Mixed Messages: Measuring Conformance and Non-Interference in TypeScript

Jack Williams[1], J. Garrett Morris[2], Philip Wadler[3], and Jakub Zalewski[4]

1 **University of Edinburgh, Edinburgh, Scotland**
`jack.williams@ed.ac.uk,`
2 **University of Edinburgh, Edinburgh, Scotland**
`Garrett.Morris@ed.ac.uk`
3 **University of Edinburgh, Edinburgh, Scotland**
`wadler@inf.ed.ac.uk`
4 **University of Edinburgh, Edinburgh, Scotland**
`jakub.zalewski@ed.ac.uk`

Abstract

TypeScript participates in the recent trend among programming languages to support gradual typing. The DefinitelyTyped Repository for TypeScript supplies type definitions for over 2000 popular JavaScript libraries. However, there is no guarantee that implementations conform to their corresponding declarations.

We present a practical evaluation of gradual typing for TypeScript. We have developed a tool for use with TypeScript, based on the polymorphic blame calculus, for monitoring JavaScript libraries and TypeScript clients against the TypeScript definition. We apply our tool, TypeScript TPD, to those libraries in the DefinitelyTyped Repository which had adequate test code to use. Of the 122 libraries we checked, 62 had cases where either the library or its tests failed to conform to the declaration.

Gradual typing should satisfy non-interference. Monitoring a program should never change its behaviour, except to raise a type error should a value not conform to its declared type. However, our experience also suggests serious technical concerns with the use of the JavaScript proxy mechanism for enforcing contracts. Of the 122 libraries we checked, 22 had cases where the library or its tests violated non-interference.

1998 ACM Subject Classification D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords and phrases Gradual Typing, TypeScript, JavaScript, Proxies

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.39

1 Introduction

We have good news and we have bad news. The good news: gradual typing can be used to enforce conformance between type definitions and implementations of JavaScript libraries used with TypeScript clients. The bad news: technical concerns with the use of JavaScript proxies to enforce contracts are a real problem in practice.

Optional typing integrates static and dynamic typing with the aim of providing the best of both worlds, and can be found in languages including C#, Clojure, Dart, Python, and TypeScript. TypeScript [16] extends JavaScript with optional type annotations, with an aim to improving documentation and tooling. For example, auto-completion is made more precise by providing suggestions compatible with the inferred type. TypeScript is *unsound* *by design*: type inference provides a plausible candidate for the type of code, but falls short of a guarantee that values returned by code will conform to the inferred type. TypeScript instead favours convenience and ease of interoperation with JavaScript [4].

JavaScript's popularity depends upon (and leads to) the existence of a large number of libraries. TypeScript allows developers to import JavaScript libraries into TypeScript clients, using a definition file to specify the types at which the client may invoke library members. The definition file is separate from the library in order to permit legacy JavaScript libraries to be imported without change. The DefinitelyTyped repository [5] is the primary hub for aggregating definition files, with over 2000 definitions.

JavaScript libraries and TypeScript clients should *conform* to the definition file. For instance, when calling a library function the client should supply an argument of the correct type, and the library should return a result of the correct type. Some static type checking (not necessarily sound) is done for a client's conformance to the definition, and no checking is carried out on the library itself. Since many of the contributors of definition files are not authors of the corresponding JavaScript library, mistakes can easily creep in. Further, maintenance of the definition file may not keep in lock step with maintenance of the library. As a consequence, developers may be provided with misleading auto-complete suggestions, and, more insidiously, be led to introduce hard-to-detect bugs.

Gradual typing [24, 32] is a method for integrating dynamic and static types whilst guaranteeing soundness, by inserting run-time checks at the boundaries between typed and untyped code. The theory of gradual typing has been extended to support references [24], objects [23], refinement types [39], polymorphism [1], intersections and unions [13].

We have developed a tool, 'TypeScript: The Prime Directive', or TPD for short, which applies gradual typing to TypeScript. (In previous versions of the paper the tool was named *TypeScript: The Next Generation.*) TPD dynamically monitors libraries and clients to ensure they conform to the corresponding definition. We tested our system on every library in the DefinitelyTyped repository that runs under Node.js and was accompanied by a test suite that passed all its tests (without using TPD). At the time this work began, the repository contained 500 libraries, of which 122 satisfied our criteria. For each such library, we applied TPD and classified failures to conform to the definition. TPD revealed failures in 62 of the 122 libraries, totalling 179 distinct errors.

The intention of TPD is to provide gradual typing for a JavaScript library and its client without having to modify existing code. Each definition generates wrapper code that enforces a *contract* between library and client by monitoring whenever a field is read or written or a method is invoked. The wrapper code checks that values passed by the client and returned from the library conform to the correct type, and assigns *blame* appropriately to either the client or library when the contract is violated. To this end, TPD uses opaque proxies to implement type wrappers.

Gradual typing should satisfy *non-interference.* Monitoring a program should never change its behaviour, except to raise a type error should a value not conform to its declared type. A similar principle exists in the *Star Trek* universe known as the *Prime Directive.* Monitoring a planet should never interfere with the development of said planet. Unfortunately, the current design of proxies for JavaScript makes it impossible to ensure non-interference in all cases, a bit like adherence to the Prime Directive in the television shows. Van Cutsem and Miller [33, 34] propose a proxy mechanism for JavaScript, which has been adopted in the most recent JavaScript standard. Keil and Thiemann [12] show that using proxies may cause interference in theory, but don't address the question of how likely one is to encounter the issue in practice.

Work by Keil et al. [11] evaluates interference caused by the use of opaque proxies for contract checking. They study the effect of proxies on individual equality tests during the execution of annotated benchmarks. An object and its proxy do not have the same object identity. It may be that in the original code an object is compared with itself, while in monitored code an object is compared with a wrapped version of itself, or two differently wrapped version of the same object are compared. Hence, a comparison that previously returned true might return false in monitored code—a violation of non-interference.

Keil et al. [11] only consider interference due to proxies changing identity. We identify a new source of interference caused by the use of dynamic sealing to enforce parametric polymorphic contracts. Sealed data may react differently to certain operations, notably `typeof`, leading to violations of non-interference. Our evaluation considers all types of interference caused by proxies. TPD caused interference in 22 of the 122 libraries tested, of which 12 were due to proxy equality, five were due to sealing, 4 were due to reflection, and 2 were due to issues in the proxy implementation.

Gradual typing provides a mechanism for ensuring that library and client conform to the type definition. Our application and evaluation of gradual typing for TypeScript provides mixed messages. Our experiments, along with others [7], show that definition files are prone to error. Proxies should be an ideal technique for implementing gradual typing; a proxy allows wrapper code to be attached without having to modify the source. Our results show that using opaque proxies to implement gradual typing in JavaScript is not a viable method. For gradual typing to succeed in JavaScript, programmers must be provided with an alternative to opaque proxies. Some alternatives [11] have been presented that ameliorate the problem of proxy identity, but do not consider the problem of dynamic sealing. We consider the challenges associated with implementing dynamic seals and whether a suitable solution exists.

The main contributions of this paper are:

- We present the core concepts behind the implementation of TypeScript TPD, including the use of proxies to implement polymorphic wrappers and seals. We discuss the issues that arise when implementing dynamic seals using proxies. (Section 2).
- We give a series of examples from the DefinitelyTyped repository that illustrate how library or client files may fail to conform to the given definition file, and show how TPD helps to detect such failures. (Section 3).
- We present examples of proxies causing interference in monitored JavaScript libraries drawn from our testing of the DefinitelyTyped repository. Examples of both interference caused by identity changes and interference caused by dynamic seals are shown. (Section 4).
- We give the results of our measurements on 122 libraries in the DefinitelyTyped repository, recording when the tool detected a library that did not conform to its definition. We analyse the different causes for failure of non-interference, and count the number of cases of each kind of failure. (Section 5).

