inference tool TSINFER works in three phases: (1) it concretely initializes the library in a browser and records a snapshot of the resulting runtime state, much like the first phase of TSCHECK (see Section 2); (2) it performs a static analysis of all the functions in that snapshot, similarly to the third phase of TSCHECK; (3) lastly it emits a TypeScript declaration file. As two of the phases are quite similar to the approach used by TSCHECK, we here focus on what TSINFER does differently.

[5] Our data material from this study is available at `http://www.brics.dk/tstools/`.

[6] An example is Backbone.js, until our patch `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/10462`.

3.1 The Snapshot Phase

In JavaScript, library code needs to actively put entry points into the heap in order for it to be callable by application code. This initialization, however, often involves complex metaprogramming, and statically analyzing the initialization of a library like jQuery can therefore be extremely complicated [2]. We sidestep this challenge by concretely initializing the library in a real browser and recording a snapshot of the heap after the top-level code has finished executing. This is done in the same way as described by TSCHECK, and we work under the same assumptions, notably, that the library API has been established after the top-level code has executed. We have, however, changed a few things.

For all functions in the returned snapshot, we record two extra pieces of information compared to TSCHECK: (1) the result of calling the function with the `new` operator (if the call returned normally), which helps us determine the structure of a class if the function is found to be a constructor; (2) all calls to the function that occur during the initialization, which we use to seed the static analysis phase.

The last step is to create a class hierarchy. JavaScript libraries use many different and complicated ways of creating their internal class structures, but after the initialization is done, the vast majority of libraries end up with constructor functions and prototype chains. The class hierarchy is therefore created by making a straightforward inspection of the prototype chains.

3.2 The Static Analysis Phase

The static analysis phase takes the produced snapshot as input and performs a static analysis of each of the functions. It produces types for the parameters and the return value of each function.

The analysis is an unsound, flow-insensitive, context-insensitive analysis that has all the features described in previous work [8], including the treatment of properties and native functions. There are, however, some important changes.

TSCHECK analyzes each function separately, meaning that if a function `f` calls a function `g`, this information is ignored when analyzing function `g`. This works well for creating an analysis such as TSCHECK that only infers the return type of functions. When creating an analysis that also infers function parameter types, the information gained by observing calls to a function is important. Our analysis therefore does not analyze each function separately, but instead performs a single analysis that covers all the functions.

While TSCHECK opts for a unification-based analysis, we find that switching to a subset-based analysis is necessary to gain the scalability needed to infer types for the bigger JavaScript libraries, as discussed in Section 2. The subset-based analysis is similar to the one described by Pottier [15], as it keeps separate constraint variables for upper-bounds and lower-bounds. After the analysis, the types for the upper-bound and lower-bound constraint variables are merged to form a single resulting type for each expression.

Compared to TSCHECK, some constraints have been added to improve precision for parameter types, for example, so that the arguments to operators such

as - and * are treated as numbers. (Due to the page limit, we omit the actual analysis constraints used by TSINFER.)

A subset-based analysis gives more precise dataflow information compared to a unification-based analysis, however, more precise dataflow information does not necessarily result in more precise type inference. For example, consider the expression `foo = bar || ""`, where `bar` is a parameter to a function that is never called within the library. A unification-based analysis, such as TSCHECK, will unify the types of `foo`, `bar` and `""`, and thereby conclude that the type of `bar` is possibly a string. A more precise subset-based analysis will only constrain the possible types of `foo` to be a superset of the types of `bar` and `""`, and thereby conclude that the type of `bar` is unconstrained. In a subset-based analysis with both upper-bound and lower-bound constraint variables, the example becomes more complicated, but the result remains the same. This shows that changing from unification-based to subset-based analysis does not necessarily improve the precision of the type inference. We investigate this experimentally in Section 5.

3.3 The Emitting Phase

The last phase of TSINFER uses the results of the preceding phases to emit a declaration for the library. A declaration can be seen as a tree structure that resembles the heap snapshot, so we create the declaration by traversing the heap snapshot and converting the JavaScript values to TypeScript types, using the results from the static analysis when a function is encountered.

