Open access • Proceedings Article • DOI:10.1145/2660193.2660215

Checking correctness of TypeScript interfaces for JavaScript libraries — Source link

Asger Feldthaus, Anders Møller

Institutions: Aarhus University

Published on: 15 Oct 2014 - Conference on Object-Oriented Programming Systems, Languages, and Applications

Topics: TypeScript, JavaScript library, JavaScript, Declaration and Interface (Java)

Checking Correctness of TypeScript Interfaces for JavaScript Libraries

Asger Feldthaus
Aarhus University
asf@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

Abstract

The TypeScript programming language adds optional types to JavaScript, with support for interaction with existing JavaScript libraries via interface declarations. Such declarations have been written for hundreds of libraries, but they can be difficult to write and often contain errors, which may affect the type checking and misguide code completion for the application code in IDEs.

We present a pragmatic approach to check correctness of TypeScript declaration files with respect to JavaScript library implementations. The key idea in our algorithm is that many declaration errors can be detected by an analysis of the library initialization state combined with a light-weight static analysis of the library function code.

Our experimental results demonstrate the effectiveness of the approach: it has found 142 errors in the declaration files of 10 libraries, with an analysis time of a few minutes per library and with a low number of false positives. Our analysis of how programmers use library interface declarations furthermore reveals some practical limitations of the TypeScript type system.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Verification

1. Introduction

The TypeScript [10] programming language is a strict superset of JavaScript, one of the main additions being *optional types*. The raison d'être of optional types is to harness the flexibility of a dynamically typed language, while providing some of the benefits otherwise reserved for statically typed languages: for instance, the compiler can quickly catch obvious coding blunders, and editors can provide code completion. Types also carry documentation value, giving programmers a streamlined language with which to document APIs. TypeScript does not ensure *type safety*, in other words, the type system is unsound by design. Even for programs that pass static type checking, it is possible that a variable at runtime has a value that does not match the type annotation. This is seen as a trade-off necessary for keeping the type system unrestrictive.

TypeScript is designed to compile to JavaScript using a shallow translation process. Each TypeScript expression translates into one JavaScript expression, and a value in TypeScript is the same as a value in JavaScript. This makes it possible to mix TypeScript code with JavaScript code, in particular existing JavaScript libraries, without the use of a foreign function interface as known from other programming languages. However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript. Once such a description has been written for a JavaScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository,[1] which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may reject perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes no attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it

OOPSLA '14, October 19–21, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660215

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

Fragment of `d3.js` (library implementation)

```
 1 var d3 = function() {
 2   var d3 = {...};
 3   ...
 4   d3.scale = {};
 5   ...
 6   d3.scale.threshold = function() {
 7       return d3_scale_threshold([.5], [0, 1]);

 8   };
 9   function d3_scale_threshold(domain,range) {
10       ...
11   }
12   return d3;
13 }();
```

Fragment of `d3.d.ts` (declaration file)

```
14 declare module D3 { ...
15     module Scale { ...
16         interface ScaleBase { ...
17             theshold(): ThresholdScale;
18         }
19         interface ThresholdScale { ... }
20     }
21     interface Base { ...
22         scale : Scale.ScaleBase;
23     }
24 }
25 declare var d3 : D3.Base;
```

Figure 1. Fragments of the `d3.js` library and the associated declaration file found on the Definitely Typed repository. The ellipses ("...") represent fragments taken out for brevity.

can easily remain undetected long enough for the buggy declaration file to get published to the community. Hence, application programmers who depend on the declaration file may occasionally find themselves debugging their own code, only to find that the bug was in the declaration file.

Figure 1 shows an example of a bug found in the declaration file for the `d3.js` library. The initialization code for the library creates the `d3` and `d3.scale` objects, and exposes the `threshold` function on `d3.scale`. The library uses the `d3` object as a namespace for all its exposed features. Note that the function expression spanning lines 1–13 is invoked immediately on line 13, exposing the `d3` object through a global variable. This trick is used to hide the `d3_scale_threshold` function (and other features not shown in the fragment).

The declaration file declares a module `D3` to provide a namespace for type declarations, and declares that the global variable `d3` must satisfy the type `D3.Base`, which refers to the interface on line 21. The interface `D3.Base` declares a property `scale` of type `Scale.ScaleBase`, which refers to `D3.Scale.ScaleBase` defined on line 16. It follows that the `d3.scale` object created during library initialization must satisfy the `D3.Scale.ScaleBase` interface. This interface declares a `theshold` method, which is intended to document the function created on line 6, but the declaration file contains a misspelling of the word "threshold". This has consequences for users of the declaration file: First, editors that provide code completion (also called autocompletion) will suggest that an expression like `d3.scale.th` is completed to `d3.scale.theshold()`, which will pass the static type check, but fail at runtime. Second, if the programmer discovers the problem and fixes the typo in his client code, the static type checker will reject his otherwise correct code, claiming that `ScaleBase` has no method called `threshold`.

Some declaration files, including the one for `d3.js`, consist of over a thousand effective lines of code and are not trivial to write or proofread. Currently, the main line of defense against declaration bugs is to write tests using a code completing editor, deliberately using the completion suggestions. These tests invoke certain features of the library, and if the tests both type check and do not fail at runtime, the declaration file is likely to be correct for those features. An obvious downside to this approach is that the programmer effectively has to duplicate parts of the library's test suite (which is not written in TypeScript, and hence cannot be checked by the compiler). Such tests are present for many libraries in the Definitely Typed repository, but they are incomplete and many bugs have escaped the scrutiny of these tests. For instance, there are 1,800 lines of code for testing the declaration file for `d3.js`, yet the aforementioned bug was not caught.

The API of a JavaScript library is not expressed declaratively as in other languages but defined operationally by its initialization code that sets up the objects and functions that constitute the library interface. This initialization is typically done during execution of the top-level code of the library, that is, immediately when the JavaScript file is loaded and not later when events are processed. Based on this observation, we expect that many mismatches between TypeScript declarations and JavaScript implementations can be detected by considering the effects of the library initialization code.

In this work we present a technique for checking the correctness of TypeScript declaration files against a JavaScript implementation. Our technique proceeds in three phases:

1. We first execute the library's initialization code and extract a snapshot of its state.
2. Next we check that the global object and all reachable objects in the snapshot match the types from the declaration file using a structural type checking algorithm.
3. We then complement the preceding phase with a lightweight static analysis of each library function to check that it matches its signature in the declaration file.

The error in Figure 1 is one of those found by type checking the heap snapshot (phase 2). We shall later see examples of errors that are caught by the static analysis of the function code (phase 3).

With our implementation, TSCHECK, we observe that many large declaration files published to the community contain bugs, many of which cause other tools to misbehave. We also observe that a large portion of the bugs are easily fixed, while others are harder to address due to limitations in TypeScript's type system.

In summary, our contributions are as follows:

- We point out the need for tool support to check correctness of TypeScript declaration files against JavaScript library implementations, and we present an algorithm that is designed to detect many kinds of errors in such declarations. Our algorithm checks that the runtime object structure after library initialization and the library function code are consistent with the given declaration file.
- As part of describing the algorithm, we formalize the relationship between values and types in TypeScript, which is otherwise only described informally in the language specification.
- Our experimental results show that the algorithm is capable of detecting 142 errors in a collection of 10 existing declaration files, with a low number of false positives. The snapshot type checking phase takes less than a second to execute for each library, and the static analysis checks each function in about one second.
- The development of our algorithm and experiments has exposed some practical limitations of TypeScript's type system, which may be useful information for evolution of the TypeScript language.

Our algorithms naturally depend on what it means for a runtime value to have a certain type according to the TypeScript language specification. The snapshot type checking phase described in Section 4 is designed to be sound and complete with respect to this meaning of types, modulo two assumptions we explain in Section 3 about library initialization and two deviations that we introduce for reasons explained in Section 4.1.

As mentioned, TypeScript's static type checker is deliberately unsound, as exemplified by the following program:

```
26 var x : string = "foo";
27 var y : any = x;
28 var z : number = y;
```

The assignment on line 28 would require a type cast in a Java-like type system (which would consequently fail at runtime), but the assignment is deliberately allowed in TypeScript. Our algorithm for statically analyzing the function code is independent of TypeScript's type checking procedure. Still, we follow the same pragmatic approach and intentionally sacrifice strong type safety in the design of our static analysis algorithm in Section 5 to ensure high performance and reduce the number of false positives.

2. The TypeScript Declaration Language

TypeScript extends the JavaScript language with additional statements and expressions for interfacing with the type system. We here give an overview of the declaration language in TypeScript and introduce the notation and terminology that we use in the remainder of the paper.

Declarations in TypeScript can do one or both of the following: (1) contribute to the type namespace at compile time, and (2) have an effect at runtime. Declarations that have no effect at runtime are called *ambient declarations*. Declaration files consist exclusively of ambient declarations, so we will focus on the subset of TypeScript that may occur in such declarations.

An *interface declaration* declares a type and has no effect at runtime. For example, the interface below declares a type `IPoint` with members `x` and `y` of type `number`.

```
interface IPoint {
   x: number;
   y: number;
}
```

TypeScript uses a structural type system; any object satisfies the `IPoint` type if it has properties `x` and `y` with numeric values. This stands in contrast to nominal type systems known from Java-like languages where an object only implements an interface if the class of the object is explicitly declared to implement the interface.

