Flow: Static Type Checker for JavaScript

Andranik Barseghyan

https://github.com/Rebiss/Presentation

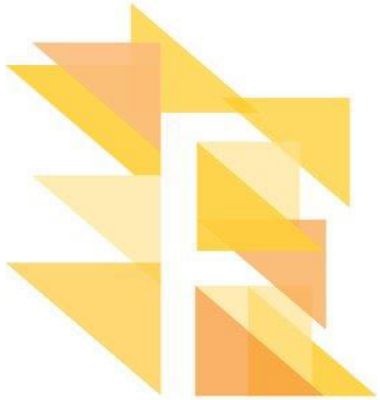
Static Type Checker (Flow)

Overview

- Static type checker
- Flow && TypeScript
- What is Flow?
- How dose it work?
- Basic Setup
- What dose it look like?

  Code Example.
- Conclusion
- Resources

Static Type Checker



TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.

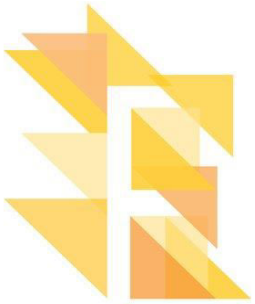
Visual Studio Code

Microsoft

https://www.typescriptlang.org

Flow

OCaml



React

BABEL

https://flow.org/

Flow vs TypeScript

-Checker	-Compiler
-Non-nullable by default	-Nullable by default
-Focused on **Soundness**	-Focused on **Tooling & Scalability**
-Written in OCaml: OSX Linux	-Written in **TypeScript: any OS**
-Works without any annotations	-Great IDE/Editor
-Works out of the box with React	-Used as default by more and more libraries

How dose it work?

Parse Code and generate AST

Inference phase

. Traverse AST and gather information
. Create Flow Graph

Evaluation

. Traverse Flow Graph and find errors

For multiple modules

. First, create Flow Graph for each module in parallel
. Then connect Flow Graphs at their touch points

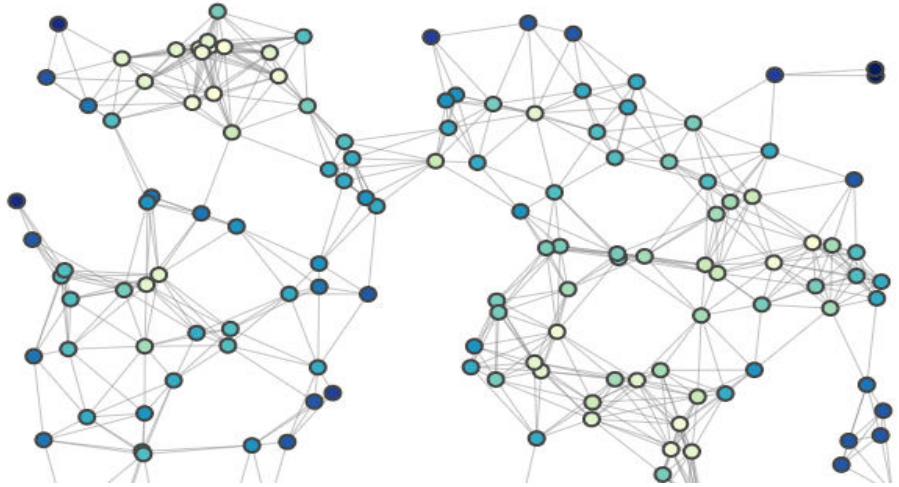
Flow Graph

Not the typical eager approach for type checking

. Type is decided as late as possible

Possibilities beyond type-checking

. Tracking data through the application
. Dead code elimination
(inverse approach of tree-shaking)



Features

Maybe Types	Union and Intersection Types
Array and Tuples	Type Aliases
Classes	TypeoOf type
Object	Dinamic Type Tests
Function	Disjoint Unions
Destructuring	Modules
Declaration	

