Testing and Evolving TypeScript Declaration Files with Program Analysis

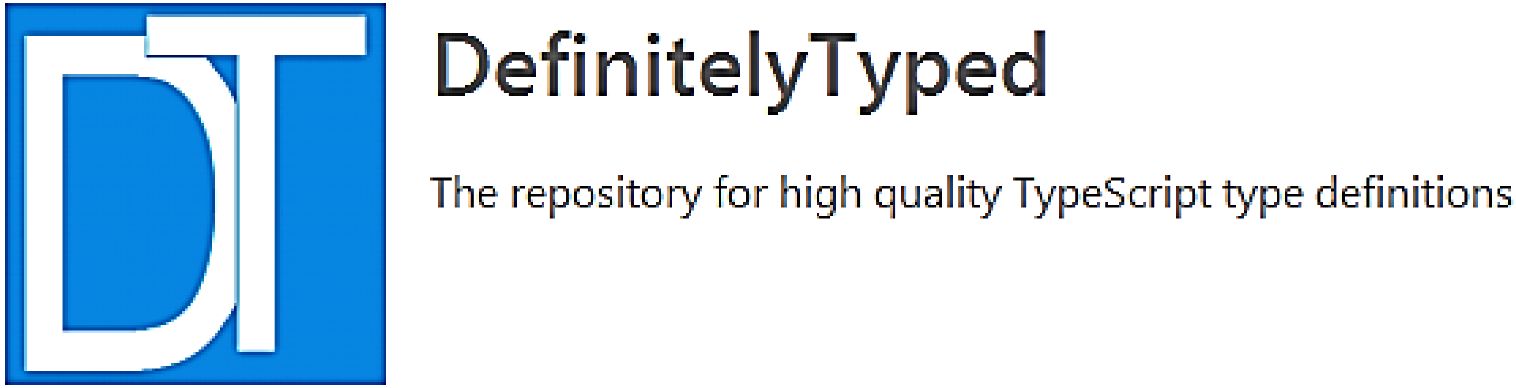
Erik Krogh Kristensen

joint work with Anders Møller

TypeScript

- Microsoft's extension of JavaScript
- Adds **optional types**
  - optional type declarations
  - classes, modules, ...
- Static type checking
- Compiled to JavaScript

Using JavaScript libraries in TypeScript applications

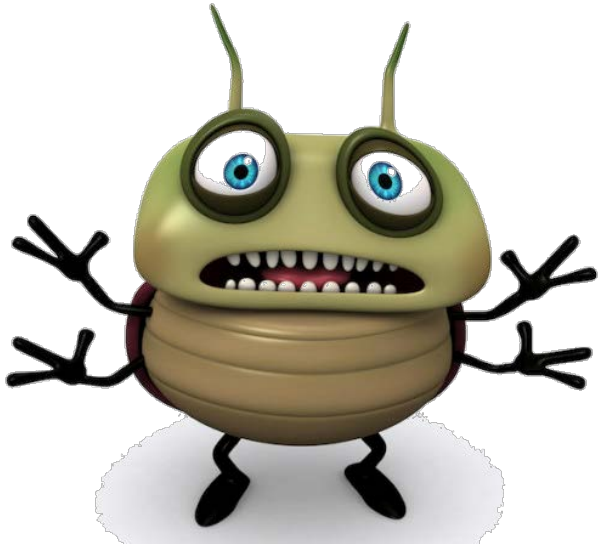


DefinitelyTyped

The repository for high quality TypeScript type definitions

- Contains declarations for >4000 libraries
- Hand-written, lots of bugs

  ⇒ mislead type checking
  and code completion!