Implementing this phase is conceptually straightforward, although it does involve some technical complications, for example, handling cycles in the heap snapshot and how to combine a set of recursive types into a single type.

4 TSEVOLVE: Evolution of Type Declarations

The goal of TSEVOLVE is to create a list of changes between an old and a new version of a JavaScript library. To do this it has access to three input files: the JavaScript files for the old version `old.js` and the new version `new.js` and an existing TypeScript declaration file for the old version `old.d.ts`.

To find the needed changes for the declaration file, a naive first approach would be to compare `old.d.ts` with the output of running TSINFER on `new.js`. However, this will result in a lot of spurious warnings, both due to imprecisions in the analysis of `new.js`, but also because of intentional discrepancies in `old.d.ts`, as discussed in Section 2.

Instead we choose a less obvious approach, where TSEVOLVE uses TSINFER to generate declarations for both `old.js` and `new.js`. These declarations are then traversed as trees, and any location where the two disagree is marked as a change. The output of this process will still contain spurious changes, but unchanged features in the implementation should rarely appear as changes, as imprecisions in unchanged features are likely the same in both versions. We then use `old.d.ts` to filter out the changes that concern features that are not declared in `old.d.ts`, which removes many of the remaining spurious changes.

Relevant function sources code from `old.js` and `new.js` are also printed as part of the output, which allows for easy manual identification of many of the remaining spurious changes. As the analysis does not have perfect precision, it is necessary to manually inspect and potentially adjust the suggested changes before modifying the declaration file.

As an extra feature, in case a partially updated declaration file for the new version is available, TSEVOLVE can use that file to filter out some of the changes that have already been made.

5 Experimental Evaluation

Our implementations of TSINFER and TSEVOLVE, which together contain around 20 000 lines of Java code and 1 000 lines of JavaScript code, are available at `http://www.brics.dk/tstools/`.

We evaluate the tools using the following research questions.

- RQ1: Does the subset-based approach used by TSINFER improve analysis speed and precision compared to the unification-based alternative?
- RQ2: A tool such as TSCHECK that only aims to check existing declarations may blindly assume that some parts of the declarations are correct, whereas a tool such as TSINFER must aim to infer complete declarations. For this reason, it is relevant to ask: How much information in declarations is blindly ignored by TSCHECK but potentially inferred by TSINFER?
- RQ3: Can TSINFER infer useful declarations for libraries? That is, how accurate is the structure of the declarations and the quality of the types compared to handwritten declarations?
- RQ4: Is TSEVOLVE useful in the process of co-evolving declaration files as the underlying libraries evolve? In particular, does the tool make it possible to correctly update a declaration file in a short amount of time?

We answer these questions by running the tools on randomly selected JavaScript libraries, all of which have more than 5 000 stars on GitHub and a TypeScript declaration file of at least 100 LOC. Our tools do not yet support the `require` function from Node.js,[7] so we exclude Node.js libraries from this evaluation. All experiments have been executed on a Windows 10 laptop with 16GB of RAM and an Intel i7-4712MQ processor running at 1.5GHz.

RQ1 (subset-based vs. unification-based static analysis)

To compare the subset-based and unification-based approaches, we ran TSINFER on 20 libraries. The results can be found in the left half of Table 1. The *Funcs* column shows the number of functions analyzed for each library. The *Unification* and *Subset* columns show the analysis time for the unification-based and subset-based analysis, respectively, using a timeout of 30 minutes.

[7] `https://nodejs.org/`

Table 1. Analysis speed and precision.