Example

```
1   /**@flow */
2
3   function foo(x) {
4       return x * 10;
5   }
6
7   foo('string');
```

```
1   /**@flow */
2
3   function foo(x: number, y) {
4       return x * 10 * y;
5   }
6
7   foo(5,'string');
```

```
1   /**@flow */
2
3   function foo(x: number, y: number): number {
4       return x * 10 * y;
5   }
6
7   foo(5, 11);
```

Primitive Types

- Number (Infinity, NaN)
- String
- Boolean
- null
- undefined (void in Flow types)
- Symbols (not yet supported in Flow)

```
const num: ?number = 11;

const str: string = 'string';

const bool: boolean = true;
```

Type

```flow
/**@flow */
type bookType = {
    id: number,
    author: string,
    isActive: boolean,
    other: {
        title: string,
        count: number,
        published: number,
        date: string | number,
    }
};

const book: bookType = {
    id: 1558669888963,
    author: 'Hermann Hesse',
    isActive: true,
    other: {
        title: 'The Glass Bead Game',
        count: 450,
        published: 1942,
        date: ' Wed, 18 Nov 1942 10:43:12 GMT'
    }
}
```

```flow
/**@flow */
/** export.js */
export type bookType = {
    id: number,
    author: string,
    isActive: boolean,
    other: {
        title: string,
        count: number,
        published: number,
        date: string | number,
    }
};
```

```flow
/**@flow */
/** import.js */

import { bookType } from './export';
```

Mixed & Any Types

```
function stringifyBasicValue(value: string | number) {
  return '' + value;
}
```

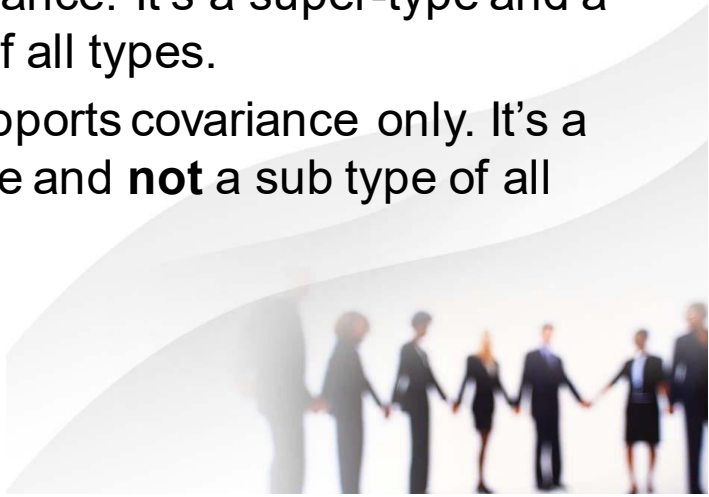
```
function getTypeOf(value: mixed): string {
  return typeof value;
}
```

```
// @flow
function add(one: any, two: any): number {
  return one + two;
}

add(1, 2);     // Works.
add("1", "2"); // Works.
add({}, []);   // Works.
```

The difference is the 'vice-verca': any can flow into other types but mixed can not.

Any supports covariance and contravariance. It's a super-type and a subtype of all types.

Mixed supports covariance only. It's a super-type and **not** a sub type of all types.



Variables, Array & Tuple Types

```
1  // @flow
2  let foo: number = 1;
3  foo = 2;   // Works!
4  // $ExpectError
5  foo = "3"; // Error!
```

```
1  let arr1: Array<boolean> = [true, false, true];
2  let arr2: Array<string> = ["A", "B", "C"];
3  let arr3: Array<mixed> = [1, true, "three"]
```

```
1  let arr: number[] = [0, 1, 2, 3];
```

```
1  // @flow
2  let arr1: ?number[] = null;   // Works!
3  let arr2: ?number[] = [1, 2]; // Works!
4  let arr3: ?number[] = [null]; // Error!
```

```
1  // @flow
2  var fooVar /* : number */ = 1;
3  let fooLet /* : number */ = 1;
4  var barVar: number = 2;
5  let barLet: number = 2;
```

```
1  let tuple1: [number] = [1];
2  let tuple2: [number, boolean] = [1, true];
3  let tuple3: [number, boolean, string] = [1, true, "three"];
```