A *class declaration* declares a type and creates a constructor for the class at runtime. For example, consider the class below:

```
class Point {
   x: number;
   y: number;
   constructor(x: number, y: number) {
      this.x = x;
      this.y = y;
   }
}
```

This declares a type `Point` but will also at runtime create a constructor function for the class and expose it in the variable `Point`, so that the class can be instantiated by an expression, such as, `new Point(4,5)`. Class types are only satisfied by objects that are created using its constructor, so for instance, an object created by the expression `{x:4, y:5}` satisfies the `IPoint` type but not the `Point` type, while an object created by the expression `new Point(4,5)` satisfies both `Point` and `IPoint`. This makes class types behave like the nominal types from Java-like languages. In TypeScript nomenclature, such types are called *branded types*.

The class declaration above has effects at runtime, and thus cannot occur in a declaration file. An *ambient class declaration* may be used instead, which instructs the TypeScript compiler to behave as if the class declaration had been there (for type checking purposes), but no code will be emitted for the declaration:

```
declare class Point {
    x: number;
    y: number;
    constructor(x: number, y: number);
}
```

2.1 TypeScript Declaration Core

The TypeScript language has non-trivial name resolution and inheritance features, which we will not describe in this work. We instead present TypeScript Declaration Core that captures the expressiveness of TypeScript's declaration language, after all type references have been resolved and inheritance hierarchies have been flattened.

We now describe the syntax and semantics of this core type system. We let S denote the set of Unicode strings for use as property names and string constants. A TypeScript declaration core consists of the following finite disjoint sets,

$$\begin{array}{rcl} T \in N & : & \text{type names} \\ v \in V & : & \text{type variables} \\ e \in E & : & \text{enum type names} \end{array}$$

along with a type environment, the name of the global object type, and a mapping from enum types to their members:

$$\begin{array}{cccl} \Sigma & : & N \rightarrow \textit{type-def} & \text{type environment} \\ G & \in & N & \text{name of the global object type} \\ \Gamma & : & E \rightarrow \mathcal{P}(\overline{S}) & \text{enum members} \end{array}$$

The type environment Σ maps type names to their definitions, e.g. `IPoint` would map to the type `{x:number; y:number}`. In JavaScript, all global variables are actually properties of a special object known as *the global object*. We use G as the name of the global object type. The global object must satisfy the global object type, hence $\Sigma(G)$ can be used to describe the types of global variables. Enum type names $e \in E$ describe finite enumeration types; the enum member mapping Γ denotes what values belong to each enum type. Enum types will be discussed in more detail later in this section.

The syntax of types is given below:

$$\begin{array}{rcl} \textit{type-def} & ::= & \langle\ \overline{V}\ \rangle\ \textit{obj} \\ \tau \in \textit{type} & ::= & \mathsf{any} \mid \mathsf{void} \mid \mathsf{bool} \mid \mathsf{num} \mid \mathsf{str} \mid \text{"}S\text{"} \mid \\ & & \textit{obj} \mid N\langle\overline{\textit{type}}\rangle \mid E \mid V \\ o \in \textit{obj} & ::= & \{\ \overline{\textit{prty}}\ ;\ \overline{\textit{indexer}}\ ;\ \overline{\textit{fun-sig}}\ ;\ \overline{\textit{brand}}\ \} \\ \rho \in \textit{prty} & ::= & \mathsf{opt}?\ S : \textit{type} \\ i \in \textit{indexer} & ::= & [\mathsf{num} : \textit{type}] \mid [\mathsf{str} : \textit{type}] \\ c \in \textit{fun-sig} & ::= & \mathsf{new}?\ \mathsf{vargs}?\ \langle\ \overline{\textit{type-parm}}\ \rangle(\ \overline{\textit{parm}}\ ) \Rightarrow \textit{type} \\ p \in \textit{parm} & ::= & \mathsf{opt}?\ \textit{type} \\ \textit{type-parm} & ::= & V\ \mathsf{extends}\ \textit{type} \\ b \in \textit{brand} & ::= & \overline{S} \end{array}$$

We will now informally describe the meaning of these types.

Type Definitions A type definition $\langle\overline{v}\rangle o$ defines a potentially generic type, with $\overline{v}$ being the names of the type variables.

Example The `IPoint` interface would give rise to the type definition $\Sigma($`IPoint`$) = \langle\rangle$`{x:number; y:number}`.

Type References A type of form $T\langle\overline{\tau}\rangle$ denotes the instantiation of the type with name T with type arguments $\overline{\tau}$. Here, T must be the name of a type definition that has the same number of type parameters. If zero type parameters are specified, this is simply a reference to the type T.

Objects An object type consists of properties, indexers, function signatures, and brands. Each member restricts the set of values that satisfy the type. These meaning of these members are detailed below.

Properties A property $f : \tau$ denotes that an object of this type must have a property f with a value of type τ. If the opt flag is specified, the property is optional. An optional property opt $f : \tau$ denotes that *if* the object has a property named f, then the value of that property must have type τ.

Indexers An indexer [str : τ] denotes that all enumerable properties of the object must have a value of type τ. Such indexers are used for describing the types of objects that are used as dictionaries. Enumerability is a boolean flag set on each property, indicating if it should be seen by reflective operators, such as JavaScript's for-in loop. Properties are generally enumerable by default, but certain natively defined properties are non-enumerable so that they do not interfere with dictionary-like objects.

An indexer [num : τ] denotes that all properties whose name is a valid array index (i.e. "0", "1", "2", etc.) must satisfy τ. JavaScript treats array indices as properties whose names are number strings, so number indexers can be used to describe the types of arrays and array-like objects.

Example The declaration file `lib.d.ts`, which the TypeScript compiler uses as a model for JavaScript's native API, uses number indexers to describe the type of arrays (TypeScript uses a slightly different syntax for indexers than our core language):

```
interface Array<T> { ...
  [n: number]: T;
}
```

Indexers are not only used for actual arrays. The declaration file for jQuery shows that a `JQuery` object can be seen as an array of `HTMLElements`:

```
interface JQuery { ...
  [n: number]: HTMLElement;
}
```

Function Signatures A function signature in an object type denotes that the object must be callable as a function; such an object is typically called a *function*. In its simplest form, a signature $(\tau_1, ..., \tau_n) \Rightarrow \tau$ specifies that when the function is invoked with n arguments satisfying types $\tau_1, ..., \tau_n$, a value of type τ should be returned. If the new flag is present, the signature only applies to constructor calls, otherwise, it only applies to non-constructor calls. If the vargs flag is present, the signature additionally applies to any call with more than n arguments where the extra arguments satisfy the type τ_n. The vargs flag cannot be used if $n = 0$. If the last k parameters are annotated with the opt flag, the signature additionally applies to calls with $n-k$ arguments. It is invalid to use the opt flag for a parameter unless all following parameters also use it. A function signature is polymorphic if it declares one or more type parameters; in this case the function must satisfy all valid instantiations of the function signature. An instantiation is valid if the types substituting a type variable satisfies its bound.

Brands An object type with a brand b is called a *branded type*. Brands serve to emulate nominal types in TypeScript's structural type system, so that otherwise equivalent types are distinguishable. In TypeScript, classes are branded types while interfaces are unbranded. In our core type system, a brand is an access path pointing to the prototype object associated with a class. An access path is a sequence of property names, for example, the path `foo.bar` points to the value of the `bar` property of the object in the global variable `foo`. For an object to satisfy a brand b, this object must have the object pointed to by b in its prototype chain.

Given a class declaration `declare class C {}`, the branded type $\Sigma(\texttt{C}) = \langle\rangle\{$ `C.prototype` $\}$ would be generated. This interpretation of branded types is based on how constructor calls work in JavaScript: when an object o is created by an expression of form `new F(...)`, the internal prototype link of o is initialized to the object referred to by `F.prototype`. Thus, the prototype chain indicates what constructor was used to create a given object.

Enums An enum type of name e is satisfied by the values pointed to by the access paths in $\Gamma(e)$.

Example The enum declaration below declares an enum type of name `FontStyle` $\in E$.

```
declare enum FontStyle {
  Normal, Bold, Italic
}
```

The declaration also introduces a global variable `FontStyle` with members `Normal`, `Bold`, `Italic`, all of type `FontStyle`. The values for these three members are not specified in the type and can be chosen arbitrarily by the implementation. The specific values used by the implementation of this enum can be found by inspecting the access paths `FontStyle.Normal`, `FontStyle.Bold`, and `FontStyle.Italic`. For instance, here are two valid JavaScript implementations of the enum type:

```
var FontStyle = // implementation 1
  { Normal: 0, Bold: 1, Italic: 2 }
var FontStyle = // implementation 2
  { Normal: "normal", Bold: "bold",
    Italic: "italic" }
```

In both cases, we have $\Gamma(\texttt{FontStyle}) = \{$`FontStyle.Normal`, `FontStyle.Bold`, `FontStyle.Italic`$\}$, but the set of values satisfied by the enum type depends on the implementation being checked against.

3. Library Initialization

In this section we discuss what it means to initialize a library, the assumptions we make about the library initialization process, and the notion of a snapshot of the runtime state.

Loading a library in JavaScript amounts to executing the library's top-level code. In order to expose its functionality to clients, this code must introduce new global variables and/or add new properties to existing objects. Although not strongly enforced in any way, it is typically the case that a library is initialized and its API is ready to be used when the end of its top-level code is reached. In principle, it is possible for a library to postpone parts of its initialization until the client calls some initialization function or an event is fired from the browser, but this is seldom seen in practice. We exploit this common case using the following assumption:

Initialization assumption: The library's public API has been established at the end of its top-level code.

Another way to phrase the initialization assumption is that the declaration file is intended to describe the state of the heap after executing the top-level code. The experiments discussed in Section 6 confirm that this assumption only fails in rare cases.