	Speed			*Precision*			
Library	*Funcs*	*Unification*	*Subset*	*Unification*	*Subset*	*Equal*	*Unclear*
Ace	1 249	`timeout`	13.8s	-	-	-	-
AngularJS	609	193.3s	7.8s	1	14	17	0
async	169	28.2s	4.9s	2	22	20	6
Backbone.js	176	28.7s	4.8s	1	9	44	0
D3.js	1 030	181.7s	15.8s	4	19	44	2
Ember.js	2 902	`timeout`	319.7s	-	-	-	-
Fabric.js	1 032	`timeout`	15.7s	-	-	-	-
Hammer.js	122	32.5s	3.2s	0	2	61	3
Handlebars.js	280	9.2s	6.9s	0	3	12	1
Jasmine	51	135.4s	4.6s	2	4	71	0
jQuery	500	`timeout`	41.2s	-	-	-	-
Knockout	325	168.8s	14.4s	2	7	41	8
Leaflet	758	`timeout`	11.6s	-	-	-	-
Moment.js	446	872.6s	12.4s	1	27	21	2
PixiJS	1 527	`timeout`	308.0s	-	-	-	-
Polymer.js	748	424.2s	8.5s	1	10	41	3
React	1 261	`timeout`	14.0s	-	-	-	-
three.js	1 243	`timeout`	208.8s	-	-	-	-
Underscore.js	298	81.2s	4.2s	0	4	47	0
vue.js	433	`timeout`	6.2s	-	-	-	-
Total	15 159	-	1 026.5s	14	121	419	25

The results show that our subset-based analysis is significantly faster than the unification-based approach. This is perhaps counterintuitive for readers familiar with Andersen-style [1] (subset-based) and Steengaard-style [20] (unification-based) pointer analysis for e.g. C or Java. However, it has been observed before for JavaScript, where the call graph is usually inferred as part of the analysis, that increased precision often boosts performance [19,2].

We compared the precision of the two approaches by their ability to infer function signatures on the libraries where the unification-based approach does not reach a timeout. Determining which of two machine generated function signatures is the most precise is difficult to do objectively, so we randomly sampled some of the function signatures and manually determined their precision. To minimize bias, each pair of generated function signatures was shown randomly.

The results from these tests are shown in the right half of Table 1 where the function signatures have been grouped into four categories: *Unification* (the unification-based analysis inferred the most precise signature), *Subset* (the subset-based analysis was the most precise), *Equal* (the two approaches were equally precise), and *Unclear* (no clear winner). The results show that the subset-based approach in general infers better types than the unification-based approach. The unification-based did in some cases infer the best type, which is due to the fact that a more precise analysis does not necessarily result in a more precise type inference, as explained in Section 3.2.

RQ2 (information ignored by TSCHECK but considered by TSINFER)

TSCHECK only checks the return types of the functions where the corresponding signature in the declaration file do not have a `void`/`any` return type, which may

Table 2. Features in handwritten declaration files ignored by TSCHECK but taken into account by TSINFER.

Library	`void`/`any` *functions*	*(all)*	*Parameters*	*Classes*	*Fields*
Ace	301	(460)	370	2	4
AngularJS	8	(26)	39	0	0
async	64	(80)	222	0	0
Backbone.js	67	(149)	210	7	31
D3.js	7	(219)	271	5	12
Ember.js	270	(629)	991	58	103
Fabric.js	93	(330)	382	25	17
Hammer.js	33	(53)	53	16	24
Handlebars.js	20	(20)	19	1	0
Jasmine	1	(1)	1	1	0
jQuery	19	(53)	88	1	0
Knockout	68	(125)	226	6	0
Leaflet	48	(325)	435	26	17
Moment.js	0	(70)	71	0	0
PixiJS	338	(522)	639	86	584
Polymer.js	3	(4)	3	0	0
React	3	(21)	30	1	4
three.js	328	(993)	1295	180	632
Underscore.js	36	(121)	241	0	0
vue.js	7	(23)	42	1	8
Total	1714	(4224)	5628	416	1436

detect many errors, but the rest of the declaration file is blindly assumed to be correct. In contrast, TSINFER infers types for all functions, including their parameters, and it also infers classes and fields.