An example: p2.js

JavaScript library implementation

```
function Constraint(...) {
    /**
     * Equations to be solved in
     * this constraint
     *
     * @property equations
     * @type {Array}
     */
    this.equations = [];
    ...
}
...
```

TypeScript type declaration

```
class Constraint {
    constructor(...);
    equeations: Equation[];
    ...
}
```

Another example: d3.js

JavaScript library implementation

```
d3.layout.bundle = function() {
  return function(links) {
    var paths = []
    for (var i=0; i<links.length; ++i) {
      paths.push(d3_layout_bundlePath(links[i]))
    }
    return paths;
  };
};
function d3_layout_bundlePath(link) {
  var start = link.source
  var end = link.target
  var lca = d3_layout_bundleLCA(start, end)
  var points = [ start ]
  while (start !== lca) {
    start = start.parent
    points.push(start)
  }
  var k = points.length
  while (end !== lca) {
    points.splice(k, 0, end)
    end = end.parent
  }
  return points
}
```

TypeScript type declaration

```
declare module d3.layout {
    function bundle(): BundleLayout
    interface BundleLayout{
        (links: GraphLink[]): GraphNode[]
    }
    interface GraphLink {
        source: GraphNode
        target: GraphNode
    }
    interface GraphNode {
        parent: GraphNode
        /* some properties omitted ... */
    }
}
```

Three research challenges

1. How to **detect mismatches** between library implementations and type declarations**?**
2. How to **infer** type declarations for libraries**?**
3. How to **evolve** type declarations, as the library code evolves**?**

Three research challenges

1. How to **detect mismatches** between library implementations and type declarations**?**
2. How to **infer** type declarations for libraries**?**
3. How to **evolve** type declarations, as the library code evolves**?**

Existing approaches are limited

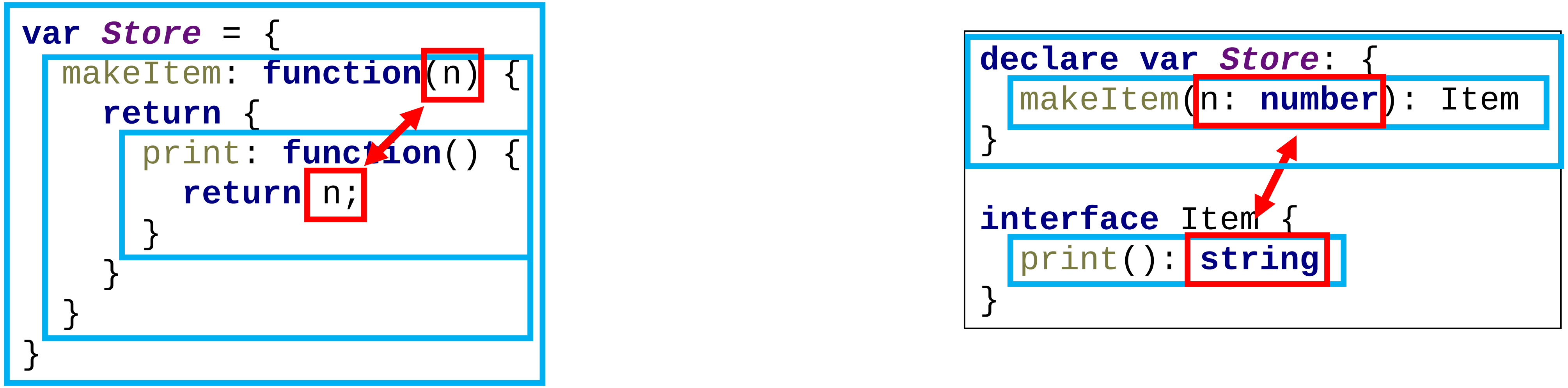
- **TSCheck** (Feldthaus and Møller 2014)**:** Based on static analysis, imprecise
- **TPD** (Williams et al. 2017)**:** Require existing unit tests

TSTest – feedback-directed random testing

[Type Test Scripts for TypeScript Testing, Kristensen & Møller, OOPSLA 2017]

Based on *automated testing*

(Randoop: Feedback-directed random test generation, Pacheco, Lahiri, Ernst, and Ball, ICSE'07)