JavaScript uses an event-based single-threaded programming model. This means that it is not possible for a JavaScript library to read any input at the top-level, neither through the UI nor using AJAX calls. Though it may initiate an AJAX call, it cannot react to the response until after the top-level code has finished and the event loop has resumed. Thereby we can execute the top-level code in isolation without needing to simulate inputs from the environment.

JavaScript has multiple execution platforms; notably, there are at least five major browsers as well as the server-side platform node.js.[2] These platforms provide incompatible native APIs, which means it is possible, and often necessary, for a library to detect what platform it is running on. Although the top-level code cannot read any external input, the platform itself can be seen as an input. We could take a snapshot using every version of every major platform, but this is somewhat impractical. Fortunately, libraries go to great lengths to provide platform-agnostic APIs. As long as the API is completely platform-agnostic, any declaration bug we can detect on one platform should be detectable on *all* supported platforms. In practice, it should therefore suf-

[2] `http://nodejs.org`

fice to run the library on just one platform. This leads to our second assumption:

Platform assumption: Executing the library on a single platform (as opposed to multiple platforms) is sufficient for the purpose of uncovering bugs in the declaration file.

Note that our goal is to find errors in the TypeScript declarations, not in the JavaScript implementation of the library. If we were looking for bugs in the library implementation, the platform assumption might not be as realistic. As with the previous assumption, experiments will be discussed in the evaluation section, which confirm that this assumption is valid in practice.

While some libraries are completely portable, there are also some that are specific to the browser platforms, and some are specific to the node.js platform. Our implementation supports two platforms: the phantom.js platform[3] based on WebKit (the engine used in Chrome and Safari) and the node.js platform. Most JavaScript libraries support at least one of these two platforms.

We now describe the structure extracted from the program state at runtime. The call stack is known to be empty when extracting the snapshot, because we are at the end of the top-level code, so this is effectively a snapshot of the heap.

We use boldface fonts for symbols related to the snapshot. A snapshot consists of a finite set of object names $\mathbf{O}$,

$$\mathbf{k} \in \mathbf{O} \quad : \quad \text{object names}$$

along with a name for the global object $\mathbf{g}$, and a store $\mathbf{s}$,

$$\begin{array}{lcll} \mathbf{g} & \in & \mathbf{O} & \text{(global object)} \\ \mathbf{s} & : & \mathbf{O} \to \textbf{\textit{obj-def}} & \text{(store)} \end{array}$$

where the definition of objects and values are as follows:

$$\begin{array}{rcl} \mathbf{x}, \mathbf{y} \in \textbf{\textit{value}} & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{num}(\mathbb{D}) \mid \mathbf{str}(S) \mid \\ & & \mathbf{null} \mid \mathbf{undef} \mid \mathbf{obj}(\mathbf{O}) \\ \textbf{\textit{obj-def}} & ::= & \{\ \mathsf{properties}\text{: } \overline{\textbf{\textit{prty}}}; \\ & & \quad \mathsf{prototype}\text{: } \textbf{\textit{value}}; \\ & & \quad \mathsf{function}\text{: } \textbf{\textit{fun}}; \\ & & \quad \mathsf{env}\text{: } \textbf{\textit{value}}\ \} \\ \textbf{\textit{fun}} & ::= & \mathbf{user}(U) \mid \mathbf{native}(S) \mid \mathbf{none} \\ \mathbf{p} \in \textbf{\textit{prty}} & ::= & \{\ \mathsf{name} : S \\ & & \quad \mathsf{enumerable} : \mathbf{true} \mid \mathbf{false} \\ & & \quad \mathsf{value} : \textbf{\textit{value}}; \\ & & \quad \mathsf{getter} : \textbf{\textit{value}}; \\ & & \quad \mathsf{setter} : \textbf{\textit{value}}\ \} \end{array}$$

Here, $\mathbb{D}$ is the set of 64-bit floating point values, and U is a set of identifiers for the functions in the source code from which the snapshot was taken.

An object has a list, properties, representing its ordinary properties. A property can have a value field, or it can have a getter/setter pair denoting functions that are invoked when the property is read from or written to, respectively. For properties with a non-**null** getter or setter, the value field is always **null**.

An object's prototype field represents JavaScript's internal prototype link; the object inherits the properties of its prototype object. This field is set to **null** for objects without a prototype.

The function field indicates whether the object is a user-defined function, a native function, or not a function. Variables captured in a closure are represented as properties of environment objects. For user-defined functions, the env field refers to the environment object that was active when the function was created, thus binding its free variables. For environment objects, the env field points to the enclosing environment object, or the global object. Environment objects cannot be addressed directly in JavaScript (we ignore `with` statements), and thus can only be pointed at by the env field. Environment objects do not have a prototype.

4. Type Checking of Heap Snapshots

We now show how a value from the heap snapshot can be checked against a type from the declaration file. In this section we ignore function signatures (i.e. they are assumed to be satisfied) and getters and setters (i.e. they are assumed to satisfy any type), which makes this type checking problem decidable.

We introduce typing judgements with five different forms:

$$\begin{array}{rcl} \sim & \subseteq & \textbf{\textit{value}} \times \textit{type} \ \cup \\ & & \textbf{\textit{prty}} \times \textit{type} \ \cup \\ & & \mathbf{O} \times \textit{prty} \ \cup \\ & & \mathbf{O} \times \textit{indexer} \ \cup \\ & & \mathbf{O} \times \textit{brand} \end{array}$$

The judgement of form $\mathbf{x} \sim \tau$ means the value $\mathbf{x}$ satisfies the type τ. A judgement $\mathbf{p} \sim \tau$ means the concrete property $\mathbf{p}$ has a value satisfying τ (or it has a getter/setter). A judgement $\mathbf{k} \sim \rho$ means the object with name $\mathbf{k}$ satisfies the property type ρ; likewise, $\mathbf{k} \sim i$ means the object with name $\mathbf{k}$ satisfies the indexer type i, and $\mathbf{k} \sim b$ means the object with name $\mathbf{k}$ satisfies the brand b.

We use the rules in Figure 2 to derive typing judgements. When a question mark occurs after a flag in the inference rules it means the rule can be instantiated with or without that flag in place. Type checking is initiated by checking if the global object satisfies its type, i.e., we attempt to derive the typing judgement $\mathbf{obj}(\mathbf{g}) \sim G\langle\rangle$ by goal-directed application of these rules.

The inference rules are cyclic, that is, the derivation of a typing judgement may loop around and depend on itself. When we encounter a cycle, which may happen due to cyclic structures in the heap snapshot and the corresponding types, we do not recursively check the judgement again, but coinductively assume that the nested occurrence of the judgement holds.

[3] `http://phantomjs.org`

$$\frac{}{\mathbf{x} \sim \mathsf{any}}\ [\text{any}] \qquad \frac{}{\mathbf{str}(s) \sim \mathsf{str}}\ [\text{str}]$$

$$\frac{}{\mathbf{num}(z) \sim \mathsf{num}}\ [\text{num}] \qquad \frac{}{\mathbf{str}(s) \sim \mathsf{"}s\mathsf{"}}\ [\text{str-const}]$$

$$\frac{}{\mathbf{true} \sim \mathsf{bool}}\ [\text{true}] \qquad \frac{}{\mathbf{undef} \sim \mathit{void}}\ [\text{undef}]$$

$$\frac{}{\mathbf{false} \sim \mathsf{bool}}\ [\text{false}] \qquad \frac{}{\mathbf{null} \sim \tau}\ [\text{null}]$$

$$\frac{\exists \overline{f} \in \Gamma(e).\ \mathcal{L}^*(\mathbf{g}, \overline{f}) = \mathbf{x}}{\mathbf{x} \sim e}\ [\text{enum}]$$

$$\frac{\Sigma(T) = \langle \overline{v} \rangle o, \quad \mathbf{x} \sim o[\overline{v} \to \overline{\tau}]}{\mathbf{x} \sim T\langle \overline{\tau} \rangle}\ [\text{ref}]$$

$$\frac{\begin{array}{c}\text{ToObj}(\mathbf{x}) = \mathbf{obj}(\mathbf{k}),\\ \forall \rho \in \overline{\rho}.\ \mathbf{k} \sim \rho, \quad \forall i \in \overline{i}.\ \mathbf{k} \sim i, \quad \forall b \in \overline{b}.\ \mathbf{k} \sim b\end{array}}{\mathbf{x} \sim \{\overline{\rho}\ ;\ \overline{i}\ ;\ \overline{c}\ ;\ \overline{b}\}}\ [\text{obj}]$$

$$\frac{\mathcal{L}(\mathbf{k}, f) = \mathbf{p}, \quad \mathbf{p} \sim \tau}{\mathbf{k} \sim \mathit{opt}?\ f : \tau}\ [\text{prty}] \qquad \frac{\mathcal{L}(\mathbf{k}, f) = \text{nil}}{\mathbf{k} \sim \mathit{opt}\ f : \tau}\ [\text{prty}_{\text{nil}}]$$

$$\frac{\forall \mathbf{p} \in \mathcal{L}(\mathbf{k}, *).\ \text{if } \mathbf{p}.\mathsf{enumerable} \text{ then } \mathbf{p} \sim \tau}{\mathbf{k} \sim [\mathsf{str} : \tau]}\ [\text{str-idx}]$$

$$\frac{\forall \mathbf{p} \in \mathcal{L}(\mathbf{k}, *).\ \text{IsNum}(\mathbf{p}.\mathsf{name}) \Rightarrow \mathbf{p} \sim \tau}{\mathbf{k} \sim [\mathsf{num} : \tau]}\ [\text{num-idx}]$$

$$\frac{\mathbf{p}.\mathsf{getter} = \mathbf{p}.\mathsf{setter} = \mathbf{null}, \quad \mathbf{p}.\mathsf{value} \sim \tau}{\mathbf{p} \sim \tau}\ [\text{p-value}]$$

$$\frac{\mathbf{p}.\mathsf{getter} \neq \mathbf{null}\ \vee\ \mathbf{p}.\mathsf{setter} \neq \mathbf{null}}{\mathbf{p} \sim \tau}\ [\text{p-getter}]$$

$$\frac{\mathcal{L}^*(\mathbf{g}, b) = \mathbf{obj}(\mathbf{k}'), \quad \mathbf{k}' \in \text{Protos}(\mathbf{k})}{\mathbf{k} \sim b}\ [\text{brand}]$$

$$\frac{\mathcal{L}^*(\mathbf{g}, b) = \text{nil}}{\mathbf{k} \sim b}\ [\text{brand}_{\text{nil}}]$$

Figure 2. Type derivation rules.