Union Types

```flow
// @flow
function toStringPrimitives(value: number | boolean | string) {
  return String(value);
}

toStringPrimitives(1);       // Works!
toStringPrimitives(true);    // Works!
toStringPrimitives('three'); // Works!

// $ExpectError
toStringPrimitives({ prop: 'val' }); // Error!
// $ExpectError
toStringPrimitives([1, 2, 3, 4, 5]); // Error!
```

```flow
// @flow
type Success = { success: true, value: boolean };
type Failed  = { success: false, error: string };

type Response = Success | Failed;

function handleResponse(response: Response) {
  if (response.success) {
    var value: boolean = response.value; // Works!
  } else {
    var error: string = response.error; // Works!
  }
}
```

```flow
// @flow
// $ExpectError
function toStringPrimitives(value: number | boolean | string): string {
  if (typeof value === 'number') {
    return String(value);
  } else if (typeof value === 'boolean') {
    return String(value);
  }
}
```

Intersection Types

```flow
// @flow
type A = { a: number };
type B = { b: boolean };
type C = { c: string };

function method(value: A & B & C) {
  // ...
}

// $ExpectError
method({ a: 1 }); // Error!
// $ExpectError
method({ a: 1, b: true }); // Error!
method({ a: 1, b: true, c: 'three' }); // Works!
```

```flow
// @flow
type One = { foo: number };
type Two = { bar: boolean };

type Both = One & Two;

var value: Both = {
  foo: 1,
  bar: true
};
```

```flow
// @flow
type A = { a: number };
type B = { b: boolean };
type C = { c: string };

function method(value: A & B & C) {
  var a: A = value;
  var b: B = value;
  var c: C = value;
}
```

Maybe Types

```
1  // @flow
2  function acceptsMaybeNumber(value: ?number) {
3    // ...
4  }
5
6  acceptsMaybeNumber(42);        // Works!
7  acceptsMaybeNumber();          // Works!
8  acceptsMaybeNumber(undefined); // Works!
9  acceptsMaybeNumber(null);      // Works!
10 acceptsMaybeNumber("42");      // Error!
```

```
1  // @flow
2  function acceptsMaybeNumber(value: ?number) {
3    if (value !== null && value !== undefined) {
4      return value * 2;
5    }
6  }
```

Generic Types

```flow
// @flow

type IdentityWrapper = {
  func<T>(T): T
}

function identity(value) {
  return value;
}

function genericIdentity<T>(value: T): T {
  return value;
}

// $ExpectError
const bad: IdentityWrapper = { func: identity }; // Error!
const good: IdentityWrapper = { func: genericIdentity }; // Works!
```

```flow
// @flow
function identity<T>(value: T): T {
  return value;
}

let one: 1 = identity(1);
let two: 2 = identity(2);
// $ExpectError
let three: 3 = identity(42);
```

```flow
// @flow
function identity<T>(val: T): T {
  return val;
}

let foo: 'foo' = 'foo';           // Works!
let bar: 'bar' = identity('bar'); // Works!
```

Class (Interface)

```flow
// @flow
interface Serializable {
  serialize(): string;
}

class Foo {
  serialize() { return '[Foo]'; }
}

class Bar {
  serialize() { return '[Bar]'; }
}

const foo: Serializable = new Foo(); // Works!
const bar: Serializable = new Bar(); // Works!
```

```flow
class MyClass {
  method(value: string): number { /* ... */ }
}
```

```flow
// @flow
class MyClass {
  method() {
    // $ExpectError
    this.prop = 42; // Error!
  }
}
```

```flow
// @flow
class MyClass {
  prop: number;
  method() {
    this.prop = 42;
  }
}
```