Section 6 discusses alternatives and proposed solutions in the context of our results, Section 7 presents related work, and Section 8 concludes.

2 Concepts of TPD: Functions, Polymorphism, and Proxies

This section outlines the design of TPD, including two central concepts, function wrapping and dynamic sealing. We describe how they are implemented using proxies, and also describe various problems associated with this use of opaque proxies.

2.1 Wrapping

TypeScript TPD takes a JavaScript library and wraps each library export according to its type as specified in the corresponding TypeScript definition file. All libraries export a single object that provides their API. A definition file may explicitly declare that object using the `export =` notation. Below is an example definition file using the explicit notation.

```
// example1.d.ts
declare function foo(x: number): number
export = foo;
```

The object exported by the library is the function `foo`, which accepts an argument of type number, and returns a result of type number. TPD will wrap the function `foo` in a contract for the corresponding type `(x: number) => number`.

Another way to write a definition file is to declare the individual members of the exported object. Below is an example definition that declares the exported object's members.

```
// example2.d.ts
export var y: string;
export function bar(z: boolean): boolean;
```

Property `y` and function `bar` are properties of the single object exported by the library. TPD will wrap the library in a contract for the combined type `{y: string; bar: (z: boolean) => boolean}`.

The implementation of contracts in TPD builds on existing work on gradual typing and in particular the *blame calculus* [39]. In the blame calculus run-time type coercions, or casts, are used to integrate dynamically typed and statically typed regions of code. The work by Findler and Felleisen [8] and Wadler and Findler [39] guides our implementation of function wrappers, and the work by Ahmed et al. [2] guides our implementation of polymorphic contracts.

2.2 Functions

We begin by discussing the implementation of wrappers for primitive values. Wrappers for base types such as `number` and `boolean` can verify that their target conforms to the type by immediately inspecting the value. For example, a wrapper for the type `number` can be implemented as follows:

```
function wrapNumber(value,label) {
  if(typeof value != "number") {
    blame(label);
  }
  return value;
}
```

The function tests that the value supplied has the correct run-time type. If the value is not of type `number` then a call to function `blame` is issued with the appropriate label. The function `blame` does not immediately halt the program, as it would in the blame calculus. This is because we wanted to catch all blame errors in a single execution rather that stop at the first. A call to `blame` will log an error message and proceed with the execution.

Ensuring conformance to a function type cannot be done by inspecting the value that is being wrapped. For functions, a particular context may supply a incorrect argument to the function, or the function may only return an incorrect result for a particular argument.

Similar problems arise in passing any non-primitive type, such as objects. Our approach to function wrappers follows that of Findler and Felleisen [8] and Wadler and Findler [39]. A function wrapper must adhere to the value it supervises and wrap each function application. When applied, a function wrapper will wrap each argument, apply the function, and then wrap the result. We refer the reader to Wadler [38] for a summary of the blame calculus and higher-order blame attribution.

TPD implements function wrappers using the Proxy API [33], where each function wrapper corresponds to a single proxy. A proxy constructor takes a *target* object that is replaced by the proxy, and a *handler* object that contains trap functions to attach to the proxy. Traps intercept a variety of operations including property access, property update, application, and construction. If a trap is not present in the supplied handler then the default behaviour is assumed. Some operations have invariants that must be preserved by the handler passed to the proxy, otherwise a run-time error is thrown[1].

A function wrapper can be implemented using a proxy with an *apply* trap that performs the wrapping. A simplified implementation is defined as follows:

```
1  function wrapFunction(fun,label,type) {
2    if(typeof fun != "function") {
3      blame(label);
4      return fun;
5    }
6    var handler = {
7      apply: function(target, thisArg, argumentsList) {
8        var wrappedArguments = wrap(argumentsList, negate(label),
            type.domain);
9        var result = target.apply(thisArg, wrappedArguments);
10       return wrap(result, label, type.range);
11     }
12   }
13   return new Proxy(fun,handler);
14 }
```

The arguments to the `apply` trap are the target object, the *this* argument for the call, and the list of arguments for the call. The body of the trap wraps the argument list according the domain of `type`, negating the blame label for each argument. The target object is then applied to the wrapped arguments. The handler then wraps the function result using the range of the function type before returning. We distinguish errors in the arguments to a function from errors in the function itself; following Wadler and Findler [39] we call the former negative blame, and the latter positive blame. This is why the blame label on line 8 is negated.

2.3 Polymorphism and Sealing

As well as functions TPD also provides support for polymorphic, or generic, types. Generic types are monitored via *sealing*, which enforces *parametricity*. Parametricity [20, 37] enforces data abstraction: a parametric function acts identically on its arguments irrespective of their types. TPD builds on existing work by Ahmed et al. [2] to ensure parametricity. For instance, if the definition file includes

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/handler/getPrototypeOf#Invariants`

```
declare function sort<X>
  (list: List<X>, comp: (x: X, y: X) => boolean): List<X>
```

then for all types `X`, `sort` accepts a list with elements of type `X` and a comparator function and returns a list of `X`, where the comparator function accepts a pair of elements of type `X` and returns a `boolean`. Parametricity ensures that if we change the representation of `X`, say from dates in one format to dates in another format, then sorting a list in the first representation gives the same answer as sorting in the second representation—so long as the comparator applied to two elements in the first representation gives the same answer as for two corresponding elements in the second representation. It is surprising that a wrapper can ensure this property without examining the code of `sort`! TPD does so by using the types to guide sealing and unsealing of values of variable type. In this case, elements of the argument list are sealed when passed into `sort`, unsealed before being passed to the comparator, and unsealed when returning the final list. Sealing the input list ensures the desired parametricity property because it guarantees that only the comparison function may operate on the elements. Any other operation on the elements will raise blame because the operation will not appropriately unseal the elements.

Some functions do not satisfy parametricity.

```
function weird(x) {
  if (typeof x === "number") return x+1;
  else return x
}
```

It is fine to declare that `weird` accepts a value of any type and returns a value of any type.

```
1 declare function weird(x: any): any  // ok
```

However, it is not correct to declare a generic type that says function `weird` for all types `X` accepts a value of type `X` and returns a value of type `X`.

```
2 declare function weird<X>(x: X): X  // not ok
```

Even though it always returns a value of the same type passed to it, function weird violates parametricity as it does not treat all types the same. We are not able to give `weird` the type as shown on line 2. Indeed, parametricity guarantees that the only total function with the generic type declared for `weird` is the identity function; and the only partial functions with that type are those that always raise an exception.

Type variables play an important role in correctly implementing sealing and they parameterise both the seal and unseal operations. To illustrate their significance consider the following example.

```
declare function badSwap<X,Y>(p: {x: X, y: Y}): {x: Y, y: X};
```

Parametricity tells us that the only total function that satisfies the type attached to the function `badSwap` is the swap function. The function takes an object with fields `x` and `y`, and returns a new object with the same properties but the contents swapped. Suppose `badSwap` is incorrectly implemented as follows.

```
function badSwap(p) {
  return {x: p.x, y: p.y};
}
var z = badSwap({x: true, y: 3});
```

When `badSwap` is wrapped by TPD, accessing property `x` or `y` on argument `p` will return a sealed value. When accessing property `x` or `y` on result `z`, TPD will unseal the contents. Without type variables there is no way to identify which seal corresponds to the argument's `x` field (of type `X`), or `y` field (of type `Y`). Although both fields in `z` contain seals, they have not been swapped as the type requires! To implement polymorphic wrappers in TPD both the seal and unseal operations take a key, a type variable. Every seal is associated with the type variable it was sealed under, and unsealing takes a type variable that must match the seal's type variable, raising blame otherwise. In the example, accessing property `x` on result `z` will unseal the contents using type variable `Y`, but the value stored is sealed under type variable `X`, resulting in blame.