Several auxiliary functions are used in Figure 2; these are described in the following paragraphs.

The *property lookup* function $\mathcal{L} : \mathbf{O} \times S \to \textit{\textbf{prty}} \cup \{\text{nil}\}$ maps an object name and a property name to the corresponding property definition (if any) while taking into account inheritance from prototypes:

$$\mathcal{L}(\mathbf{k}, f) = \begin{cases} \mathbf{p} & \text{if } \mathbf{s}(\mathbf{k}) \text{ has a property } \mathbf{p} \text{ named } f \\ \mathcal{L}(\mathbf{k}', f) & \text{else if } \mathbf{s}(\mathbf{k}).\mathsf{prototype} = \mathbf{obj}(\mathbf{k}') \\ \text{nil} & \text{otherwise} \end{cases}$$

This is well-defined since prototype chains cannot be cyclic.

We use the notation $\mathcal{L}(\mathbf{k}, *)$ to denote the finite set of all properties available on the object with name $\mathbf{k}$. The phrase "has a property" used in the above definition means the properties sequence of the object should contain a property with the given name.

The *path lookup* function $\mathcal{L}^*$ operates on access paths instead of single property names. It returns a ***value*** instead of a ***prty***, and defaults to nil if the path cannot be resolved:

$$\mathcal{L}^* : \mathbf{O} \times S^* \to \textit{\textbf{value}} \cup \{\text{nil}\}$$

$$\mathcal{L}^*(\mathbf{k}, \overline{f}) = \begin{cases} \mathbf{obj}(\mathbf{k}) & \text{if } \overline{f} = \varepsilon \\ \mathcal{L}^*(\mathbf{k}', \overline{f'}) & \text{if } \overline{f} = f_1\overline{f'}\ \wedge\ \mathcal{L}(\mathbf{k}, f_1) = \mathbf{p} \\ & \quad \wedge\ \mathbf{p}.\mathsf{value} = \mathbf{obj}(\mathbf{k}') \\ \text{nil} & \text{otherwise} \end{cases}$$

The ToObj function, ToObj : ***value*** $\to$ ***value***, converts primitive values to objects:

$$\text{ToObj}(\mathbf{x}) = \begin{cases} \mathbf{obj}(\mathbf{k}) & \text{if } \mathbf{x} = \mathbf{obj}(\mathbf{k}) \\ \mathcal{L}^*(\mathbf{g}, \texttt{Number.prototype}) & \text{if } \mathbf{x} = \mathbf{num}(z) \\ \mathcal{L}^*(\mathbf{g}, \texttt{String.prototype}) & \text{if } \mathbf{x} = \mathbf{str}(s) \\ \mathcal{L}^*(\mathbf{g}, \texttt{Boolean.prototype}) & \text{if } \mathbf{x} = \mathbf{true} \\ \mathcal{L}^*(\mathbf{g}, \texttt{Boolean.prototype}) & \text{if } \mathbf{x} = \mathbf{false} \\ \mathbf{null} & \text{otherwise} \end{cases}$$

A path such as `Number.prototype` should be read as the path consisting of two components: the string `Number` followed by the string `prototype`. Note that the property named "`prototype`" should not be confused with the internal prototype link.

In JavaScript, a primitive value is automatically coerced to an object when it is used as an object. For instance, when evaluating the expression `"foo".toUpperCase()` in JavaScript, the primitive string value `"foo"` is first coerced to a string object, and this object inherits the `toUpperCase` property from the `String.prototype` object. To allow such expressions to be typed, TypeScript permits an automatic type conversion from primitive types to their corresponding object types. The ToObj function is used in the [obj] rule to mimic this behavior.

The Protos function, Protos : $\mathbf{O} \to \mathcal{P}(\mathbf{O})$, maps the name of an object to the set of objects on its prototype chain:

$$\text{Protos}(\mathbf{k}) = \begin{cases} \text{Protos}(\mathbf{k}') \cup \{\mathbf{k}\} & \text{if } \mathbf{s}(\mathbf{k}).\mathsf{prototype} = \mathbf{obj}(\mathbf{k}') \\ \{\mathbf{k}\} & \text{otherwise} \end{cases}$$

The substitution operator $o[\overline{v} \to \overline{\tau}]$ used in the [ref] rule produces a copy of o where for all i, all occurrences of the type variable v_i have been replaced by the type τ_i.

We will now discuss some of the typing rules in more detail.

Primitives The rules for the primitive types, any, num, bool, str, and $\mathsf{"}s\mathsf{"}$ are straightforward. Following the TypeScript specification, **null** satisfies any type, and **undef** satisfies the void type.

Enums As mentioned in Section 2, $\Gamma(e)$ denotes a set of access paths pointing to the members of enum e. The [enum] rule uses the path lookup function $\mathcal{L}^*$ to resolve each access path and check if the value is found at any of them.

Example Below is an enum declaration and an implementation in JavaScript:

```
60 // declaration file:
61 declare enum E { X, Y }

62 // JavaScript file:
63 var E = { X: "foo", Y: "bar" }
```

To check that the string value `"bar"` satisfies the enum type `E`, we can apply the [enum] rule with $\overline{f} = \texttt{E.Y} \in \Gamma(\texttt{E})$, and find that $\mathcal{L}^*(\mathbf{g}, \texttt{E.Y}) = \texttt{"bar"}$.

References The [ref] rule looks up a reference in the type environment Σ, instantiates the type with the given type arguments (if any), and recursively checks the value against the resolved type.

Example Below is a generic type `Pair<T>` and a variable whose type is an instantiation of this type:

```
64 // declaration file:
65 interface Pair<T> { fst: T; snd: T };
66 declare var x : Pair<number>;

67 // JavaScript file:
68 var x = { fst: 42, snd: 24 };
```

If we let $\mathbf{x}$ denote the concrete object created on line 68, the type checker will attempt to derive $\mathbf{x} \sim \texttt{Pair}\langle\mathsf{num}\rangle$. The [ref] rule will then be applied, after which the type checker will recursively attempt to derive $\mathbf{x} \sim \{\mathsf{fst} : \mathsf{num}; \mathsf{snd} : \mathsf{num}\}$.

Note that TypeScript requires that a generic type does not contain a reference to itself with type arguments wherein one of its type parameters have been wrapped in a bigger type. For instance, the following type declaration violates this rule:

```
69 interface F<T> { x: F<F<T>> }
```

This means only regular types can be expressed. Without this condition, termination of our type checking procedure would not be guaranteed.

Objects The [obj] rule coerces the value to an object, and checks separately that the resulting object (if any) satisfies each property, indexer, and brand member of the type.

Properties The [prty] and [prty$_{\mathrm{nil}}$] rules use the lookup function $\mathcal{L}$ to find the property with the given name (if any). The [prty$_{\mathrm{nil}}$] rule allows optional property members to be satisfied when the concrete property does not exist. If the property exists, the [prty] rule will check its type; in most cases this rule will be followed up by an application of the [p-value] rule. In case the property is defined by a getter and/or setter, the [p-getter] rule automatically accepts the typing judgement.

Indexers The [str-idx] rule checks that every enumerable property satisfies the type, and [num-idx] checks that every array entry (i.e. property whose name is a number string) satisfies the type. As with the [prty] rule, these rules are typically followed up by an application of [p-value].

Brands The [brand] and [brand$_{\mathrm{nil}}$] rules use the path lookup function $\mathcal{L}^*$ to find the brand's associated prototype object, and then checks that this object exists on the prototype chain. The [brand$_{\mathrm{nil}}$] rule exists to avoid duplicate errors in case a class appears to be missing, since this would otherwise be reported as a missing class constructor *and* as missing brands for every instance of the class.

Functions The [obj] rule as shown in Figure 2 ignores the function signatures of the type being checked against because checking these requires static analysis of the function body. We collect all such objects and corresponding function signatures in a list that will be checked in the static analysis phase described in Section 5.

4.1 Relationship to TypeScript

These type checking rules formalize the relation between values and types in TypeScript, which is only described implicitly and informally in the TypeScript language specification. We have strived toward a faithful formalization, however, in two situations we have decided to deviate from the language specification in order to match common uses of TypeScript types in declaration files.

First, the specification states that enum types are assignable to and from the number type, but different enum types are not assignable to each other (assignability is not transitive in TypeScript). While this may be appropriate for code written entirely in TypeScript, it does not reflect how JavaScript libraries work, where string values are often used as enum-like values. Despite the tight coupling to numbers, we see declaration files use enums to describe finite enumerations of string values. Thus, we developed the access-path based interpretation of enums to allow for more liberal use of the enum construct when checking against a JavaScript library.

