Type Systems for JavaScript

Elm, Flow, and TypeScript

Jfokus 2017, Stockholm, Sweden

Slides for this talk: http://bit.ly/types-jfokus

Oliver Zeigermann / @DJCordhose

Extended Version (constantly updated): http://bit.ly/js-types

Disclaimer

I am a practitioner, using both TypeScript and Flow in my projects

I am no type theory expert

I have not used Elm in any real world project, yet

Not a single cat image in this talk

Even worse: not even any image

My compensation to make this a valid talk

https://twitter.com/Creatuluw/status/749151998415634432

Why using type systems?

type systems make code easier to maintain

type annotations / inferred types

- can make code more readable
- can make code easier to analyse
- can allow for reliable refactoring
- can allow for generally better IDE support
- can catch some (type related) errors early

Anders Hejlsberg@Build2016: *Big JavaScript codebases tend to become "read-only".*

TypeScript

ease of use and tool support over soundness

- http://www.typescriptlang.org/
- By Microsoft (Anders Hejlsberg)
- Based on ES6 (probably ES7/ES8)
- Adds optional type annotations, visibility, and decorators
- Compiler checks and removes annotations
- 2.x with major changes released recently

We restrict ourselves to version 2 here

Flow

soundness, no runtime exceptions as goal

- http://flowtype.org/
- By Facebook
- *Flow is a static type checker, designed to quickly find errors in JavaScript applications*
- Not a compiler, but checker
- If present, type annotations can very easily be removed by babel for runtime

Elm

simplicity, soundness, no runtime exceptions

- http://elm-lang.org/
- Language of its own
- functional, not object-oriented
- no null, no mutation
- Geared towards Web Applications
- Compiler creates JavaScript

Demo

Some basic TypeScript 2.2 hacking in Visual Studio Code 1.10

TypeScript (similar to what we just hacked)

```
let foo: string;
// variables can have type information
let foo: string;
foo = 'yo';
// Error: Type 'number' is not assignable to type 'string'.
foo = 10;
```

```
// types can be inferred (return type)
function sayIt(what: string) {
    return `Saying: ${what}`;
}
const said: string = sayIt(obj);
```

```
class Sayer {
    what: string; // mandatory

    constructor(what: string) {
        this.what = what;
    }

    // return type if you want to
    sayIt(): string {
        return `Saying: ${this.what}`;
    }
}
```

Flow

```
// variables can have type information
let foo: string;
foo = 'yo';
// Error: number: This type is incompatible with string
foo = 10;
```

```
// types can be explicit (parameter) or inferred (return type)
function sayIt(what: string) {
    return `Saying: ${what}`;
}
const said: string = sayIt(obj);
```

```
class Sayer {
    what: string; // type also mandatory

    constructor(what: string) {
        this.what = what;
    }

    // return type if you want to
    sayIt(): string {
        return `Saying: ${this.what}`;
    }
}
```

Flow and TypeScript basics are pretty similar

Those basic features help with documentation, refactoring, and IDE support

Elm: a totally different story

```
let
    -- declaration using type
    foo : String
    foo = "yo"
    -- Error: everthing is const, can not re-assign
    foo = "yo yo"

    foo2 : String
    -- Error: `The definition of `obj2` does not match its type annotation.`
    foo2 = 10
```

```
let
    -- type annotations are optional, can be inferred
    sayIt : String -> String
    sayIt what =
        "Saying: " ++ what

    said : String
    said = sayIt obj
```

No classes and methods in elm

Structural Typing for both TypeScript and Flow

```
class Dog {
  name: string;
  woof() {...}
}

interface NamedObject {
    name: string;
}
// this is fine class does not need to explicitly implement it
let namedObject: NamedObject = dog;
```

```
// same thing, also fine
let namedObject: NamedObject = {
    name: "Olli"
};

// not fine in either, missing name
let namedObject: NamedObject = {
    firstName: "Olli"
};
```

Structural vs Nominal Typing

- Nominal Typing: types are compatible when their declared types match
- Structural Typing: types are compatible when their structures match
- Java, C#, C++, C all use nominal typing exclusively
- Flow classes are also treated as nominal types
- TypeScript classes are treated as structural types
- Everything else in both Flow and TypeScript uses structural typing
- Elm always uses structural typing with exact matches on Records