Seals are implemented in TPD using proxies that raise blame on all traps. Every seal is recorded in a WeakMap[2] alongside the type variable under which the object was initially sealed. Wrapping an object in a proxy will return type `"object"` but wrapping a function in a proxy directly will return type `"function"`. To ensure type tests behave uniformly on all seals, all values are wrapped in an additional object prior to sealing. A seal will always return type `"object"` when queried using `typeof`; in the `weird` example, sealing `x` causes the function to behave as the identity. Primitives must be wrapped because they cannot be the direct target of a proxy.

We present an outline implementation of sealing and unsealing.

```
var SEALS = new WeakMap(); // Global Seal Store
function seal(x, tyVar, label) {
  var wrappedVal = {contents: x}; // Mask x's type
  var handler {
    get: function(target, property, receiver) {
      blame(negate(label));
      return x[property];
    },
    set: function(target, property, value, receiver) {
      blame(negate(label));
      return x[property] = value;
    },
    ... // rest of traps omitted for brevity
  }
  var seal = new Proxy(wrappedVal,handler);
  SEALS.set(seal,{v:x, tyVar: tyVar};
  return seal;
}
function unseal(x, tyVar, label) {
  if(SEALS.has(x)) {
    var contents = SEALS.get(x);
    if(contents.tyVar !== tyVar) {
      blame(label); // Sealed under different tyVar
    }
    return contents.v;
  } else {
    blame(label); // Not a sealed value
    return x;
  }
}
```

[2] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/WeakMap

The `seal` function takes a value `x` to be sealed, the type variable associated with the seal, and the blame label associated with the operation. Sealing first wraps the target `x` in an additional object to mask its underlying type. A handler is created that raises blame on all traps, we omit most cases for brevity (marked by $\cdots$). The action of each trap is to first raise blame on the label. After raising blame, the trap forwards the operation to the original value `x`. The handler is used to create a new seal proxy which is then added to the seal store. Every seal in the store is associated with the value it seals and type variable the value was sealed under.

The `unseal` function takes a value `x` to be unsealed, the type variable associated with the unseal, and the blame label associated with the operation. Unsealing a value `x` first determines if `x` is a seal; if the value is not a seal then blame is allocated to `label` and the value is immediately returned. If the value supplied to `unseal` is a seal then the corresponding sealed value and type variable are examined. If the type variable attached to the seal matches the type variable supplied to the function, then the unsealed value is returned. Otherwise, blame is first raised for trying to unseal a value sealed under a different type variable, then the unsealed value is returned.

Polymorphic wrappers in TPD introduce interference by altering a value's type when sealing.

```
function interfere(x) {
  return (typeof x === "number") ? 1 : 0;
}
```

Applying `interfere` to 42 will return 1. Wrapping `interfere` at type `<X>(x: X) => number` and applying the function to 42 will return 0 because the sealed input now has type `"object"`. To satisfy non-interference, applying a type test to a sealed object should cause blame. However, attaining such behaviour is impossible with wrapper code alone; one would need to rewrite the JavaScript interpreter to change the semantics of `typeof`, so that it raised an error if it attempted to find the type of a sealed value.

3 Failure to Conform

TypeScript definition files are often provided by contributors other than the authors of the libraries they describe, and definition and library may fail to conform. TypeScript might be said to fall foul of the adage "Do as I say, not as I do" when what the definition file *says* and what the library *does* fail to conform.

In this section we consider two examples taken from the wild. The lack of conformance between definition and library leads to problems, all of which were uncovered through monitoring with TPD. In each case we state whether the blame was positive (originating in the library) or negative (originating in the client). In case of failure, it may be that either the definition, the implementation, or the client diverges from what was intended. In all of the cases we examined, divergence arose from an error in the definition.

Each example was found by instrumenting a library with TPD and observing output from TPD indicating a type error. All code fragments below are taken from the DefinitelyTyped repository and the JavaScript library in question. Some blank lines have been deleted, and elisions are indicated by ellipses "$\cdots$".

```
export class Valve {
  ...
  check(obj: any, options: ICheckOptions, callback: (err: any, cleaned: any) => void): void;
  check(obj: any, callback: (err: any, cleaned: any) => void): void;
}
```

Figure 1 swiz - Definition

```
Valve.prototype.check = function(_obj, options, callback) {
  ...
  if (!this.schema) {
    callback('no schema specified');
    return;
  }
  if (options.strict) {
    for (key in obj) {
      if (obj.hasOwnProperty(key) && !this.schema.hasOwnProperty(key)) {
        callback({'key': key, 'message': 'This key is not allowed'});
        return;
      }
    }
  }
  checkSchema(obj, this.schema, [], false, this.baton, function(err, cleaned) {
    if (err) {
      callback(err);
      return;
    }
    if (finalValidator) {
      finalValidator(cleaned, function(err, finalCleaned) {
        if (err instanceof Error) {
          throw new Error('err argument must be a swiz error object')
        }
        callback(err, finalCleaned);
      });
    }
    else {
      callback(err, cleaned);
    }
  });
};
```

Figure 2 swiz - Library

```
var validity = swiz.defToValve(defs), v = new Valve(validity.
    Server);
// Valid payload
var goodServer = {
   ...
  'ipaddress' : '42.24.42.24'
};
v.check(goodServer, function(err, cleaned) {
  console.log('Success:');
  console.log(cleaned);
});
// Invalid payload
var badServer = {
   ...
  'ipaddress' : '127.0'
};
v.check(badServer, function(err, cleaned) {
  console.log('Error - invalid ip:');
  console.log(err);
});
```

Figure 3 `swiz` - Client

3.1 Higher-order Positive Blame

Here is an example of positive blame, where the library fails to conform to the definition.

▶ **Example 1** (`swiz`). The `swiz` library is a framework for object serialisation and validation. The library was written by Rackspace [18], the accompanying definition was written by Goddard [9], and the client was written by Rackspace [18].

Definition Figure 1 shows the definition for class `Valve`; we draw focus to the overloaded function `check`. The first overload accepts three arguments, one of type `any`, one of type `ICheckOptions`, and one of function type that accepts two arguments of type `any` and returns nothing. The second overload only accepts two arguments, the first and third of the prior overload. Both overloads return `void`.

Library Figure 2 shows the implementation for the function `check`. The function declares three arguments: `_obj`, `options`, and `callback`. We draw the reader's attention to the uses of `callback` within the definition. On lines 4, 10, and 17 the function is applied to one argument, on lines 25 and 29 the function is applied to two arguments. The three applications of the callback with a single argument are inconsistent with the definition file that states the callback takes two arguments.

Client Figure 3 shows client code from a tutorial provided by the library authors. In the example there are two applications of the `check` function, the first passes `goodServer`, and the second passes `badServer`. As to be expected, the authors understand their library and the callback in the second application of `check` does not utilise its second argument. This is because a bad server was passed.