Second, the TypeScript specification treats branded types purely as a compile-time concept. Since they are only used for class types, there is a simple runtime interpretation of brands, which we decided to include as reflected by the [brand] rule.

5. Static Analysis of Library Functions

The type checking algorithm presented in the preceding section is designed to detect mismatches between the object structure in the snapshot and the declarations, however, it does not check the function code. In this section we describe a static analysis for determining if a given function implemented in JavaScript satisfies a given function signature. The problem is undecidable, so any analysis must occasionally reject a correct function implementation, or conversely, accept an incorrect one. Following the philosophy of optional types, in which usability is valued higher than type safety, we aim toward the latter kind of analysis so as not to disturb the user with false warnings.

Figure 3 shows an example of a bug that was found using the algorithm presented throughout this section. On line 97,

Fragment of `d3.js` (library implementation)

```
70 d3.layout.bundle = function() {
71   return function(links) {
72     var paths = []
73     for (var i=0; i<links.length; ++i) {
74       paths.push(d3_layout_bundlePath(links[i]))
75     }
76     return paths;
77   };
78 };
79 function d3_layout_bundlePath(link) {
80   var start = link.source
81   var end = link.target
82   var lca = d3_layout_bundleLCA(start, end)
83   var points = [ start ]
84   while (start !== lca) {
85     start = start.parent
86     points.push(start)
87   }
88   var k = points.length
89   while (end !== lca) {
90     points.splice(k, 0, end)
91     end = end.parent
92   }
93   return points
94 }
95 function d3_layout_bundleLCA(a,b) {...}
```

Fragment of `d3.d.ts` (declaration file)

```
96  declare module d3.layout {
97      function bundle(): BundleLayout
98      interface BundleLayout{
99          (links: GraphLink[]): GraphNode[]
100     }
101     interface GraphLink {
102         source: GraphNode
103         target: GraphNode
104     }
105     interface GraphNode {
106         parent: GraphNode
107         /* some properties omitted ... */
108     }
109 }
```

Figure 3. Example of an incorrect function signature found in `d3.d.ts` (slightly modified for readability).

a function `bundle` is declared to return a `BundleLayout` that, as per line 99, itself must be a function taking a single `GraphLink[]` argument and returning a `GraphNode[]`. The implementation, however, returns a two-dimensional array of `GraphNode`s; the return type on line 99 should thus have been `GraphNode[][]`.

Our starting point is the collection of pairs of function objects and corresponding function signatures that was produced in connection with the [obj] rule, as explained in Section 4. For each of these pairs, we now analyze the function body and check that it is compatible with the function signature. For each program expression, the analysis will attempt to compute a type that over-approximates the set of values that may flow into the given expression. Once an approximate type has been found for every expression, we determine if the type inferred for the function's possible return values is assignment compatible with the function signature's declared return type. We will discuss the notion of assignment compatibility later, but for now, it can be thought of as the types having a nonempty intersection. If the types are not assignment compatible, i.e. their intersection is empty, then *any* value the function might return will violate the declared return type, indicating a clear mismatch between the declaration file and the library implementation. If the types are assignment compatible, on the other hand, we view the function signature as satisfied.

The analysis of each function can be seen as a two-step process. In the first step, a set of constraints is generated by a single traversal over the AST of the library code and the heap snapshot. The heap snapshot is used in the process, for example to resolve global variables. In the second step, the constraints are solved by finding a least fixed-point of some closure rules. We will introduce various types of constraints and discuss the most relevant AST constructs in the following subsections.

The analysis will involve all functions that might transitively be called from the function being checked. As an example, the implementation of `bundle` on lines 70–78 in Figure 3 uses the function `d3_layout_bundlePath` for which we do not have a function signature, so checking `bundle` also involves the code in `d3_layout_bundlePath`.

5.1 Atoms and Unification

To reason about the types of program expressions, we introduce a new type, called an *atom*:

$$\mathit{type} ::= \ldots \mid \mathit{atom}$$

There is an atom for every expression in the AST and for every object in the heap snapshot, and some additional atoms, which we will introduce as needed:

$$a \in \mathit{atom} ::= \text{ast-node} \mid \mathbf{O} \mid \ldots$$

These atoms can be seen as type variables, representing the type of a program expression (which is unknown due to lack of type annotations in the source code) or the type of a heap object (which is known at the time of the snapshot, but may later be modified by the program code, hence should also be seen as unknown).

We define a *term* to be an atom or an atom combined with a field name. A field can either be the name of a property (i.e. a string) or the special env field name that we use to model the chain of environment objects:

$$t \in \mathit{term} ::= \mathit{atom} \mid \mathit{atom} \diamond \mathit{field}$$
$$f \in \mathit{field} ::= S \mid \mathsf{env}$$

Like atoms, terms can be thought of as type variables. Intuitively, a compound term $a \diamond f$ acts as placeholder for the type one gets by looking up the property f on a. This lets us address the type of a's properties at a time where the type of a is still unknown. In an analogy to points-to analysis, the compound term $a \diamond f$ can be thought of as a placeholder for whatever is in the points-to set for $a.f$.

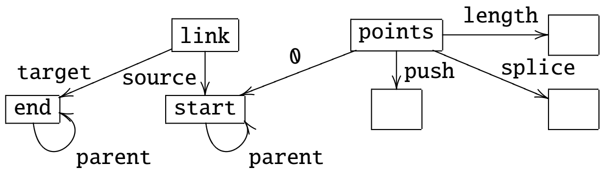
Our analysis computes an equivalence relation $\equiv \subseteq \mathit{term} \times \mathit{term}$ of terms that are deemed to have the same type. When $t_1 \equiv t_2$, we say that t_1 and t_2 have been *unified*. The analysis assigns a single type to each equivalence class in $\equiv$, so once terms have been unified, they can effectively be thought of as being the same term.

In addition to being an equivalence relation, $\equiv$ is closed under the [prty] rule, which states that once two atoms have been unified, all their fields must also be pointwise unified. In summary, $\equiv$ is closed under following four rules (the first three ones being the rules for equivalence relations).

$$\frac{t_1 \equiv t_2,\ t_2 \equiv t_3}{t_1 \equiv t_3}\ \text{[trans]} \qquad \frac{}{t \equiv t}\ \text{[refl]} \qquad \frac{t_1 \equiv t_2}{t_2 \equiv t_1}\ \text{[sym]}$$

$$\frac{a_1 \equiv a_2}{a_1 \diamond f \equiv a_2 \diamond f}\ \text{[prty]}$$

To gain some intuition about the meaning of this relation, one can think of it as a points-to graph in which every equivalence class of $\equiv$ is a vertex, and there is an edge labelled f from the class of an atom a to the class of $a \diamond f$. The following graph visualizes the state of $\equiv$ generated from `d3_layout_bundlePath` from Figure 3 (if ignoring function calls), where the labels inside each vertex indicate which atoms reside in the class:



We use a union-find data structure to represent the equivalence classes of $\equiv$. Every root node in the data structure contains its outgoing edges in a hash map. When two nodes are unified, their outgoing edges are recursively unified as well. More details of this procedure can be found in our previous work on JavaScript analysis for refactoring [2], which follows a related approach with a simpler constraint system.

The $\equiv$ relation allows us to handle three basic forms of program expressions:

Local Variables We unify all references to the same variable, thus capturing all data flow in and out of the variable. Closure variables will be discussed in Section 5.4.

Properties An expression e of form $e_1 . f$ is translated into the constraint $e \equiv e_1 \diamond f$. By the transitivity and the [prty] rule, e will become unified with all other occurrences of f on a similarly typed expression.

Assignment An expression e of form $e_1 = e_2$ is translated into the constraints $e \equiv e_1$ and $e \equiv e_2$.

5.2 Types

For every equivalence class in $\equiv$ we compute a set of types:

$$\mathit{Types} : \mathit{term} \xrightarrow{\equiv} \mathcal{P}(\mathit{type})$$

We use the $\xrightarrow{\equiv}$ symbol to denote functions that are congruent with $\equiv$, so that whenever $a_1 \equiv a_2$, we must have $\mathit{Types}(a_1) = \mathit{Types}(a_2)$. Our implementation maintains one set of types per root in the union-find data structure and merges the sets when two roots are unified.

The primary purpose of these types is to encode the types of the parameters from the function signature being checked. A type τ associated with a term t can be seen as a bound on the type of t. In an analogy to dataflow analysis, one can also think of τ as a set of values that may flow into t.

Types are propagated to other terms by the following rule, where $\tau.f$ indicates the lookup of property f on the type τ:

$$\frac{\tau \in \mathit{Types}(a)}{\tau.f \in \mathit{Types}(a \diamond f)}\ \text{[type-field]}$$

We unify two atoms if the type of one becomes bound by the other:

$$\frac{a_1 \in \mathit{Types}(a_2)}{a_1 \equiv a_2}\ \text{[type-same]}$$

An example is shown in Section 5.6. We also introduce primitive values as types, so we can use types for a kind of constant propagation (note that we never use **obj** values as types):

$$\mathit{type} ::= \ldots \mid \textbf{\textit{value}}$$

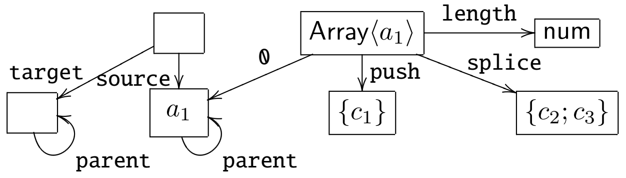
This lets us translate two more forms of basic expressions:

Literals A constant expression e with primitive value $\mathbf{x}$ is translated into the constraint $\mathbf{x} \in \mathit{Types}(e)$. Similarly, if e is a simple object literal {}, the constraint $\{\} \in \mathit{Types}(e)$ is generated.