Table 2 gives an indication of the amount of extra information that TSINFER can reason about compared to TSCHECK. For each library, we show the number of functions that have return type `void` or `any` (and in parentheses the total number of functions), and the number of parameters, classes, and fields, respectively. The numbers are based on the existing handwritten declaration files.

We see that on the 20 benchmarks, TSCHECK ignores 1 714 of the 4 224 functions, silently assumes 5 628 parameter types to be correct, and ignores 1 436 instance fields spread over 416 classes. In contrast TSINFER, and thereby also TSEVOLVE, does consider all these kinds of information.

RQ3 (usefulness of TSINFER)

As mentioned in Section 2, TSCHECK is effective for checking declarations, but not for inferring them. We are not aware of any other existing tool that could be considered as an alternative to TSINFER. To evaluate the usefulness of TSINFER, we therefore evaluate against existing handwritten declaration files, knowing that these contain imprecise information.

We first investigate the ability of TSINFER to identify classes, modules, instance fields, methods, and module functions (but without considering inheritance relationships between the classes and types of the fields, methods, and functions). These features form a hierarchy in a declaration file. For example,

Table 3. Precision of inferring various features of a declaration file.

	Classes			*Modules*			*Class fields*			*Class methods*			*Module functions*		
Library	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*
Ace	0	2	0	1	0	1	0	0	0	0	0	0	3	2	0
AngularJS	0	0	0	2	1	0	0	0	0	0	0	0	22	2	4
async	0	0	0	1	1	0	0	0	0	0	0	0	88	6	0
Backbone.js	5	0	2	1	1	0	18	3	12	183	8	3	12	10	2
D3.js	5	13	0	1	9	9	12	4	0	15	4	2	56	247	12
Ember.js	62	64	54	16	32	7	8	187	35	40	54	74	333	678	112
Fabric.js	25	21	0	7	3	1	16	193	1	248	402	8	165	24	3
Hammer.js	8	8	7	2	0	1	7	64	0	39	6	0	16	9	9
Handlebars.js	2	4	0	4	3	2	0	3	0	20	4	0	28	8	3
Jasmine	2	22	0	1	4	0	0	0	0	0	8	0	28	33	3
jQuery	2	6	0	4	29	2	0	6	0	0	6	0	90	59	6
Knockout	5	3	1	14	11	1	0	4	0	14	3	0	91	63	2
Leaflet	33	10	0	22	21	1	5	75	12	241	248	2	137	135	1
Moment.js	0	2	0	1	0	0	0	0	0	0	0	0	89	25	6
PixiJS	70	2	16	31	8	2	812	46	52	450	37	7	128	14	16
Polymer.js	0	2	0	1	19	0	0	0	0	0	0	0	2	9	0
React	1	0	0	4	3	0	3	7	1	2	0	1	26	6	130
three.js	169	12	11	12	18	0	2 348	71	33	907	105	24	241	26	8
Underscore.js	0	1	0	1	0	0	0	0	0	0	0	0	117	1	3
vue.js	1	1	0	2	4	0	8	22	0	23	21	0	12	1	1
Total	390	173	91	128	167	27	3 237	685	146	2 182	906	121	1684	1358	321
Precision/Recall	*Prec*: 69.3% *Rec*: 80.9%			*Prec*: 43.4% *Rec*: 82.6%			*Prec*: 82.5% *Rec*: 95.7%			*Prec*: 70.7% *Rec*: 94.8%			*Prec*: 55.36% *Rec*: 84.0%		

`PIXI.Matrix.invert` identifies the `invert` method in the `Matrix` class in the `PIXI` module of *PixiJS*. When comparing the inferred features with the ones in the handwritten declaration files, a true positive (*TP*) is one that appears in both, a false positive (*FP*) exists only in the inferred declaration, and a false negative (*FN*) exists only in the handwritten declaration. In case of *FP* or *FN* we exclude the sub-features from the counts. The quality of the types of the fields and methods is investigated later in this section; for now we only consider their existence.