The callback may use the second parameter in such a way that when its value is `undefined`, erroneous behaviour occurs. Such an example is reading or writing a property

```
declare module "asciify" {
  function asciify(text: string, callback: AsciifyCallback): void;
  function asciify(text: string, options: string, callback:
      AsciifyCallback): void;
  function asciify(text: string, options: AsciifyOptions, callback
      : AsciifyCallback): void;
   ...
}
```

Figure 4 `asciify` - Definition

```
module.exports = function (text, opts, callback) {
  // Ensure text is a string
  text = text + '';
  if (typeof opts === 'function') {
    callback = opts;
    opts = null;
  }
          ...
}
```

Figure 5 `asciify` - Library

```
asciify(138, 'pyramid', function(err, res){
   ...
  }
);
asciify(false, 'pyramid', function(err, res){
   ...
  }
);
```

Figure 6 `asciify` - Client

of an undefined object. TPD detects the mismatch and allocates positive blame, indicating that the source of the mismatch was the in library rather than the client. This particular case is higher-order positive blame and occurs when a wrapped function receives an argument, itself a function, and uses that function argument incorrectly. In this example, it does not provide sufficient arguments to the callback function.

3.2 Negative Blame

Here is an example of negative blame, where the client fails to conform to the definition.

▶ **Example 2** (`asciify`)**.** The package `asciify` is a library and command-line tool for generating ASCII art. The library was written by Evans and Shaw [6], the accompanying definition was written by Norbauer [17], and the client was written by Evans and Shaw [6].

Definition Figure 4 shows an excerpt from the definition file. We focus on the function `asciify`, that has three overloads. The first overload accepts two arguments, a string to be transformed and a callback. The second overload accepts three arguments, the additional

```
declare module 'gulp-if' {
  import fs = require('fs');
  import vinyl = require('vinyl');

  interface GulpIf {
    ...
    (condition: boolean, stream: NodeJS.ReadWriteStream,
       elseStream?: NodeJS.ReadWriteStream): NodeJS.
       ReadWriteStream;
    ...
  }
  var gulpIf: GulpIf;
  export = gulpIf;
}
```

Figure 7 `gulp-if` - Definition

argument is `options` of type `string`. The third overload also accepts three arguments, but `options` is of type `AsciifyOptions`.

Library Figure 5 shows part of the implementation for the function `asciify`. The `text` argument is coerced to a string by appending the empty string, exploiting JavaScript's implicit type coercions. If `opts` is of function type the first overload is assumed, `callback` is then updated to have the value of `opts`.

Client Figure 6 shows an extract from the unit tests accompanying the library. The first test case applies `asciify` to the number 138, the second applies to the boolean `false`. TPD interprets overloaded functions as having an intersection type; Keil and Thiemann [13] determined that a context (client) satisfies an intersection type if it respects at least one constituent of the intersection. The first overload of the function is violated as three arguments are supplied, rather than two. The second and third overloads are violated as the `text` argument does not have type `string`. TPD detects the mismatch and allocates negative blame, indicating that the source of the mismatch was the in client rather than the library. This mismatch demonstrates that the type of the function is too conservative: the function argument `text` may be of `any` type, rather than `string`.

4 Examples of Interference

The alteration of object identity through the use of proxies has been considered in existing work [11, 12]. We are not aware of any work that extensively presents interference caused by using proxies as dynamic seals. In this section we present a range of examples taken from the wild that demonstrate how proxies cause violations of non-interference, reinforcing the claim that proxy interference is a real problem. Each example was found by instrumenting a library with TPD and observing a unit test fail, when previously it did not. All code fragments below are taken from the DefinitelyTyped repository and the JavaScript library in question. Some blank lines have been deleted, and elisions are indicated by ellipses "$\cdots$".

```
'use strict';

var match = require('gulp-match');
var ternaryStream = require('ternary-stream');
var through2 = require('through2');

module.exports = function (condition, trueChild, falseChild, minimatchOptions) {
  if (!trueChild) {
    throw new Error('gulp-if: child action is required');
  }

  if (typeof condition === 'boolean') {
    // no need to evaluate the condition for each file
    // other benefit is it never loads the other stream
    return condition ? trueChild : (falseChild || through2.obj());
  }

  function classifier (file) {
    return !!match(file, condition, minimatchOptions);
  }
  return ternaryStream(classifier, trueChild, falseChild);
};
```

Figure 8 `gulp-if` - Library

4.1 Proxy Identity

Here is an example of interference caused by a proxy changing object identity, where a wrapped and unwrapped version of the same object are compared.

▶ **Example 3** (`gulp-if`). The `gulp-if` library is a plugin for the streaming build system `gulp`. The definition was written by Skeen and Asana [27], the library and client were written by Richardson [22].

Definition Figure 7 shows an extract from the definition file for `gulp-if`. The type of the exported library is defined by interface `GulpIf`; an `interface` describes an object. Objects in TypeScript may have properties, methods, and be directly callable like functions. If an interface only contains function signatures, we can interpret the interface as a function type. Line 8 elides overloaded call signatures that do not feature in our example code. Function signatures are of the form `(args): type`, where `args` is a possibly empty list of `name: type` pairs. Placing `?` after a field or argument name indicates that it is optional. This function signature accepts a condition of type `boolean`, a stream of type `NodeJS.ReadWriteStream`, and optionally another stream of the same type, and the function returns a stream of the same type.

Library Figure 8 shows the entire implementation of the library. The library exports a single function that returns a new stream based on the truth value of `condition` passed to the function. When the condition is satisfied the stream passed as `trueChild` is returned. If the condition is not satisfied then if an *else* stream was passed as `falseChild`, that stream is returned, otherwise a default stream is returned instead.

```
 ...
describe('when given a boolean,', function() {
  var tempFile = './temp.txt';
  var tempFileContent = 'A test generated this file and it is safe
      to delete';

  it('should call the function when passed truthy', function(done)
      {
    // Arrange
    var condition = true;
    var called = 0;
    var fakeFile = {
      path: tempFile,
      contents: new Buffer(tempFileContent)
    };

    var s = gulpif(condition, through.obj(function (file, enc, cb)
        {
      // Test that file got passed through
      (file === fakeFile).should.equal(true);

      called++;
      this.push(file);
      cb();
    }));

    // Assert
    s.once('finish', function(){

      // Test that command executed
      called.should.equal(1);
      done();
    });
    // Act
    s.write(fakeFile);
    s.end();
  });
   ...
}
```

Figure 9 gulp-if - Client

Client Figure 9 shows an extract from the unit tests accompanying the `gulp-if` library. In the example we present the test that exhibited interference caused by proxies changing object identity. The `describe` function indicates a set of tests and the `it` function indicates a particular test case. On line 15 the library (bound to `gulpif`) is used to create a new stream `s`. This particular test case checks that the stream passed as the `trueChild` argument correctly receives the data when the condition is true. The *true* stream is created using the function `through.obj` that creates a basic stream using a transform function supplied as an argument; the transformer function is defined on lines 16–21. A transform function receives as argument the piped data, in this case a file, an encoding string, and a callback to execute when done. For this test, the transform function asserts that the correct file was supplied to the stream using an equality test, and then increments a counter to indicate it was evaluated. Lines 25–30 add a finalising handler to stream `s` that asserts the *true* stream was piped to by checking that `counter` has be incremented once. Line 32 initiates the test by writing the file `fakeFile` to the stream.

Before and After Before wrapping the expected outcome is that the assertions on lines 17 and 28 should hold. The correct file should be piped through the stream (line 17), and the final callback should be fired (line 28).