Prop Types

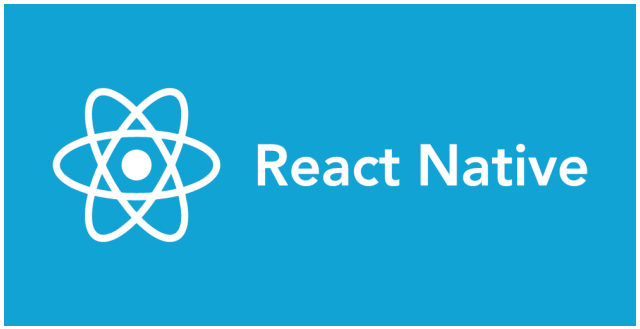
```
1  import React, { Component }from 'react';
2  import PropTypes from 'prop-types';
3
4  class MyComponent extends Component {
5
6      static propTypes = {
7          className: PropTypes.string,
8          children: PropTypes.node,
9          isOpen: PropTypes.bool,
10         /** ..... */
11     };
12
13     static defaultProps = {
14         className: '',
15         children: null,
16         isOpen: false,
17         /** ..... */
18     };
19
20     render() {
21         /**.........  */;
22     }
23 }
24
25 export default MyComponent;
```

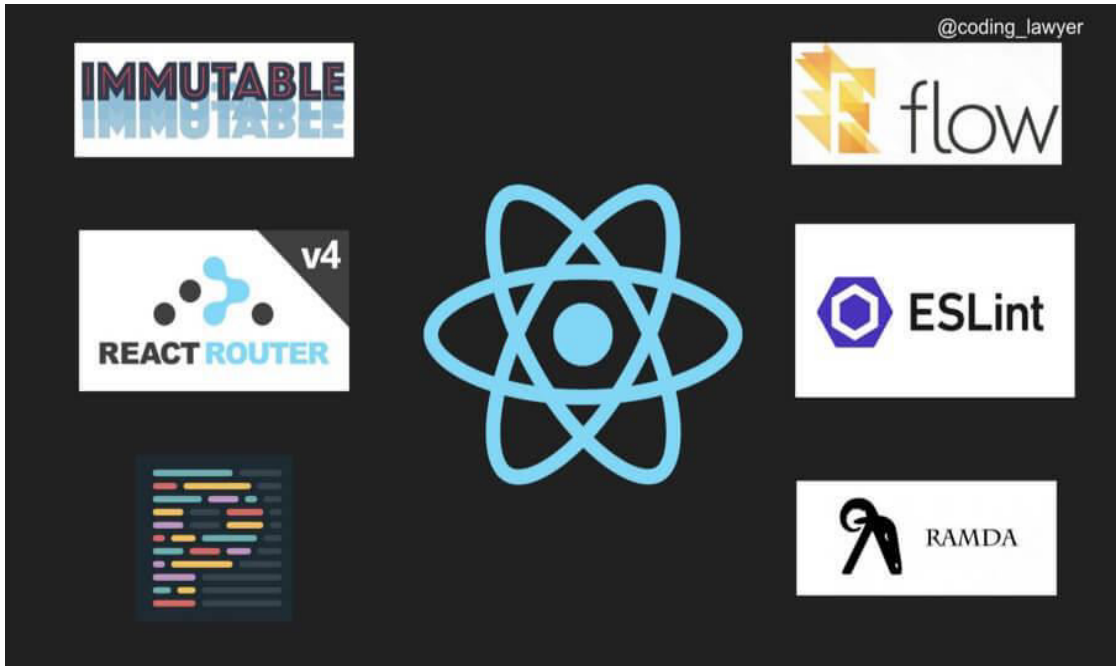
Class & Functional Component

```tsx
import * as React from 'react';

type Props = {
  foo: number,
  bar?: string,
};

class MyComponent extends React.Component<Props> {
  render() {
    this.props.doesNotExist; // Error! You did not define a `doesNotExist`

    return <div>{this.props.bar}</div>;
  }
}

<MyComponent foo={42} />;
```

```tsx
import * as React from 'react';

type Props = {
  foo: number,
  bar?: string,
};

function MyComponent(props: Props) {
  props.doesNotExist; // Error! You did not define a `doesNotExist` prop.

  return <div>{props.bar}</div>;
}

<MyComponent foo={42} />
```

Objective Caml Facebook.





Conclusion

- Bug & Error
- Documentation
- Refactoring
- Unit Testing


Thank you