Nullability

One of my main sources of runtime exceptions when programming Java

Even after many years it is still surprising how many corner cases I miss in complex code

Flow

what is the result here in pure JavaScript?

```
function foo(num) {
    if (num > 10) {
        return 'cool';
    }
}
console.log(foo(9).toString());
```

```
"Uncaught TypeError: Cannot read property 'toString' of undefined"
```

What the flow checker thinks about this

```
// error: call of method `toString`.
// Method cannot be called on possibly null value
console.log(foo(9).toString());
```

To fix this, we need to check the result

```
const fooed = foo(9);
if (fooed) {
    fooed.toString();
}
```

Types are non-nullable by default in flow

TypeScript

```
// both TypeScript and flow allow
// to put the type annotation here instead of using inference
function foo(num: number) {
    if (num > 10) {
        return 'cool';
    }
}
```

```
// same as flow
const fooed: string|void = foo(9);
if (fooed) {
    fooed.toString();
}
```

```
// or tell the compiler we know better (in this case we actually do)
fooed!.toString();
```

Only applies to TypeScript 2.x

Only works when *strictNullChecks* option is checked

All types nullable by default in TypeScript 1.x

Elm

There neither is `null` nor `undefined` in elm

Rather Maybe plus pattern matching

```
-- Maybe is predefined
-- http://package.elm-lang.org/packages/elm-lang/core/latest/Maybe
type Maybe a = Nothing | Just a
```

```
foo : Int -> Maybe String
foo num =
    if num > 10 then
        Just "cool"
    else
        Nothing
```

```
-- pattern matching (need to match all cases)
case (foo 11) of
    Just message -> message
    Nothing -> ""
```

Generic Type information

Types can be parameterized by others

Most common with collection types

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// nope, no cat
cats.push(10);
```

```
// nope, no cat
cats.push(new Animal('Fido'));
```

```
// cool, is a cat
cats.push(new Cat('Purry'));
```

```
// cool, cat is a sub type of animal
animals.push(new Cat('Purry'));
```

Up to this point this pretty much works in Flow and TypeScript the same way ...

... but wait

TypeScript

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// error TS2322: Type 'Animal[]' is not assignable to type 'Cat[]'.
//  Type 'Animal' is not assignable to type 'Cat'.
//    Property 'purrFactor' is missing in type 'Animal'.
cats = animals;
```

```
// wow, works, but is no longer safe
animals = cats;
```

```
// because those are now all cool
animals.push(new Dog('Brutus'));
animals.push(new Animal('Twinky'));
```

```
// ouch:
cats.forEach(cat => console.log(`Cat: ${cat.name}`));
// Cat: Purry
// Cat: Brutus
// Cat: Twinky
```

TypeScript allows for birds and dogs to be cats here :)

Flow

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// ERROR
// property `purrFactor` of Cat. Property not found in Animal
cats = animals;
```

```
// same ERROR
animals = cats;
```

Flow does not have have this caveat

The flipside

This code is safe (as we access cats in a readonly fashion)

```
function logAnimals(animals: Array<Animal>) {
    animals.forEach(animal => console.log(`Animal: ${animal.name}`));
}

logAnimals(cats);
```

- This works in TypeScript (and it should)
- however, potentially not safe, there is nothing to keep us from writing to cats
- Flow does not allow this, even though it is safe

much despised Java generics excel here as they can actually make that code safe (another difference: Use-site variance)

```
// Java
void logAnimals(List<? extends Animal> animals) {
    animals.forEach(animal -> System.out.println("Animal: " + animal.name));
    // illegal:
    animals.add(new Animal("Twinky"));
}
```

Some Type Inference Magic

Consider

```
class Dog { woof() { } }

const animals = [];
animals.push(new Dog());
```

both TypeScript and Flow know this is safe, as we have only added Dogs so far

```
animals.forEach((animal: Dog) => animal.woof());
```

Adding Cats *later* and thus changing array type later

```
class Cat { meow() { } }
animals.push(new Cat());
```

does not affect TypeScript (correct), but makes Flow fail

Elm

does not have classes or subtypes

has Records (like JavaScript Objects) and generic data structures (e.g. List)

```
type alias Animal = { name : String }
someAnimal1 = { name = "Patrick"}

animals : List Animal -- generic data structure
animals = [ someAnimal1, someAnimal2, ... ]
```