After wrapping with TPD the first assertion (line 17) fails. To understand why, first consider the type of the stream returned from the call to the library. From the definition file the library function returns a `NodeJS.ReadWriteStream`, this has a method with the following signature [3].

```
write(buffer: Buffer, cb?: Function): boolean;
```

Recall that TPD wraps both the argument and result to every function call. When the instrumented library returns a result of type `NodeJS.ReadWriteStream`, the function result will be wrapped by TPD according to the type `NodeJS.ReadWriteStream`. Consequently, when the test case initiates the test using the `write` method, TPD will wrap the function argument using the type `Buffer`. As a `Buffer` is an object type a proxy is used in place, thus giving the argument a new identity. This proxy is passed to the transform function as parameter `file`, and when compared for identity with `fakeFile`, returns false when previously the comparison returned true.

To address this particular example the equality test must be explicitly replaced with a proxy-aware version. Another alternative is to use transparent proxies [11] that retain the identity of the object they wrap. Membranes [12, 34] will not work because only one object in the comparison is wrapped in a proxy; membranes only work when comparing objects on the same side of the membrane.

4.2 Dynamic Sealing

Here is an example of interference caused by a proxy used as dynamic seal, where the type of a sealed object is changed.

▶ **Example 4** (`clone`). The `clone` library provides deep cloning for objects, arrays, and other JavaScript data types. The definition was written by Simpson [26], the library and client were written by Vorbach [36].

[3] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/node/index.d.ts`

```
declare module "clone" {
  ...
  function clone<T>(val: T, circular?: boolean, depth?: number): T

  module clone {
    function clonePrototype<T>(obj: T): T;
  }
  export = clone
}
```

Figure 10 clone - Definition

```
...
function clone(parent, circular, depth, prototype,
    includeNonEnumerable) {
  if (typeof circular === 'object') {
    depth = circular.depth;
    prototype = circular.prototype;
    includeNonEnumerable = circular.includeNonEnumerable;
    circular = circular.circular;
  }
  ...
  if (typeof circular == 'undefined')
  circular = true;

  if (typeof depth == 'undefined')
  depth = Infinity;

  // recurse this function so we don't reset allParents and
      allChildren
  function _clone(parent, depth) {
    // cloning null always returns null
    if (parent === null)
    return null;

    if (depth === 0)
    return parent;

    var child;
    var proto;
    if (typeof parent != 'object') {
      return parent;
    }
    ...
  }
  return _clone(parent, depth);
}
...
```

Figure 11 clone - Library

```
exports["clone number"] = function (test) {
  test.expect(5); // how many tests?

  var a = 0;
  test.strictEqual(clone(a), a);
  a = 1;
  test.strictEqual(clone(a), a);
  a = -1000;
  test.strictEqual(clone(a), a);
  a = 3.1415927;
  test.strictEqual(clone(a), a);
  a = -3.1415927;
  test.strictEqual(clone(a), a);

  test.done();
};
```

Figure 12 `clone` - Client

Definition Figure 10 shows the definition file for `clone`. The library exports the generic `clone` function that accepts a value to be cloned of type `T`, an optional optimisation parameter `circular` of type `boolean`, and an optional `depth` parameter of type `number`, and returns a value of type `T`. Readers familiar with generics and parametricity will realise that a clone function cannot possibly have this type as it violates parametricity! However, we would hope that monitoring the function would alert the programmer to this error by raising blame, rather than violating non-interference.

Library Figure 11 shows an extract from the implementation of the `clone` library. We have removed a significant amount of code to focus on the sections relevant to our example. All redactions are indicated by "$\cdots$". The function `clone` defines an inner recursive function `_clone` that does most of the heavy-lifting. We draw the reader's attention to the initial segment to the `_clone` function, ranging from line 18 to line 29. When the function is passed a `null` pointer it returns `null`. When the (optional) cloning depth has been reached the current pointer is returned. If the value to be cloned is not an object, for example a number, the value is immediately returned. A number is trivially a clone of itself as numbers, and other primitives, have no notion of identity.

Client Figure 12 shows an extract from the unit tests accompanying the `clone` library. In the example we present the test that exhibited interference caused by dynamic sealing. Specifically, this example shows a violation of non-interference resulting from a seal changing the type of the value it wraps. The `exports` object acts as a map that associates string test descriptions to test functions. A test function accepts a single argument `test` that acts as the testing API, offering functions such as `expect`, `strictEqual`, and `done`. This test function expects five tests, each cloning a primitive number and asserting that the result is the same.

Before and After Before wrapping the expected outcome is that each call to the `clone` function should return the argument without change.

After wrapping with TPD every call to the `clone` function will seal the argument because the argument has generic type `T`. Recall that before sealing, every value is wrapped in an object to mask the value's type, and to fix the seal's type to `object`. When each number in the test is passed to clone, the number is first wrapped in an object, sealed, and then passed to the `clone` function. The type test on line 27, a type test that originally returned `number`, will now return `object`. As the condition is not met the function will fail to return immediately, instead it will proceed to clone the seal. The function result will be a clone of the sealed number rather than the number itself.

To address this particular example a proxy must be able to trap the `typeof` operation and throw an error when the type of a seal is queried. This is not possible in JavaScript, so one would have to replace all `typeof` operations with proxy-aware type tests.

5 Evaluation

We used the DefinitelyTyped [5] repository as a corpus of libraries and definitions to evaluate our gradual typing tool TPD. We believe there are two important conclusions from our experiments. First, TypeScript definitions are prone to error. Second, interference caused by proxies is a problem in practice. The artifact containing the libraries, definition files, and source code is available on the Dagstuhl Research Online Publication Server (DROPS). The source code for the tool is also available online[4].

5.1 Method

We selected the libraries that targeted the Node.js run-time, that could be installed and executed without manual configuration, and that had a set of unit tests accompanying the library source code that all passed. Libraries were wrapped automatically using TPD and their unit tests executed. We recorded failures of the library or client to conform to the definition, classifying the error. In addition, we recorded violations of non-interference. As all libraries passed their tests *prior* to wrapping, we attributed any failing tests *after* wrapping as interference. In total we tested 122 libraries, and all libraries are listed in the appendix. Testing was conducted using a MacBook Pro with a 2.6 GHz i5 and 8GB RAM.

5.2 Failures to Conform

Table 1 shows failures to conform detected by TPD. We distinguish four error kinds and give the blame polarity of the error. In total there were 179 distinct errors found in 62 libraries.

Value Type Value type errors occur when a value does not have the expected run-time type tag, such as `number` or `string`, tested using the built-in `typeof` operator.

Function Arity If the arguments passed to a function are too many, or too few, it is classed as an arity error. Typically this was due to definition authors not understanding which arguments should be optional. The majority of errors were the fault of the client, which is expected given that first-order functions are more prevalent than higher-order functions.

Void Return Type When a function returns a value but its type states it returns `void`, we class this as a void return type error. These errors were typically caused by incorrectly considering a synchronous function as asynchronous.

[4] `https://github.com/jack-williams/tpd`

Table 1 Classification of Failures to Conform.

	Blame	
Error Kind	(+) Library	(−) Client
Value Type	47	47
Function Arity	23	43
Void Return Type	14	2
Parametricity	3	0
Distinct Errors	87	92
Distinct Libraries	40	48

Table 2 Classification of Interference.

Cause of Interference		
Proxy Identity	TI	7
	TII	5
Sealing		5
Reflection		4
Proxy Implementation		2
Distinct Libraries		22

Parametricity Parametricity errors were due to functions being incorrectly typed as parametric. There are two ways to elicit parametric blame: returning a value that is not a seal, or tampering with a sealed value. A common example of an incorrectly typed function would be one that takes an object and, using reflection, creates a new object with the same property mappings. The use of reflection to access properties amounts to tampering, raising blame.

5.3 Violations of Non-interference

Table 2 shows the violations of non-interference observed. We distinguish four causes and give their frequency. After instrumenting the code, 22 libraries violated non-interference.