Array Expressions For an expression e of form $[e_1, \ldots, e_n]$, we generate a new atom a to represent the members of the array being created. We then add the constraints $\mathsf{Array}\langle a \rangle \in \mathit{Types}(e)$ and $a \equiv e_i$ for every i.

Arithmetic Operators Arithmetic operators, such as `*` and `-`, are assumed to return values of type `num`, while the `+` operator is assumed to return a value of type `str` or `num`.

We can now continue the previous example with types on the vertices (again, we show the function in isolation without interprocedural information):



The atom a_1 represents the members of the `points` array, and c_1 is short for the function signature $\mathsf{vargs}(a_1) \Rightarrow \mathsf{num}$, which is the type for the `push` method on arrays of type $\mathsf{Array}\langle a_1\rangle$. Likewise, c_2 and c_3 represent the two signatures of the overloaded `splice` method on arrays. The [type-field] rule propagates types along the four outgoing edges from the node with type $\mathsf{Array}\langle a_1\rangle$, for example, the type num along the `length` edge.

5.3 Dynamic Property Accesses

A JavaScript expression of form $\mathtt{e_1[e_2]}$ is called a *dynamic property access*. When evaluated, the result of evaluating the expression $\mathtt{e_2}$ is coerced to a string, which is then looked up as a property on the object obtained from $\mathtt{e_1}$. This type of expression is also used to access the elements of an array. To handle this construct in the library code, we generate constraints of form $(a_{\text{obj}}, a_{\text{prty}}, a_{\text{res}}) \in \mathsf{DynAccess}$ where

$$\mathsf{DynAccess} \subseteq \overbrace{\mathit{atom}}^{\text{object}} \times \overbrace{\mathit{atom}}^{\text{property name}} \times \overbrace{\mathit{atom}}^{\text{result}}$$

We use the types of the property name to determine what properties might be addressed. For concrete primitive values we convert the value to a string and use that as a property name (where ToString models coercion of primitive values but ignores objects):

$$\frac{(a_{\text{obj}}, a_{\text{prty}}, a_{\text{res}}) \in \mathsf{DynAccess},\ \mathbf{x} \in \mathit{Types}(a_{\text{prty}})}{a_{\text{obj}} \diamond \mathsf{ToString}(\mathbf{x}) \equiv a_{\text{res}}}\ \text{[dyn-val]}$$

If the property is a computed number, we merge all numeric properties of the object:

$$\frac{(a_{\text{obj}}, a_{\text{prty}}, a_{\text{res}}) \in \mathsf{DynAccess},\ \mathsf{num} \in \mathit{Types}(a_{\text{prty}})}{\forall i \in \mathbb{N} : a_{\text{obj}} \diamond i \equiv a_{\text{res}}}\ \text{[dyn-num]}$$

If the property is a computed string, we merge *all* properties of the object:

$$\frac{(a_{\text{obj}}, a_{\text{prty}}, a_{\text{res}}) \in \mathsf{DynAccess},\ \mathsf{str} \in \mathit{Types}(a_{\text{prty}})}{\forall s \in S : a_{\text{obj}} \diamond s \equiv a_{\text{res}}}\ \text{[dyn-str]}$$

The last two rules are implemented by keeping two flags on every root in the union-find data structure indicating whether its numeric properties, or all its properties, respectively, should be merged.

5.4 Function Calls

Functions are first-class objects in JavaScript. To reason about the possible targets of a function call inside the library we therefore track a set of functions for each atom:

$$\mathit{Funcs} : \mathit{atom} \xrightarrow{\equiv} \mathcal{P}(\boldsymbol{fun})$$

The set $\mathit{Funcs}(a)$ contains the user-defined and native functions that may flow into a.

We use the function FunDef to map function names U to the atoms representing the input and output of the function:

$$\mathsf{FunDef} : U \to \overbrace{\mathit{atom}}^{\text{fun. instance}} \times \overbrace{\mathit{atom}}^{\text{this arg}} \times \overbrace{\mathit{atom}}^{\text{arg tuple}} \times \overbrace{\mathit{atom}}^{\text{return}}$$

The first three atoms represent inputs to the function: the function instance itself, the `this` argument, and the `arguments` object that contains the actual argument in its properties. The last atom represents the function's return value.

Similarly, we maintain a set *Calls*, also consisting of tuples of four atoms:

$$\mathit{Calls} \subseteq \overbrace{\mathit{atom}}^{\text{fun. instance}} \times \overbrace{\mathit{atom}}^{\text{this arg}} \times \overbrace{\mathit{atom}}^{\text{arg tuple}} \times \overbrace{\mathit{atom}}^{\text{return}}$$

The first three atoms represent the function instance, the `this` argument, and the `arguments` object passed to the called function, and the last atom denotes where to put the value returned by the function. Each syntactic occurrence of a function or a function call gives rise to an element in *Funcs* or *Calls*, respectively. Constructor calls (i.e. calls using the `new` keyword) will be discussed separately in Section 5.8.

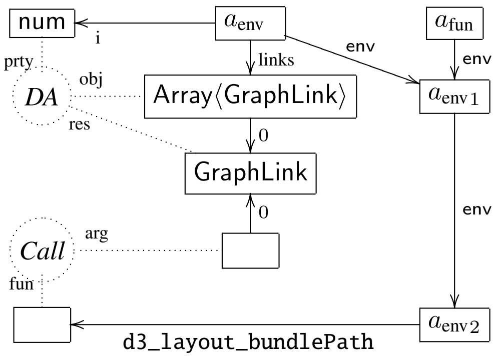
For every potential call target at a function call, we unify the corresponding atoms:

$$\frac{(a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \in \mathit{Calls},\ \mathbf{user}(u) \in \mathit{Funcs}(a_{\text{fun}})}{(a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \doteq \mathsf{FunDef}(u)}\ \text{[call]}$$

The symbol $\doteq$ should be read as the pointwise application of $\equiv$, for example, the arguments of the call are unified with the arguments from the function definition. Note that our treatment of function calls is context insensitive.

Closure variables (i.e. variables defined in an outer scope) are accessible to the called function through the function instance argument, whose `env` field points to its enclosing environment object. Each environment object is represented by an atom whose `env` field points to its own enclosing environment, ending with the global object.

As an example, the diagram below shows some of the constraints generated for the function on lines 71–77. The call to `paths.push` is omitted for brevity. The *DA* node represents an entry in DynAccess generated for the `links[i]` expression, and the *Call* node represents the entry in *Calls* that was generated for the call to `d3_layout_bundlePath` (though we omit its associated `this` and return atoms):



5.5 Entry Calls

The analysis maintains a set of *entry calls* that denote places where the library code gets invoked by an external caller (i.e.

from the client code or the native environment). An entry call consists of two atoms and a function signature:

$$\mathit{EntryCalls} \subseteq \overbrace{\mathit{atom}}^{\text{function}} \times \overbrace{\mathit{atom}}^{\text{this arg}} \times \mathit{fun\text{-}sig}$$

The first atom denotes the function instance being invoked, and the other denotes the `this` argument used in the call. The function signature associated with an entry call provides type information for the parameters.

Initially, a single entry call exists to the function being checked against a function signature. This serves as the starting point of the analysis, but more entry calls may be added during the analysis, as will be discussed in Section 5.6.

The following rule describes how entry calls are handled (for simplicity, ignoring variadic functions, optional parameters, and constructor calls):

$$\frac{\begin{array}{c}(a_{\text{fun}}, a_{\text{this}}, (\overline{\tau}) \Rightarrow \tau') \in \mathit{EntryCalls},\\ \mathbf{user}(u) \in \mathit{Funcs}(a_{\text{fun}}),\\ \mathsf{FunDef}(u) = (a'_{\text{fun}}, a'_{\text{this}}, a_{\text{arg}}, a_{\text{ret}})\end{array}}{a_{\text{fun}} \equiv a'_{\text{fun}},\ a_{\text{this}} \equiv a'_{\text{this}},\ \forall i. \tau_i \in \mathit{Types}(a_{\text{arg}} \diamond i)}\ [\text{entry}]$$

In the previous example, the [entry] rule was responsible for adding the $\mathsf{Array}\langle\mathsf{GraphLink}\rangle$ type.

Note that calls between two library functions are handled as explained in Section 5.4, not as entry calls, even if the called function has a signature in the declaration file.

5.6 Exit Calls

When we encounter a function call to a native function or a function that was provided through an argument, we handle the call using the function signature for the function. We refer to these as *exit calls* as they target functions that are defined outside the library code. For brevity, we give only an informal description of our treatment of exit calls.

When resolving a call $(a_{\text{fun}}, a_{\text{this}}, a_{\text{arg}}, a_{\text{ret}}) \in \mathit{Calls}$ where $\mathit{Types}(a_{\text{fun}})$ contains an object type with a function signature c, we check if the arguments might satisfy the parameter types of c by an assignment compatibility check, and if the check succeeds, the return type of the function signature is added to $\mathit{Types}(a_{\text{ret}})$. The check helps handle overloaded functions by filtering out signatures that are not relevant for a given call site. To model callbacks made from the external function, whenever an atom a is compared to an object type with a function signature c and there is a function $\mathbf{user}(u) \in \mathit{Funcs}(a)$, we register a new entry call to a with the function signature c.