JavaScript library implementation

```
var Store = {
  makeItem: function(n) {
    return {
      print: function() {
        return n;
      }
    }
  }
}
```

TypeScript type declaration

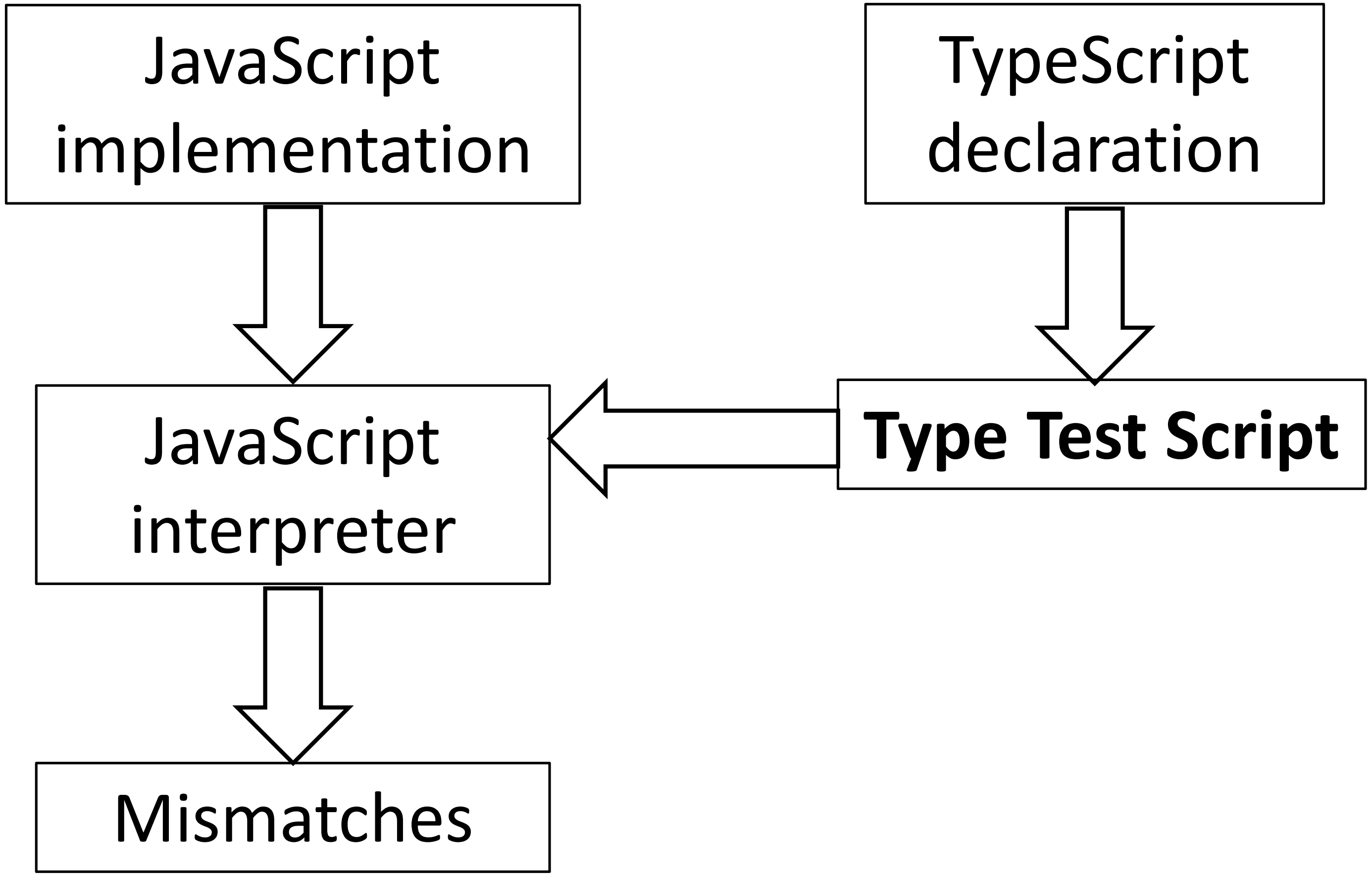
```
declare var Store: {
  makeItem(n: number): Item
}

interface Item {
  print(): string
}
```

How to adapt Randoop-style testing from Java to TypeScript?