The counts are shown in Table 3, together with the resulting precision (*Prec*) and recall (*Rec*). We see that TSINFER successfully infers most of the structure of the declaration files, although some manual post-processing is evidently necessary. For example, 80.9% of the classes and 95.7% of the fields are found by TSINFER. Having false positives in an inferred declaration (i.e., low precision) is less problematic than false negatives (i.e., low recall): it is usually easier to manually filter away extra unneeded information than adding information that is missing in the automatically generated declarations.

The identification of classes, modules, methods, and module functions in TSINFER is based entirely on the snapshots (Section 3.1), so one might expect 100% precision for those counts. (Identification of fields is partly also based on the static analysis.) The main reason for the non-optimal precision is that many features are undocumented in the manually written declarations. By manually inspecting these cases, we find that most of these are likely intentional: although

Table 4. Measuring the quality of inferred types of fields and methods.

	Class fields				*Class methods and module functions*				
Library	*Perfect*	*Good*	*Any*	*Bad*	*Perfect*	*Good*	*Any*	*Bad*	*No params*
Ace	0	0	0	0	0	3	0	0	0
AngularJS	0	0	0	0	10	10	2	0	0
async	0	0	0	0	0	26	18	0	6
Backbone.js	14	2	2	0	12	6	30	0	7
D3.js	3	0	9	0	11	36	5	2	1
Ember.js	3	3	2	0	42	37	11	5	5
Fabric.js	13	0	3	0	22	18	10	3	22
Hammer.js	0	0	1	0	7	17	9	0	8
Handlebars.js	0	0	0	0	6	22	9	2	7
Jasmine	0	0	0	0	1	12	6	0	9
jQuery	0	0	0	0	5	21	20	1	0
Knockout	0	0	0	0	5	25	24	0	1
Leaflet	3	2	0	0	14	36	7	0	19
Moment.js	0	0	0	0	8	15	21	0	6
PixiJS	32	5	13	0	38	40	21	1	0
Polymer	0	0	0	0	1	1	2	0	0
React	2	0	1	0	0	32	5	0	0
three.js	37	3	10	0	44	46	10	0	0
Underscore.js	0	0	0	0	0	11	35	3	1
vue.js	2	0	6	0	6	15	2	1	0
Total	109	15	47	0	232	429	247	18	92

they are technically exposed to the applications, the features are meant for internal use in the libraries and not for use by applications. Non-optimal recall is often caused by intentional discrepancies as discussed in Section 2 or by libraries that violate our assumption explained in Section 3.1 about the API being fully established after the initialization code has finished. Other reasons for non-optimal precision or recall are simply that the handwritten declaration files contain errors or, in cases where the version number is not clearly stated in declaration file, we were unable to correctly determine which library version it is supposed to match.

To measure the quality of the inferred types of fields and methods, we again used the handwritten declaration files as gold standard and this time manually compared the types, in places where the inferred and handwritten declaration files agreed about the existence of a field or method. Such a comparison requires some manual work, so we settled for sampling: for each library, we compared 50 fields and 100 methods (thereof 50 that were classified as constructors), or fewer if not that many were found in the library.

The result of this comparison can be seen in Table 4 where *Perfect* means that the inferred and handwritten type are identical, *Good* means that the inferred type is better than having nothing, *Any* means that the main reason for the sample not being perfect is that either the inferred or the handwritten type is `any`, *Bad* means that the inferred type is far from correct, and *No params* means that the inferred type has no parameters while the handwritten does. Obviously, this categorization to some extent relies on human judgement, but we believe it nevertheless gives an indication of the quality of the inferred types.

Table 5. Classification of TSEVOLVE output.