Proxy Identity We witnessed 12 libraries that failed tests due to proxies changing the identity of objects. Our classification adopts a similar dichotomy of identity failure as Keil et al. [11]. The first (TI) compares a wrapped and unwrapped version of the same object, of which there were seven. Avoiding interference in these cases requires rewriting all equality tests to proxy-aware alternatives, or providing transparent proxies that do not change object identity [11]. The second (TII) compares different proxies of the same object, our experiments found five cases. This problem may be addressed with identity preserving membranes, where identical objects passing through the membrane are wrapped using the same proxy, thus preserving equality inside the membrane [12, 34].

Sealing There were five libraries that presented interference caused by the use of sealing to enforce parametricity. We implemented seals using proxies that raised blame on all traps. The Proxy API only permits objects to be sealed; to seal a primitive value it must be wrapped in an object, which changes its type. As discussed earlier, we are unable to restore non-interference because the `typeof` operator cannot be trapped.

Reflection We observed tests failing due to reflection in four libraries. TPD adds wrapper code to a library, so packages that inspect their code as part of their testing process, for example linting, will observe differences when wrapped. In particular, additional libraries required by TPD would appear in the global namespace and be considered unexpected by tests that inspect the global object's properties

Proxy Implementation There were two libraries that exhibited failing tests due to issues with the underlying proxy implementation. The Proxy API is not mature, and some components of the run-time are not proxy-aware. Parts of the underlying run-time perform dynamic type checking, and if not proxy-aware, will throw an error when supplied a proxy.

5.4 Comparative Techniques

TypeScript TPD is a tool that uses gradual typing to enforce library and client conformance to a definition. TypeScript TPD is not a tool specifically designed to detect erroneous definition files. Our method of evaluated gradual typing using DefinitelyTyped allows us to detect errors in definitions. Other tools such as *TSCheck*, *TSInfer*, and *TSEvolve* [7, 14] are designed to detect errors in definitions and support the construction of new definitions. These tools have the advantage of not requiring test code to detect errors, and do not introduce interference as they use static analysis. We believe there is a place for both approaches. There are clear benefits to writing and debugging definitions using these static tools. However, even when library and definition can be guaranteed to conform, an unsound TypeScript client may deviate from the definition. TPD enables a programmer to enforce client conformance in this case.

5.5 Performance

We measured the effect of using TPD on test completion time by recording how long it took to execute the entire test suite. Such a metric does not give a precise account of the cost of run-time checks because a test suite may include other unrelated stages. Evaluating the exact cost of wrappers would require understanding of each library test suite as well as manual instrumentation of the code. Our chosen evaluation method is still relevant because executing the test suite is a common step in development, significantly slowing this down would be a considerable hindrance to adoption. Amongst the libraries that did not exhibit interference, wrapping introduced a 38% increase in testing time on average.

Rastogi et al. [19] developed a modified version of TypeScript, Safe TypeScript, that inserts run-time checks to enforce safety. They place an emphasis on performance, where we do not implement any wrapper optimisation. Predictably, there system incurs a smaller overhead of 15% on the run-time.

5.6 Threats to Validity

We identify five threats to the validity of our result. First, our result may be perceived as adding nothing new to the existing work by Feldthaus and Møller [7] that shows TypeScript definitions are prone to error. We believe there are two new components to our results. Where Feldthaus and Møller test the largest ten libraries, we test a range of sizes. Our results show that errors in definitions are not exclusive to the largest and most complex libraries. Where Feldthaus and Møller only analyse the library implementation, we also monitor clients (test code). Half of the errors we detected were the fault of the client.

Second, our approach of detecting interference is incomplete. It is possible that TPD violates non-interference but a library still passes all its unit tests. As a consequence, our results may not account for all occurrences of interference caused by proxies. We claim that the frequency of proxy interference observed is intolerable in practice; failing to detect additional cases does not weaken this claim.

Third, the client code we use to exercise the library is the library's corresponding unit test suite. The frequency of interference observed when running unit tests may not be representative of real library usage. In particular, unit tests may contain a higher number of equality tests than real code. Even if unit tests do elicit more violations of non-interference than typical code, we believe that running unit tests are an important component of library design; altering test behaviour is a significant problem for practitioners.

Fourth, the experiments were only conducted on a small proportion of the Definitely-Typed repository. We believe our criteria for selecting libraries is not biased towards exhibiting greater interference, but we cannot categorically claim that our sample is representative of all JavaScript libraries in the repository.

Fifth, unit test coverage may not be high enough to capture all common interactions with a library. As a result, we may fail to detect errors in conformance, or we may fail to observe cases of interference that may occur in practice. We believe that the errors and interference we report is significant; failing to detect additional cases does not diminish this.

6 Design Alternatives and Solutions to Interference

In this section we discuss design alternatives and solutions to the problem of interference. Keil et al. [11] survey different approaches to proxy implementation and equality. We supplement their summary using our experiences of TPD.

6.1 Rewriting

The systematic replacement of equality operations with proxy-aware versions would remedy interference associated with identity. All proxies are stored in a `WeakMap` to which the new equality operation has access. A custom equality operation allows the choice between making proxies appear opaque or transparent. The same technique can be applied to the `typeof` operation to remove interference caused by dynamic sealing. Replacing equality and type tests with proxy-aware alternatives allows blame to be raised upon the application of an operation to a seal; currently equality and type tests cannot be trapped. An approach that uses rewriting must ensure that it correctly handles dynamically loaded code and use of `eval` [11].

6.2 Transparent Proxies

An alternative to using opaque proxies as provided by JavaScript currently would be to use transparent proxies [11]. A transparent proxy forwards identity checks to the target object so wrapping an object does not alter its identity. One design aspect of transparent proxies is the use of *realms* [11]. The realm of a proxy is the context that constructed the proxy,

```
var prim_get = Reflect.get; // Reference to unpatched API
var typings  = new WeakMap();
function wrapObj(obj,label,type) {
  typings.set(obj,{label: label, type: type});
  return obj;
}
// Patch Reflect API
Reflect.get = function(target,property,receiver) {
  var value = prim_get(target,property,receiver);
  if(typings.has(target)) {
    var label = typings.get(target).label;
    var type  = typings.get(target).type;
    var prop_type = type[property] ? type[property] : Type.Any;
    return wrap(value,label,prop_type);
  } else {
    return value;
  }
}
```

Figure 13 Modified Reflect API

represented using a token. Inside a proxy's realm the proxy has a distinct identity rather than assuming the identity of the target object. Realms are essential to the implementation of dynamic seals. Two seals of the same object sealed under different type variables must be distinguishable, otherwise unsealing cannot be correctly implemented. Unsealing would take place inside the realm of seal proxies, where each seal has its own identity. Outside the realm—in client or library code—seals will inherit the identity of their target object, as desired.

6.3 Reflection

An alternative to using proxies is to use the Reflect API [5]. The Reflect interface is intentionally identical to the Proxy interface, so a comparison is natural.

Wrapping for base types remains unchanged, a type test is performed on the value. When wrapping an object a typing for the object is created rather than a proxy. A typing associates the object with its type and label in a `WeakMap`. The Reflect API is patched to use the typings to monitor operations performed through the API. When accessing properties on an object using reflection, the patched operation uses any typings to wrap the corresponding property. If no typing exists then the operation is handled by the unpatched operation. Figure 13 shows how to patch property access in this style. From the code it can be seen that wrapping an object returns the same object, therefore preserving object identity. Patching the Reflect API is an approach that requires native object operations (`foo.x`) to be rewritten to use the Reflect API (`Reflect.get(foo,"x")`). This approach is semantically different to using proxies. A proxy based approach associates a type to a particular pointer using a wrapper. A reflect based approach associates a type to the value on the heap. To illustrate, take the following example.