(structural typing, higher-order functions, generics, …)

TSTest – how it works



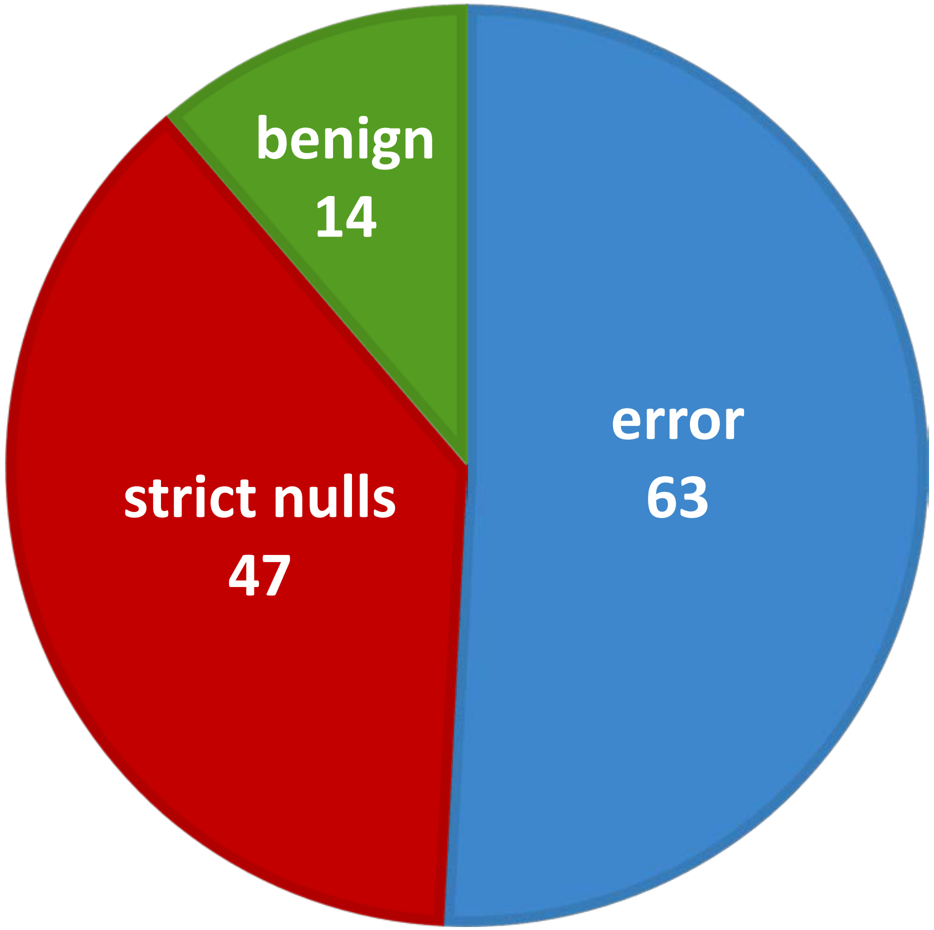
TSTest – experimental results

- 54 benchmarks

- Running each type test script for 10 seconds results in 2804 found mismatches (many with same root cause)
- Mismatches found in 49/54 benchmarks
- Finds many mismatches that are missed by previous work (TSCheck)

Are the mismatches benign or serious?