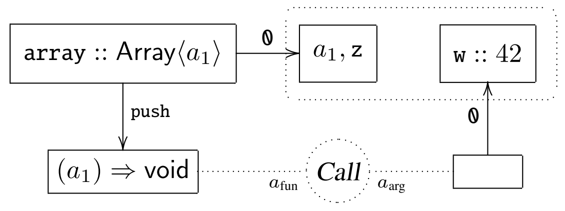
If the function signature being invoked is polymorphic, the compatibility check against the parameter types also serves to instantiate the type variables in the function signature. Whenever a type variable V is compared to an atom a in the compatibility check, a is registered as a bound on V. If the entire compatibility check succeeds then all atoms that bind the same type variable are unified.

The standard library is modeled using the types from the TypeScript compiler's `lib.d.ts` file, except for `call` and `apply` which are given special treatment by the analysis.

The example below demonstrates how exit calls can handle array manipulation:

```
110 var array = [];
111 var w = 42;
112 array.push(w);
113 var z = array[0];
```

Below is the state of the constraint system before the call to `push` has been resolved (here we use :: to separate atom names from their types):



The assignment on line 113 has caused `array ◇ 0` to be unified with `z`. The [type-field] rule has then propagated a_1 as a type to `z`, and [type-same] then caused a_1 and `z` to become unified.

The call to `push` is an exit call to a function with signature $(a_1) \Rightarrow \mathsf{void}$. The atom a_1 occurs as parameter type in this signature, and when it gets matched against the argument `w` the two atoms are unified. The dotted box above shows this unification.

This demonstrates how the value 42 propagated from `w` through the array into the variable `z`. Note that such destructive update of the arguments only happens when an atom occurs as part of a parameter type. This feature is our only way of modeling the side effects of exit calls.

5.7 Prototypes

One possible way to handle JavaScript's prototype-based inheritance mechanism is to unify two atoms a_1 and a_2 if one inherits from the other. In practice, however, almost all objects would then become unified with `Object.prototype`, thereby destroying precision entirely. We instead represent inheritance by maintaining a set of prototypes for each atom:

$$\mathit{Protos} : \mathit{atom} \xrightarrow{\equiv} \mathcal{P}(\mathit{atom})$$

Intuitively, the set $\mathit{Protos}(a)$ indicates what objects may reside on the prototype chain of a. The following rule ensures that properties that exist on a_1 will be shared with a_2:

$$\frac{a_1 \in \mathit{Protos}(a_2),\quad \mathsf{NonEmpty}(a_1 \diamond f)}{a_1 \diamond f \equiv a_2 \diamond f}\ [\text{inherit}]$$

The predicate $\mathsf{NonEmpty}(t)$ holds if the current type associated with t is nonempty; we use it to check the existence of fields in the prototype.

$$\mathsf{NonEmpty}(t) = (\mathit{Types}(t) \neq \emptyset \vee \exists a.\, t \equiv a)$$

Without this check, by transitivity all instances of the same prototype would share all properties, effectively making inheritance the same as unification.

5.8 Constructor Calls

Like regular functions calls, constructor calls are recorded in a set of tuples:

$$\mathit{NewCalls} \subseteq \overbrace{\mathit{atom}}^{\text{fun. instance}} \times \overbrace{\mathit{atom}}^{\text{arg tuple}} \times \overbrace{\mathit{atom}}^{\text{return}}$$

Constructor calls have no explicit `this` argument; instead, a newly created object is passed as `this` and is (in most cases) returned by the call.

$$\frac{(a_{\text{fun}}, a_{\text{arg}}, a_{\text{ret}}) \in \mathit{NewCalls},\ \mathbf{user}(u) \in \mathit{Funcs}(a_{\text{fun}})}{(a_{\text{fun}}, a_{\text{ret}}, a_{\text{arg}}, a_{\text{ret}}) \doteqdot \mathsf{FunDef}(u)}\ \text{[new]}$$

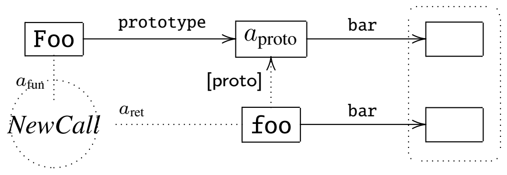
The prototype of the newly created object is the object referred to by the `prototype` property of the function being called. For every constructor call we generate an atom a_{proto} to represent the prototype of the newly created object, and add this as a prototype of the created object:

$$a_{\text{fun}} \diamond \mathsf{prototype} \equiv a_{\text{proto}}, \quad a_{\text{proto}} \in \mathit{Protos}(a_{\text{ret}})$$

As an example, consider this program fragment:

```
114 function Foo(x) { /* ... */ }
115 Foo.prototype.bar = function() { /* ... */ }
116 var foo = new Foo();
117 foo.bar();
```

Below is the constraint system before applying the [inherit] rule (not showing the *Calls* constraint):



The dotted box on the right shows the two atoms that will be unified when applying the [inherit] rule. The topmost of these corresponds to the function created on line 115 and the bottommost one corresponds to the `bar` property mentioned on line 117.

5.9 Generating Constraints from the Snapshot

We now briefly discuss how information from the heap snapshot is embedded in the constraint system. Recall that there exists an atom for every object in the heap. The following rule connects pointers between objects:

$$\frac{\mathbf{p} \in \mathcal{L}(\mathbf{k}, *),\ \mathbf{p}.\mathsf{value} = \mathbf{obj}(\mathbf{k}')}{\mathbf{k} \diamond \mathbf{p}.\mathsf{name} \equiv \mathbf{k}'}\ \text{[obj-value]}$$

For getters or setters, we generate a call to the getter/setter and unify the result/argument with the term representing the property. Thereby the effects of the getters and setters will be taken into account whenever the property is referenced, without needing to generate calls at every use site:

$$\frac{\mathbf{p} \in \mathcal{L}(\mathbf{k}, *),\ \mathbf{p}.\mathsf{get} = \mathbf{obj}(\mathbf{k}')}{(\mathbf{k}', \mathbf{k}, a_{\text{arg}}, a_{\text{ret}}) \in \mathit{Calls},\ a_{\text{ret}} \equiv \mathbf{k} \diamond \mathbf{p}.\mathsf{name}}\ \text{[obj-get]}$$

$$\frac{\mathbf{p} \in \mathcal{L}(\mathbf{k}, *),\ \mathbf{p}.\mathsf{set} = \mathbf{obj}(\mathbf{k}')}{(\mathbf{k}', \mathbf{k}, a_{\text{arg}}, a_{\text{ret}}) \in \mathit{Calls},\ a_{\text{arg}} \diamond 0 \equiv \mathbf{k} \diamond \mathbf{p}.\mathsf{name}}\ \text{[obj-set]}$$

For properties that contain primitive values, we add the primitive to the types of the corresponding term:

$$\frac{\mathbf{p} \in \mathcal{L}(\mathbf{k}, *),\ \mathbf{p}.\mathsf{value} = \mathbf{x},\ \mathsf{IsPrimitive}(\mathbf{x})}{\mathbf{x} \in \mathit{Types}(\mathbf{k} \diamond \mathbf{p}.\mathsf{name})}\ \text{[obj-primitive]}$$

Finally, the function objects are registered in *Funcs*:

$$\frac{\mathbf{k} \in \mathbf{O}}{s(\mathbf{k}).\mathsf{function} \in \mathit{Funcs}(\mathbf{k})}\ \text{[obj-func]}$$

5.10 Assignment Compatibility

TypeScript defines assignment compatibility between types as a more liberal version of subtyping. It can be seen as a bi-directional subtyping check, that is, two types are assignment compatible if either type is a subtype of the other. When checking object types for assignment compatibility, the rule is applied recursively for each property. As an example, the types $\{\mathsf{opt}\, x : \mathsf{num}; y : \mathsf{num}\}$ and $\{\mathsf{opt}\, z : \mathsf{num}; y : \mathsf{num}\}$ are assignment compatible despite neither being a subtype of the other.

We extend the notion of assignment compatibility to handle atoms and values (the two new types introduced in this section). The full definition is too verbose for this presentation, so we will briefly describe how atoms are checked against types. When checking an atom a (or any term that has been unified with an atom a) for assignment compatibility with an object type o, we recursively check the types of $a \diamond f$ against the type of every property f on o. When checking the type of a non-atom term t against a type τ, the check succeeds if any of the types in $\mathit{Types}(t)$ is assignment compatible with τ.

6. Evaluation

Our approach is implemented in the tool chain JSNAP,[4] TSCORE,[5] and TSCHECK,[5] for producing snapshots, extracting TypeScript Declaration Core declarations from declaration files, and checking a snapshot against the declarations, respectively. All tools are implemented in JavaScript.

With these tools, we conduct experiments to establish the usefulness of our approach. Specifically, we wish to determine how many warnings our technique produces in practice, and how many of these warnings are indicators of real bugs.

[4] `https://github.com/asgerf/jsnap`

[5] `https://github.com/asgerf/tscheck`

We selected the 10 largest declaration files from the Definitely Typed repository (mentioned in the introduction) for which we were able to obtain the correct version of the library code. We ran our tool on each benchmark and manually classified each warning as one of the following:

Declaration: This warning is due to a bug in the declaration file.

Library: This warning is due to a mismatch between the library implementation and its official documentation.

Spurious: This is a false warning, i.e. it does not indicate an actual bug in the declaration file or library code.

Note that when multiple warnings are due to the same bug in the declaration file, they are only counted as one, since duplicate warnings provide no additional value to the user.

We observe that some bugs in the declaration file are of little significance in practice, so we further classify the *declaration* warnings according to their impact:

High impact: The bug causes tools to misbehave, e.g. the TypeScript compiler may reject correct client code, or code completion may provide misleading suggestions.