Library	*TP*	*FP*	*FP**	*Unclear*
async 1.4 → 2.0	38	0	52	2
Backbone.js 1.0 → 1.3	34	0	42	2
Ember.js 1.13 → 2.0	55	24	40	0
Ember.js 2.0 → 2.7	44	0	54	0
Handlebars.js 3 → 4	37	3	8	59
Moment.js 2.11 → 2.14	10	0	54	2
PixiJS 3 → 4	270	13	41	2
Total	488	40	291	67

An example in the *Good* category is in *PixiJS* where TSINFER infers a perfect type for the `PIXI.Matrix().applyInverse` method, except for the first argument where it infers the type `{x: number, y: number}` instead of the correct `PIXI.Point`.

As can be seen in Table 4, the types inferred for fields are perfect in most cases, and none of them are categorized as *Bad*. The story is more mixed for method types. Here, there are relatively fewer perfect types, but function signatures are also much more complex, given that they often contain multiple parameters as well as a return type, and parameters can sometimes be extremely difficult to infer correctly. For many method types categorized as *Good*, the overall structure of the inferred type is correct but some spurious types appear in type unions for some of the parameters or the return type, or, as in the example with `applyInverse`, an object type is inferred whose properties is a subset of the properties in the handwritten type. The main reason that some method types are categorized as *No params* is that our analysis is unable to reason precisely about the built-in function `Function.prototype.apply` and the `arguments` object. We leave it as future work to explore more precise abstractions of these features.

RQ4 (usefulness of TSEVOLVE)

To evaluate if TSEVOLVE can assist in evolving declaration files, we performed a case study where TSEVOLVE was used for updating declaration files in 7 different evolution scenarios. In each case, we used the output from TSEVOLVE to make a pull request to the relevant repository. All of these libraries have more than 10 000 stars on GitHub and had a need for the declaration file to be updated, but were otherwise randomly selected. We had no prior experience in using any of the libraries.

The output from TSEVOLVE is a list of changes for each declaration file. We took the output lists from each of the 7 updates and classified each entry in each list based upon how useful it was in the process of evolving the specific library.

The result of this can be seen in Table 5 where each change listed by TSEVOLVE is counted in one of the four columns. *TP* counts true positives, i.e. changes that reflect an actual change in the library that should be reflected in the declaration file. Both *FP* and *FP** count false positives, the difference being that changes counted in *FP** could easily be identified as spurious by looking at the output

Table 6. Pull requests sent based in TSEVOLVE output.[8]

Library	*Lines added*	*Lines removed*	*Library author response*
async 1.4 → 2.0	46	13	"pretty thorough and seems to follow the 2.x API much better than what we currently have"
Backbone.js 1.0 → 1.3	27	3	
Ember.js 1.13→2.0	8	508	"LGTM[9]👍"
Ember.js 2.0→2.7	96	92	"👍"
Handlebars.js 3 → 4	49	2	
Moment.js 2.11 → 2.14	4	0	"thank you, looks good"
PixiJS 3 → 4 (pre-release)	158	261	"Awesome PR"
PixiJS 3 → 4	19	4	"I went through all of your changes and can confirm everything is perfect"

from TSEVOLVE, as explained in Section 4. *Unclear* counts the listed changes that could not be easily categorized.

In the update from *Ember.js* version 1.13 to version 2.0, all of the 24 in the *Bad* category are due to *Ember.js* breaking our assumption about the API being fully established after the top-level code has executed. None of the other libraries violate that assumption.

In the update of *Handlebars.js* from version 3 to 4, all the 59 in the *Unclear* category are due to the structures of the handwritten and the inferred declaration files being substantially different. TSEVOLVE is therefore not able to automatically filter out undocumented features, and all 59 entries are therefore filtered out manually.

From Table 5 we can see that the output from TSEVOLVE mostly points out changes that should be reflected in the corresponding declaration file. Among the spuriously reported changes, most of them can easily be identified as being spurious and are therefore not a big problem.

These outputs of TSEVOLVE were used to create pull requests, which are described in Table 6. For each pull request, we show how many lines the pull request added and removed in the declaration file,[10] along with a response from a library developer, if one was given. For *Handlebars.js*, the pull request additionally contains a few corrections of errors in the declaration file that were spotted while

[8] The pull requests: `https://gist.github.com/webbiesdk/f82c135fc5f67b0c7f175e985dd0c889`

[9] An acronym for "Looks Good To Me".