- Sampled **124** random mismatches
- No false positives



Benign?

redux.d.ts

```
export function bindActionCreators
  <A extends ActionCreator<any>>
  (actionCreator: A): A;
```

redux.js

```
function bindActionCreators(creators) {
    var result = {};
    for (var key in creators) {
        var creator = creators[key];
        if (typeof creator === 'function') {
            result[key] = bindActionCreator(creator, dispatch);
        }
    }
    return result;
}
```

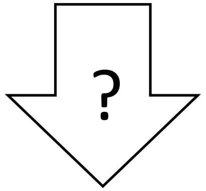
Three research challenges

1. How to **detect mismatches** between library implementations and type declarations**?**
2. How to **infer** type declarations for libraries**?**
3. How to **evolve** type declarations, as the library code evolves**?**

The Holy Grail

Infer the same declaration that an expert human would write

```
function get(obj, k) {
    return obj[k];
}
```



?

```
function get(obj: number[], k: number): number

function get<T>(obj: T[], k: number): T

function get<T>(obj: {[i: string]: T}, k: string): T

function get<T, K extends keyof T>(obj: T, k: K): T[K];
```

TSInfer – a declaration file inference tool [FASE 2017]

- Dynamically analyze library initialization
  Snapshot of heap after loading
- Extract modules, classes, fields
- Static analysis to infer function signatures

Static analysis in TSInfer

- **TSInfer** must *infer* parameter types and return types!
  Also for methods that are never called within the library

- Unification-based too imprecise
- Instead: upper-bound and lower-bound à la Pottier
  A framework for type inference with subtyping, François Pottier, ICFP’98
- Unsound, flow-insensitive, context-insensitive
- Analyze entire library once
  To get information about how the library uses itself

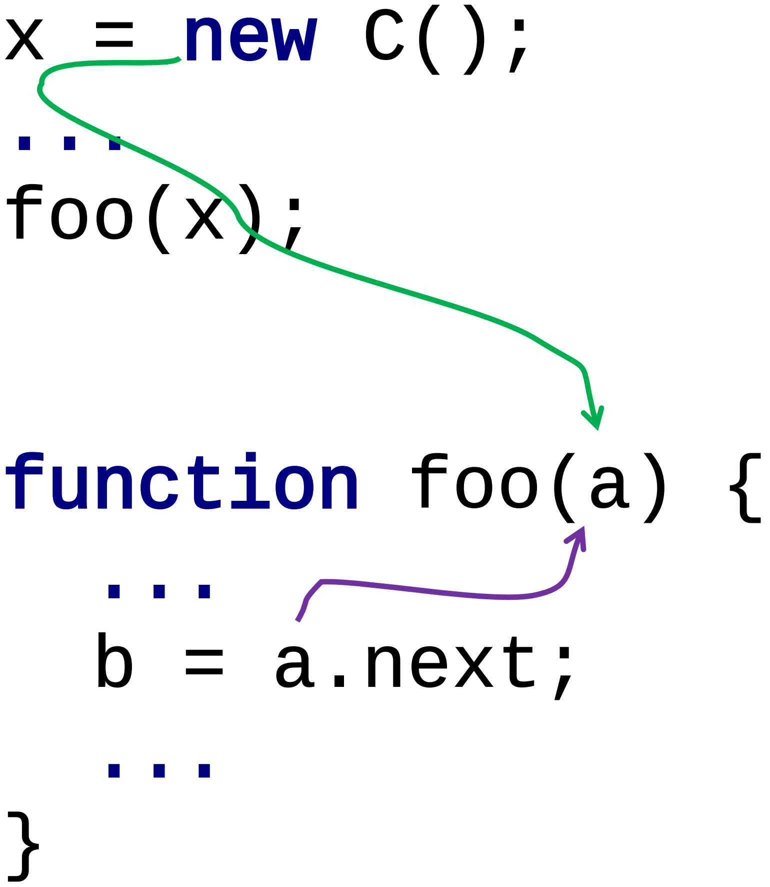
Upper-bound and lower-bound dataflow analysis à la Pottier

- Forward dataflow analysis (lower-bound)

  “what values may flow in?”