Low impact: Although the declaration is technically incorrect, the TypeScript compiler and its code completion services will behave as intended. A typical example of this is a boolean property that should be declared optional.

Likewise, the *spurious* warnings are also classified further:

Initialization: The warning is due to the initialization assumption, i.e. parts of the API are not available until some other functions have been called or an event has fired.

Platform: The warning is due to the platform assumption, i.e. parts of the API are deliberately only made available on certain platforms.

Unsound analysis: The warning was produced because the static analysis did not infer a proper over-approximation of a function's possible return values and the compatibility check subsequently failed.

The results are shown in Table 1. The *H* and *L* columns show the number of declaration bugs found, respectively of high impact and low impact. The *lib* column shows the number of library bugs found. The *I*, *P*, and *U* columns show the number of spurious warnings due to initialization, platform, and unsound analysis, respectively.

With 142 high-impact bugs, this evaluation shows that our technique is capable of finding a large number of bugs in TypeScript declaration files, and even detect some bugs the library code. Only 22% of the warnings are spurious, most of which pertain to the `sugar` library, which uses some native JavaScript functions in a way that our analysis currently does not support.

The running time is dominated by the static analyzer, which took an average of 1.1 seconds per checked function

	lines of code		*decl*		*lib*	*spurious*		
benchmark	`.js`	`.d.ts`	*H*	*L*		*I*	*P*	*U*
`ace`	13,483	615	1	0	0	0	0	0
`fabricjs`	8,584	740	8	0	0	0	1	1
`jquery`	6,043	526	0	1	0	0	0	1
`underscore`	885	641	0	4	0	0	0	0
`pixi`	5,829	361	4	0	1	1	0	0
`box2dweb`	10,718	1,139	9	0	0	0	0	3
`leaflet`	5,873	707	13	4	0	0	0	1
`threejs`	19,065	2,651	49	29	2	1	0	10
`d3`	9,225	1,558	21	1	0	0	0	6
`sugar`	4,133	650	(37)	0	1	0	0	28
TOTAL			142	39	4	2	1	50

Table 1. Classification of warnings on the 10 benchmarks.

signature on a Intel Core i3 3.0 GHz PC. The average analysis time was 176 seconds per benchmark, with the slowest being the `sugar` library whose 420 function signatures took 628 seconds to check.

Of the high-impact bugs, 68 were detected by the snapshot analysis (Section 4), and the remaining 74 were detected by the static analysis (Section 5).

6.1 Examples

A recurring type of high-impact bug was found when the result of a function gets passed through an asynchronous callback rather than being returned immediately. For instance, in `fabricjs`, we found the declaration (simplified here):

```
declare var Image : { ...
    fromURL(url: string): IImage;
}
```

`Image.fromURL` in fact takes a callback argument to be invoked when the image has been loaded (asynchronously); the function call itself returns nothing.

An example of a spurious warning was found in the following fragment of the `leaflet` library (simplified):

```
// declaration file:
declare var DomUtil : {
    setPosition(el: HTMLElement, point: Point);
    getPosition(el: HTMLElement): Point
}
// implementation:
var DomUtil = { ...
    setPosition: function(el, point) {
        el._leaflet_pos = point;
        /* rest of code omitted */
    },
    getPosition: function (el) {
        return el._leaflet_pos;
    }
}
```

A comment in the source code suggests that `getPosition` should only be used after `setPosition` and based on this rationale assumes that the position stored in `setPosition` line 129 is available. However, since we check each function signature in isolation, our static analysis does not observe the field established in `setPosition` when checking `getPosition`. The expression `el._leaflet_pos` is thus determined to return `undefined`, which fails to check against the `Point` type, and our analysis thus issues a spurious warning regarding the return type of `getPosition`.

6.2 Limitations in TypeScript

The high-impact warnings for the `sugar` library are shown in parentheses because it is clear from the declaration file that the developer is already aware of all of them. The bugs are currently unfixable due to a limitation in TypeScript that makes it impossible to extend the type of a variable that was defined in TypeScript's prelude declaration file (the prelude is implicitly included from all other declaration files, similar to the `java.lang` package in Java). These warnings were all reported by the first phase of the algorithm.

We observed another limitation of TypeScript in the declaration file for the `leaflet` library. This library configures itself based on the presence of certain global variables, such as `L_DISABLE_3D`, intended to be set by the client. It is helpful for the declaration file to document these variables, but the variables are absent by default, and global variables cannot be declared optional in TypeScript (though our core language allows it). To spare the user from such low-impact warnings that cannot be fixed, we configured our analyzer to omit all warnings regarding missing boolean properties if the property occurs in a position where it cannot be declared optional.

Lastly, we observed that the `threejs` library declares several enum types with no members, because the members of the enum are not encapsulated in a namespace object like TypeScript expects, as shown in the following fragment:

```
enum CullFace { }
var CullFaceBack: CullFace;
var CullFaceFront: CullFace;
```

To compensate, we configured our analyzer to treat empty enums as the `any` type.

None of these limitations exist in TypeScript Declaration Core (Section 2.1), so a more liberal syntax for TypeScript would alleviate the issues.

7. Related Work

Most high-level programming languages provide interfaces to libraries written in low-level languages. The challenge of ensuring consistency between the low-level library code and the high-level interface descriptions has been encountered for other languages than TypeScript and JavaScript. Furr and Foster have studied the related problem of type checking C code that is accessed via the foreign function interfaces of Java or OCaml [3]. Due to the major differences between JavaScript and C and between the notion of types in TypeScript compared to those in Java or OCaml, their type inference mechanism is not applicable to our setting. St-Amour and Toronto have proposed using random testing to detect errors in the base environment for Typed Racket numeric types, which are implemented in C [13]. We believe it is difficult to apply that technique to TypeScript types and JavaScript libraries, since TypeScript types are considerably more complex than the numeric types in Typed Racket. Another essential difference between these languages is that JavaScript libraries are initialized dynamically without any static declaration of types and operations, which we have chosen to address by the use of heap snapshots.

The dynamic heap type inference technique by Polishchuk et al. [11] is related to our snapshot type checking phase (Section 4), but is designed for C instead of TypeScript, which has a different notion of types. Unlike C, JavaScript and TypeScript are memory-safe languages where all runtime values are typed (with JavaScript types, not TypeScript types), and we can disregard the call stack, which leads to a simpler algorithm in our setting.

To the best of our knowledge, no previous work has addressed the challenge of automatically finding errors in library interface descriptions for JavaScript, nor for dynamically typed programming languages in general. TypeScript is by design closely connected to JavaScript, but other statically typed languages, such as Dart [5] or Java via GWT [4], contain similar bindings to JavaScript, to enable applications to build upon existing JavaScript libraries. Despite the young age of TypeScript, the immense volume of the Definitely Typed repository testifies to the popularity of the language and the importance of interacting with JavaScript libraries.

Several more expressive type systems for JavaScript than the one in TypeScript have been proposed, including TeJaS [9] and DJS [1]. We only use TypeScript's meaning of types, not its rules for type checking program code, and it may be possible to adapt our system to such alternative type systems, if they were to introduce language bindings to JavaScript libraries.

The static analysis in Section 5 is a unification-based analysis in the style of Steensgaard [14] and bears some resemblance to the one used in our previous work on refactoring [2], although that work did not deal with function calls, prototype-based inheritance, dynamic property access, heap snapshots, or types. Many other static analysis techniques have been developed for JavaScript, for example, TAJS, which is designed to detect type-related errors in JavaScript applications [7], WALA [12], and Gatekeeper [6]. None of these incorporate type declarations into the analysis, and they focus on soundness, not performance.

A potential alternative to our approach could be to apply hybrid type checking [8] by instrumenting the program code to perform runtime checks at the boundary between TypeScript applications and JavaScript libraries. By the use of static analysis, we avoid the need for instrumentation and high-coverage test suites.

8. Conclusion

We have presented an approach to automatically detect errors in TypeScript declaration files for JavaScript libraries. Our key insights are that the API of such a library can be captured by a snapshot of the runtime state after the initialization and that the declarations can be checked on the basis of this snapshot and the library function code using a lightweight static analysis. Our experimental evaluation shows the effectiveness of the approach: it successfully reveals errors in the declaration files of most of the libraries we have checked using our implementation.

In addition to being a useful tool for authors of TypeScript declaration files for JavaScript libraries, our implementation has led us to identify some mismatches between the TypeScript language specification and common uses of types in declaration files, and some practical limitations of the expressiveness of its type system, which may be helpful in the future development of the language.

Acknowledgments This work was supported by Google and the Danish Research Council for Technology and Production.

References

[1] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proc. 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[2] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proc. 28th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages, and Applications*, 2013.

[3] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems*, 30(4), 2008.

[4] Google. GWT - JavaScript Native Interface, March 2013. `http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html`.

[5] Google. The Dart programming language specification, March 2014. `https://www.dartlang.org/docs/spec/`.

[6] S. Guarnieri and V. B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. 18th USENIX Security Symposium*, 2009.

[7] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, 2009.

[8] K. L. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2), 2010.

[9] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *Proc. 9th Symposium on Dynamic Languages*, 2013.

[10] Microsoft. TypeScript language specification, February 2014. `http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf`.

[11] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[12] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proc. 26th European Conference on Object-Oriented Programming*, 2012.

[13] V. St-Amour and N. Toronto. Experience report: applying random testing to a base type environment. In *Proc. 18th ACM SIGPLAN International Conference on Functional Programming*, 2013.

[14] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.