```
type alias Cat = { name : String, coatColor : String }

cats : List Cat
cats = [ someCat1, someCat2, ... ]
```

```
-- sure
moreAnimals : List Animal
moreAnimals = animals

-- Error: Looks like a record is missing the `coatColor` field.
evenMoreAnimals : List Animal
evenMoreAnimals = cats

-- nope, same problem
moreCats : List Cat
moreCats = animals
```

Differences in Generic Types

- TypeScript
  - always covariant (more special):
    parametric types are compatible if the type to assign from has a more special type parameter
  - seems most intuitive, allows for obviously wrong code, though
- Flow
  - using generic types you can choose from invariant (exact match), covariant + (more special), and contravariant - (more common)
  - Array in Flow has an invariant parametric type
  - more expressive, harder to master, disallows some correct code
- Elm
  - Generic data structures using type variables
  - all types have to match exactly

Mutation, const

TypeScript and flow: same as JavaScript (const optional, immutable via lib)

TypeScript: readonly for properties

Elm: everything always immutable and const

`Changing` records in Elm

Central Question: If everything always immutable and const, how do you make modifications?

Answer:

- you do not really make mutations
- instead create a new record
- taking over some of the properties of the old record and
- setting some new properties

```
type alias Cat = { name : String, coatColor : String, age: Int}
someCat = { name = "Purry", age = 2, coatColor = "gray"}
```

```
haveBirthday : Cat -> Cat
haveBirthday cat =
  -- make a copy, but with changed age
  { cat | age = cat.age + 1 }
```

```
agedCat : Cat
agedCat = haveBirthday someCat
```

`any` type

can be anything, not specified

can selectively disable type checking

```
function func(a: any) {
    return a + 5;
}
```

```
// cool
let r1: string = func(10);

// cool
let r2: boolean = func('wat');
```

- *flow / TypeScript 2*: explicit any supported, but any never inferred
- *Elm*: does not exist, everything has exact type

Union Types

aka Disjoint Unions aka Tagged Unions aka Algebraic data types

to describe data with weird shapes

depending on some data other data might apply or not

```
// a disjoint union type with two cases
type Response = Result | Failure;

type Result = { status: 'done', payload: Object }; // all good, we have the data
type Failure = { status: 'error', code: number}; // error, we get the error code
```

Implementation both in Flow and TypeScript

```
function callback(response: Response) {
    // works, as this is present in both
    console.log(response.status);
    // does not work,
    // as we do not know if it exists, just yet
    console.log(response.payload); // ERROR
    console.log(response.code); // ERROR
```

```
    switch (response.status) {
        case 'done':
            // this is the special thing:
            // type system now knows, this is a Result
            console.log(response.payload);
            break;
        case 'error':
            // and this is a Failure
            console.log(response.code);
            break;
    }
}
```

Elm

simple and concise union types

```
type Response = Result String | Failure Int
```

switching over union type alternatives using pattern matching

```
callback : Response -> String
callback response =
    -- pattern matching
    case response of
        Result payload -> payload
        Failure code ->
                if code >= 400 && code < 500 then "you messed up"
                else "we messed up"
```

usage

```
callback (Result "response")
-- response

callback (Failure 404)
-- you messed up
```

Where do they excel?

- TypeScript: *supporting people from Java and C# land*
  - more complete IDE support
  - language server
  - large set of 3rd party declaration files
- Flow: *providing typings for idiomatic JavaScript*
  - easy to get started even with existing project
  - more powerful and flexible generics
  - nominal typing for classes
- Elm: *functional language deliberately different from JavaScript*
  - simplicity of type system (no JavaScript legacy)
  - always completely typed (no any)
  - everything immutable and constant always and everywhere
  - complete package (also great orientation for beginners)

Special thanks for giving feedback and helping with this presentation

- Daniel Rosenwasser: @drosenwasser (from the TypeScript team)
- Avik Chaudhuri: @__avik (from the Flow team)
- Richard Feldman: @rtfeldman and Evan Czaplicki: @czaplic (Elm people)

Thank you!

Questions / Discussion

Oliver Zeigermann / @DJCordhose

Slides for this talk: http://bit.ly/types-jfokus

Extended Version (constantly updated): http://bit.ly/js-types