- Backward dataflow analysis (upper-bound)

  “how may the values be used?”



```
x = new C();
...
foo(x);


function foo(a) {
  ...
  b = a.next;
  ...
}
```

Both kinds of information give useful hints to types, when analyzing libraries without the applications!

Example output from running TSInfer on PixiJS 2.2

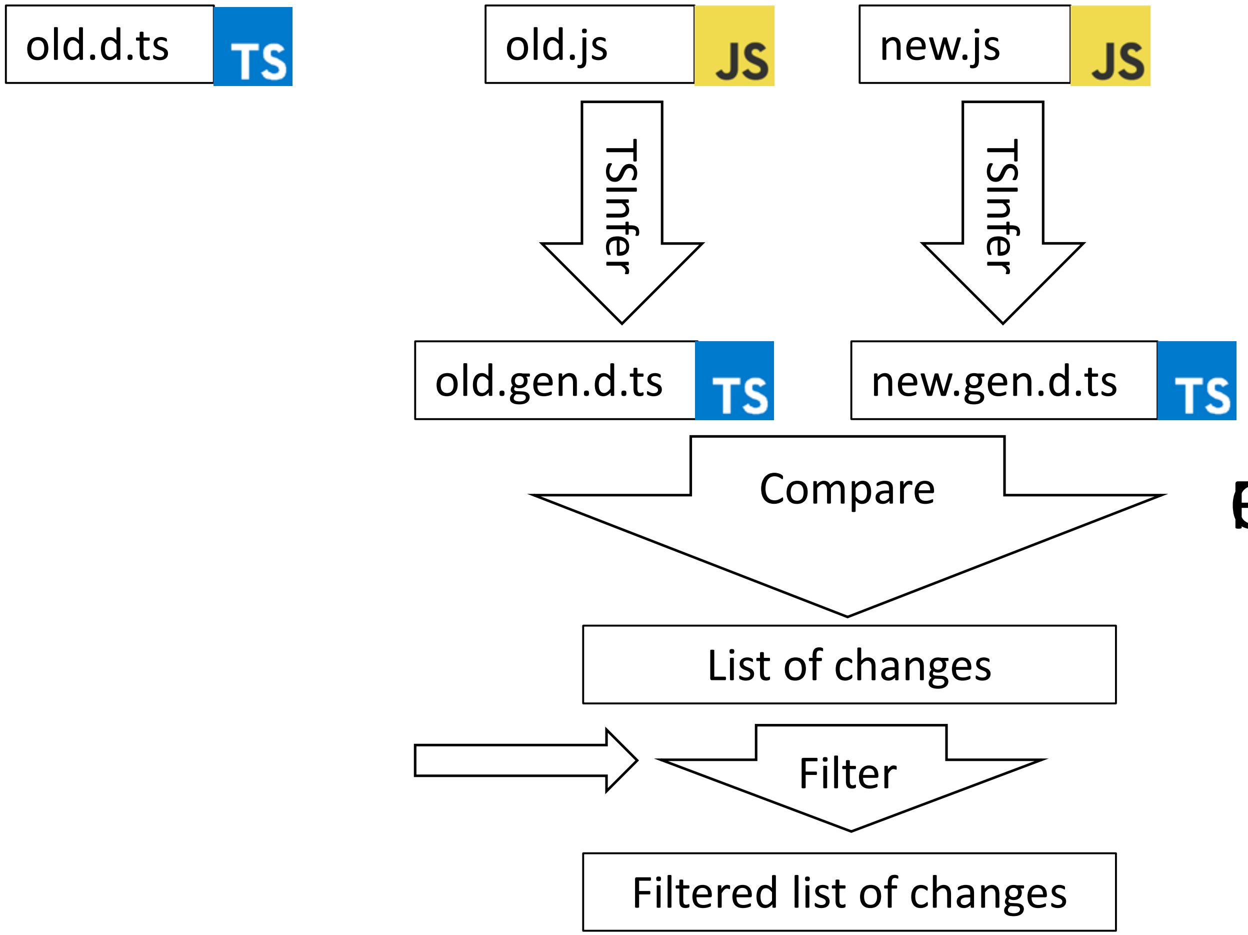
Our goal is to get close to what a human would write