[10] The complete pull requests in some cases contain more lines changed, due to minor refactorings or copying and renaming of files to match the version numbers.

reviewing the report from TSINFER. All 7 pull requests were accepted without any modifications to the changes derived from the TSEVOLVE output.

The total working time spent going from TSEVOLVE output to finished pull requests was approximately one day, despite having no prior experience using any of the libraries. Without tool support, creating such pull requests, involving a total of 407 lines added and 883 lines removed, for libraries that contain a total of 129 365 lines of JavaScript code across versions and declaration files containing 3 938 lines (after the updates), clearly could not have been done in the same amount of time.

6 Related Work

The new tools TSINFER and TSEVOLVE build on the previous work on TSCHECK [8], as explained in detail in the preceding sections. Other research on TypeScript includes formalization and variations of its type system [4,17,18,22], and several alternative techniques for JavaScript type inference exist [16,11,6], however, none of that work addresses the challenges that arise when integrating JavaScript libraries into typed application code.

The need for co-evolving declaration files as the underlying libraries evolve can be viewed as a variant of collateral evolution [14]. By using our tools to increase confidence that the declaration files are consistent with the libraries, the TypeScript type checker becomes more helpful when developers upgrade applications to use new versions of libraries.

Our approach to analyze the JavaScript libraries differs from most existing dataflow and type analysis tools for JavaScript, such as, TAJS [9,2] and SAFE [3], which are whole-program analyzers and not sufficiently scalable and precise for typical JavaScript library code. We circumvent those limitations by concretely executing the library initialization code and using a subset-based analysis that is inspired by Pottier [15], Rastogi et al. [17], and Chandra et al. [6].

Other languages, such as typed dialects of Python [23,10], Scheme [21], Clojure [5], Ruby [12], and Flow for JavaScript[7], have similar challenges with types and cross-language library interoperability, though not (yet) at the same scale as TypeScript. Although TSINFER and TSEVOLVE are designed specifically for TypeScript, we believe our solutions may be more broadly applicable.

7 Conclusion

We have presented the tools TSINFER and TSEVOLVE and demonstrated how they can help programmers create and maintain TypeScript declaration files. By making the tools publicly available, we hope that the general quality of declaration files will improve, and that further use of the tools will provide opportunities for fine-tuning the analyses towards the intentional discrepancies found in real-world declarations.

Acknowledgments This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

References

1. Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
2. Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.
3. SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. $\text{SAFE}_{\text{WAPI}}$: web API misuse detector for web applications. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.
4. Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. 28th European Conference on Object-Oriented Programming*, 2014.
5. Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *Programming Languages and Systems - 25th European Symposium on Programming*, volume 9632 of *Lecture Notes in Computer Science*. Springer, 2016.
6. Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.
7. Facebook. Flow, 2016. `http://flowtype.org/`.
8. Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.
9. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, 2009.
10. Jukka Lehtosalo et al. Mypy, 2016. `http://www.mypy-lang.org/`.
11. Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *Proc. 9th Symposium on Dynamic Languages*, 2013.
12. Yukihiro 'Matz' Matsumoto. RubyConf 2014 – opening keynote, 2014. `http://confreaks.tv/videos/rubyconf2014-opening-keynote`.
13. Microsoft. TypeScript language specification, February 2015. `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`.
14. Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. EuroSys Conference*. ACM, 2008.
15. François Pottier. A framework for type inference with subtyping. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, 1998.
16. Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.
17. Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proc. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.
18. Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. In *Proc. 29th European Conference on Object-Oriented Programming*, 2015.
19. Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*. Springer, 2012.

20. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.
21. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. 2008.
22. Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for TypeScript. In *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.
23. Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Proc. 10th ACM Symposium on Dynamic Languages*, 2014.