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect

```
var x = {a: 3}
var y = wrap(x,Type.obj({a: Type.Bool}));
Reflect.get(y,"a"); // Proxy -> Blame    | Reflect -> Blame
Reflect.get(x,"a"); // Proxy -> No Blame | Reflect -> Blame
```

A proxy based solution will raise blame when accessing the property on `y`, but not `x`. A reflection based solution will raise blame for both operations. The second operation will not raise blame when using proxies because the reference `x` is not wrapped in a proxy and therefore the look-up performs no type checking. The reflection based solution raises blame because typings are assigned to the object's underlying identity, rather than creating a new pointer to a proxy. The reflection based design is closer in nature to the work on monotonic references by Siek et al. [25], and may therefore benefit from the performance advantages that calculus provides.

6.4 Dynamic Sealing

Implementing sound and non-interfering wrappers for parametric polymorphism in JavaScript is a challenging problem—with no clear panacea. Trapping the `typeof` operation is the immediate remedy to the problems experienced by TPD. The behaviour could be constrained to only allow a trap to report the same type as the target or throw an error, which would be sufficient for TPD. Even then, the current language specification does not include the possibility of throwing an error when using `typeof`, so permitting this may be too problematic in practice.

An implementer of sealing is forced to chose between soundness and non-interference when `typeof` cannot be trapped. According to parametricity, the function `fakeConst` with type `<T>(x: T) => number` must be the constant number function; the implementation given below is *not* the constant number function.

```
function fakeConst<T>(x: T): number {
  if(typeof x === "number") { return 1; }
  else { return 2 };
}
```

A choice must be made: enforce parametricity by masking the argument's type, necessarily interfering, or satisfy non-interference by revealing the seal target's type, violating parametricity. TPD enforces parametricity so the example will always return 2 because the type of a seal is always `object`. If TPD were to enforce non-interference instead, then the violation of parametricity is not reported as a type error to the programmer because the type test cannot be trapped. If a proxy could alter behaviour of `typeof` then blame could be triggered when `typeof` is invoked, ensuring both soundness and non-interference.

A programmer may favour unsound monitoring over wrappers that change the semantics of their program. This is a problem because primitive values cannot be sealed without being wrapped in an object first, introducing interference. One approach to this problem is to use virtual values [3]. A virtual value is a value that that supports behavioural modification, much like a proxy. Applying a primitive operation to a virtual value will invoke a trap, defined by the programmer. Virtual values would allow the sealing of primitives directly. In this situation we are not *forced* to change the type of a seal. Therefore it would be possible to implement unsound, but non-interfering seals, by allowing a seal to retain the type of the target it encapsulates.

7 Related Work

Optional and Gradual Typing Mezzetti et al. [15] investigate unsoundness caused by optional typing in Dart, and whether the unsoundness in the type system can be justified by increased practicality to programmers. They conclude that most cases of unsoundness can be justified. A notable example that they argue is unjustified is bivariant function subtyping.

Takikawa et al. [30] study the performance of gradual typing and give a damning indictment. Should other language implementations see similar performance results then the future of gradual typing may be cut short. Their evaluation of gradual typing is conducted using Typed Racket, a language with arguably the most extensive support for gradual typing, including transparent proxies [28]. The study they conduct is the first systematic approach to monitoring the performance of gradual typing.

Vitousek et al. [35] implement and evaluate a gradually typed variant of Python, *Reticulated Python*. They acknowledge the problem of proxies changing identity and type but do not provided data recording the scale of the problem. Their implementation uses three mechanisms for wrapping mutable objects: guarded (proxies), transient, and monotonic. Transient and monotonic checks do not alter object identity, unlike the guarded semantics.

Guha et al. [10] present the design of parametric polymorphic contracts in Scheme and JavaScript. Their system requires polymorphic contracts to be instantiated with a type, where our system implicitly instantiates all contracts with the dynamic type, following Ahmed et al. [1]. Guha et al. [10] implement seals using standard objects rather than proxies, and their seals do not raise negative blame when tampered with.

JavaScript and TypeScript The only existing works on checking conformance between JavaScript libraries and their TypeScript definition files is that of Feldthaus and Møller [7] and Kristensen and Møller [14]. Unlike our system, Feldthaus and Møller [7] perform static analysis rather than dynamic monitoring. They combine heap snap-shot analysis and lightweight static analysis of function definitions. Both techniques are unsound but carry the advantage of incurring no run-time cost, and not causing interference. Their tool highlights a large number of mismatches, corroborating our outcome; TypeScript definition files are prone to errors and there is a real need for machine verified documentation. Kristensen and Møller [14] provide the tools *TSInfer* and *TSEvolve*, tools that help construct new definitions and maintain them. Comparing the dynamic method of checking library conformance with other static techniques is future work.

Work on providing a static type system for JavaScript was conducted by Thiemann [31] who used singleton types and first class record labels to capture the semantics of the prototype based object system of JavaScript.

Swamy et al. [29] developed TS*, a gradually-typed core of JavaScript. Their compiler inserts checks that use run-time type information (RTTI) to ensure type safety. The type safety guarantees that their system provides hold even under arbitrary interaction with JavaScript programs, programs that may dynamically load code. Memory isolation prevents the mutation of TS* objects by untrusted code. Safe TypeScript [19] is another compiler for TypeScript that guarantees type safety. Unlike Swamy et al. [29], they focus on scale and as a result their system is faster but more permissive. Their system employs *differential subtyping* to determine the minimum amount of RTTI a value must carry. Safe TS and our tool TPD differ in some of the errors they detect. For example, Safe TS treats classes nominally, while TPD does not. TPD allows the (implicit) cast from generic types to `any`, via sealing, while Safe TS does not.

Richards et al. [21] developed StrongScript, a variant of TypeScript that offers dynamic, optional, and concrete types. Dynamic types are not statically checked and may fail at run-time. Optional types can refer to any value, but operations on optional types are statically checked. Run-time checks are used to ensure optionally typed values conform to the interface. Concrete types are statically checked in full, and introduce no run-time checks. StrongScript satisfies *trace preservation*, related to non-interference. Adding optional types to a dynamically typed program, introducing run-time checks, should not break the run-time behaviour of the program. If the dynamically typed program terminates to a value, then the same program with optional types should also terminate to that value.

Proxies and Interference Trustworthy proxies within JavaScript was explored by Van Cutsem and Miller [34]. They proposed a proxy API that that retains object invariants through the use of membranes. Their work addressed the issue of frozen objects crossing over the membrane. When referencing a property of a frozen object inside the membrane the result is transitively wrapped in a new proxy. This breaks the frozen invariant whereby the returned value must be identical to the underlying field. By using a *shadow target* to store wrapped frozen properties the value returned successfully passes the invariant check.

Keil et al. [11] gave insight into the design of transparent proxies for JavaScript. Their work presents solutions to the identity issue caused by proxies, as well as implementing an extension to the SpiderMonkey engine enabling support for transparent proxies.

8 Conclusion

Gradual typing integrates statically and dynamically typed code. While there are several theoretical frameworks for gradual typing that ensure desirable properties such as conformance and non-interference, adopting these to existing languages such as JavaScript poses many difficulties. TPD is an application of gradual typing that wraps JavaScript libraries according to their TypeScript definition file. We implement wrappers using proxies, a facility in JavaScript that lets us attach type checking code without having to apply rewriting to the library. Proxies change the identity of their target and so my cause interference. We evaluate whether, in practice, this problem is prevalent enough to rule out opaque proxies as an implementation technique for gradual typing. Our results show that proxies cause interference in an intolerable number of cases, either by changing the identity of their target, or by changing the type of their target when used as a seal.