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

Three research challenges

1. How to **detect mismatches** between library implementations and type declarations**?**
2. How to **infer** type declarations for libraries**?**
3. How to **evolve** type declarations, as the library code evolves**?**

TSEvolve – a tool for fixing out-of-date declaration files



First naive approach

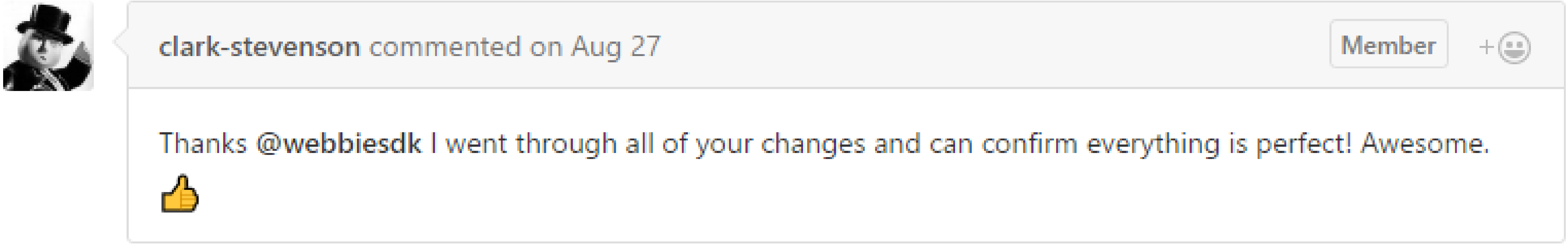
Our approach

Too many false positives, due to

- imprecise static analysis
- intentional mismatches

TSEvolve: pull requests

- Updated type declaration files for 6 different libraries
- From 30 to 516 lines patched
- No prior experience with the libraries
- Done in about 1 day of work

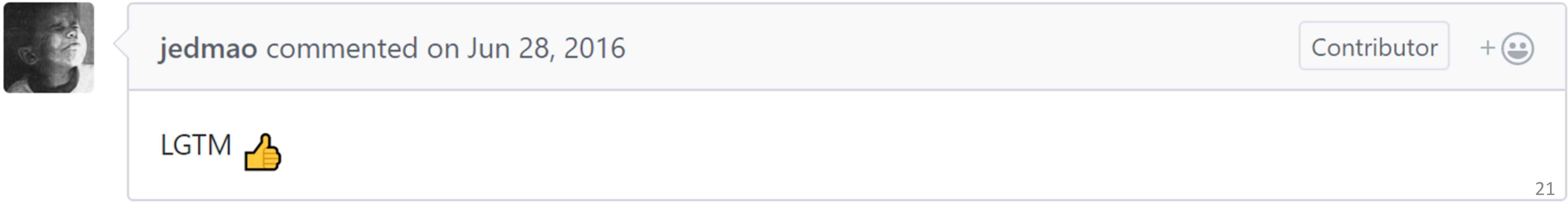


clark-stevenson commented on Aug 27

Member

Thanks **@webbiesdk** I went through all of your changes and can confirm everything is perfect! Awesome.

👍



jedmao commented on Jun 28, 2016

Contributor

LGTM 👍

Conclusion

- Optional types have become popular
- Need to interact with untyped libraries
- Static/dynamic program analysis to the rescue!
    - **TSTest** [OOPSLA 2017]
    - **TSInfer** & **TSEvolve** [FASE 2017]