There is cause for some optimism. TPD detected a significant number of mismatches between libraries and their definition files. A solution to error prone definitions files is needed and gradual typing may be the answer. The nature of TypeScript means that even if library and definition conform, errors may still come from a client. Our work has shown the value of monitoring a JavaScript library and client to ensure they correspond to the TypeScript definition file (conformance), and that by preventing proxies from redefining equality or `typeof` the current definition of JavaScript makes it impossible to monitor for conformance without changing the semantics of the program (non-interference). If we are to have the benefits of both conformance and non-interference, JavaScript will need to evolve, something which it has demonstrated it is capable of doing.

Acknowledgements This work was supported by Microsoft Research through its PhD Scholarship Programme, and by EPSRC grants EP/K034413/1 and EP/L01503X/1. The authors wish to thank anonymous reviewers of the paper and artifact.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2011. doi:10.1145/1925844.1926409.

2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for Free for Free: Parametricity, With and Without Types. In *ACM International Conference on Functional Programming (ICFP)*, 2017.

3 Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual Values for Language Extension. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2076021.2048136.

4 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.

5 DefinitelyTyped. DefinitelyTyped repository. https://github.com/DefinitelyTyped/DefinitelyTyped, May 2017.

6 Oli Evans and Alan Shaw. asciify. https://github.com/olizilla/asciify, May 2017.

7 Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2014. doi:10.1145/2714064.2660215.

8 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *ACM International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/583852.581484.

9 Jeff Goddard. swiz.d.ts. https://github.com/borisyankov/DefinitelyTyped/tree/master/types/swiz/index.d.ts, May 2017.

10 Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS)*, 2007. doi:10.1145/1297081.1297089.

11 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent Object Proxies in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.149.

12 Matthias Keil and Peter Thiemann. On the proxy identity crisis. *CoRR*, 2013.

13 Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *ACM International Conference on Functional Programming (ICFP)*, 2015. doi:10.1145/2858949.2784737.

14 Erik Krogh Kristensen and Anders Møller. Inference and evolution of typescript declaration files. In *Proc. 20th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2017. doi:10.1007/978-3-662-54494-5_6.

15 Gianluca Mezzetti, Anders Møller, and Fabio Strocco. Type Unsoundness in Practice: An Empirical Study of Dart. In *Dynamic Languages Symposium (DLS)*, 2016. doi:10.1145/2989225.2989227.

16 Microsoft. TypeScript language specification. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md, January 2016.

17 Alan Norbauer. asciify.d.ts. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/asciify/index.d.ts, May 2016.

18 Rackspace. node-swiz. https://github.com/racker/node-swiz, May 2017.

19 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2775051.2676971.

20 John Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing*, 1983.

21 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. `doi:10.4230/LIPIcs.ECOOP.2015.76`.

22 Rob Richardson. gulp-if. `https://github.com/robrich/gulp-if`, May 2017.

23 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007. `doi:10.1007/978-3-540-73589-2_2`.

24 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*, 2006.

25 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*, 2015. `doi:10.1007/978-3-662-46669-8_18`.

26 Kieran Simpson. clone.d.ts. `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/clone/index.d.ts`, May 2017.

27 Joe Skeen and Asana. gulp-if.d.ts. `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/gulp-if/index.d.ts`, May 2017.

28 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2012. `doi:10.1145/2384616.2384685`.

29 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual Typing Embedded Securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2014. `doi:10.1145/2578855.2535889`.

30 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016. `doi:10.1145/2837614.2837630`.

31 Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *European Symposium on Programming (ESOP)*, 2005. `doi:10.1007/978-3-540-31987-0_28`.

32 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*, 2006. `doi:10.1145/1176617.1176755`.

33 Tom Van Cutsem and Mark S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium (DLS)*, 2010. `doi:10.1145/1899661.1869638`.

34 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013. `doi:10.1007/978-3-642-39038-8_7`.

35 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Dynamic Languages Symposium (DLS)*, 2014. `doi:10.1145/2775052.2661101`.

36 Paul Vorbach. node-clone. `https://github.com/pvorb/node-clone`, May 2017.

37 Philip Wadler. Theorems for free! In *ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1989. `doi:10.1145/99370.99404`.

38 Philip Wadler. A complement to blame. In *Summit on Advances in Programming Languages (SNAPL)*, 2015. `doi:10.4230/LIPIcs.SNAPL.2015.309`.

39 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, 2009. `doi:10.1007/978-3-642-00590-9_1`.

A List of Libraries Tested

Library (Testing time percentage increase after wrapping)

ansicolors (9.58)	hapi (41.32)	parsimmon (25.12)
any-db (28.28)	highland (x)	promptly (15.67)
asciify (2.81)	htmlparser2 (83.95)	protobufjs (37.52)
aspnet-identity-pw (11.23	http-string-parser (3.48)	radius (20.78)
assert (35.21)	inflection (x)	readdir-stream (10.78)
assertion-error (6.83)	insight (0.44)	rimraf (4.37)
async (x)	ip (6.70)	sanitize-html (5.61)
atpl (7.34)	ix.js (12.02)	sax (40.39)
bcrypt (15.56)	jjv (29.54)	semver (19.52)
bl (6.17)	json-pointer (71.22)	sendgrid (14.96)
buffer-equal (1.83)	jsonwebtoken (3.67)	sinon (x)
bunyan-logentries (10.05)	jwt-simple (40.17)	sinon-chai (7.45)
chai (x)	lazy.js (x)	sjcl (x)
change-case (12.54)	leaflet (8.82)	socket.io (13.18)
checksum (5.90)	less (12.84)	source-map (40.67)
clone (x)	levelup (4.43)	stream-to-array (5.13)
commander (68.03)	libxmljs (x)	superagent (0.61)
consolidate (14.78)	logg (x)	supertest (97.48)
convert-source-map (2.14)	long (523.51)	swiz (28.97)
cookie (87.93)	marked (65.51)	through (215.68)
cookie.js (67.52)	mdns (1.18)	timezone-js (31.43)
deep-diff (11.99)	mime (46.33)	tv4 (27.41)
deep-freeze (x)	minimatch (x)	twig (24.44)
detect-indent (28.04)	minimist (10.38)	type-detect (17.47)
diff (66.84)	mockery (43.25)	underscore.string (x)
dustjs-linkedin (31.80)	mongoose-mock (61.83)	universal-analytics (32.95)
esprima (58.00)	mousetrap (11.06)	update-notifier (4.43)
event-loop-lag (50.02)	nconf (11.07)	uri-templates (57.39)
exit (52.73)	nedb (59.69)	validator (x)
form-data (1.00)	nexpect (9.69)	vinyl (4.64)
formidable (x)	nightmare (2.26)	vinyl-fs (17.74)
fs-extra (12.70)	noble (0.88)	vinyl-source-stream (1.41)
gruntjs (x)	node-fibers (9.99)	websocket (1.30)
gulp (x)	node-git (1.40)	which (6.05)
gulp-autoprefixer (29.38)	node-mysql (x)	winston (29.82)
gulp-if (x)	node-restify (22.13)	wrench (7.31)
gulp-istanbul (28.89)	node-tar (35.75)	ws (x)
gulp-mocha (41.05)	node-uuid (682.42)	yargs (10.94)
gulp-replace (37.87)	nodemailer (25.38)	yosay (9.47)
gulp-sass (12.74)	nopt (x)	zeroclipboard (x)
gulp-typedoc (16.65)	oboe.js (x)	