The design of JavaScript type systems

Dimitris Vardoulakis (dimvar@)
StrangeLoop 2017

Overview

Closure Compiler (Google) 

TypeScript (Microsoft) 

Flow (Facebook) 

Requirements for a JavaScript type system

Features of JS types systems

How a JS type checker works

Brief history of JS types

JS was originally intended for small inline scripts, not for large applications.
Key features missing: modules, types, classes.

Closure Compiler: types based on ES4, objects as modules, goog namespace.

```
goog.provide('a.b.c');
/**
 * @constructor
 * @param {number} x
 */
a.b.c.Foo = function (x) {
  /** @type {number} */
  this.prop = x;
}
```

Later, modules and classes in ES6, TypeScript from Microsoft, and Flow from FB.

Requirements for a JavaScript type system

Handle the native style of programming; don’t restrict the language too much.

Find many bugs, few false positives.

Let users gradually type their code; handle a mix of typed and untyped code.

Fast: O(seconds) for whole-program; sub-second for incremental checking.

These requirements are sometimes conflicting; practice informs balance.

Every other aspect of the design follows from these requirements.

Familiar features in JS type systems

Classes with single inheritance, interfaces with multiple inheritance (like Java). Prototypal inheritance too dynamic.

Type the arguments of operators; forbid most implicit conversions.

```
[] + {}; // warning
“wat” - 1; // warning
```

Fixed-arity functions.

```
function f(x) { return x + 1; }
f(2, 3); // warning
```

Balance between not restricting the language and finding many bugs.

Structural typing

Expecting an object with specific properties, rather than an instance of a specific constructor.

```
function g(x: { a: number, b: string }) { return x.a; }

class Bar {
    a: number;
    b: string;
    constructor() {
        this.a = 1;
        this.b = 'asdf';
    }
}

g(new Bar);
g({ a: 2, b: 'qwer' });
```

Unions and flow-sensitive typing

Checking nullability:

```
function area(x: Rectangle|null) {
  if (x === null) {
    return 0;
  }
  return x.width * x.height; // x is a non-nullable Rectangle here
}
```

`instanceof` to specialize object types:

```
function getFirst(x: Array<string>|string) {
  return x instanceof Array ? x[0] : x.charAt(0);
}
```

Unsoundness

The type checker can infer the wrong type for some expression ==> fail to detect a type error.

Causes of unsoundness:

- Some JS constructs are very dynamic; hard to analyze at compile time.
- Let people add types gradually, without flood of warnings for the untyped parts of the program.
- Avoid complicated type-system features such as dependent types.
- Type checker needs to be fast and infer precise types.

Sources of unsoundness

The unknown type (aka any): a subtype and supertype of all other types. We can always fall back to any for complicated code.

Assume array accesses are in bounds. Usually correct in practice.

```
function f(a: Array<string>, i: number) : string {
  return a[i];
}
```

The alternatives are worse:
- include undefined in the type of every array access.
- complicated type system to track which accesses are safe and which are not.

Check computed property accesses loosely.
In the general case, we can’t know which property is being accessed; type it as any.

And more: complicated changes to prototype chain, property deletions, eval, etc.

How does a JavaScript type checker work?

Phase 1:
Fast, whole-program analysis that collects type definitions and creates the inheritance hierarchy.

Phase 2:
Typecheck each function in isolation, using a (slower) flow-sensitive analysis.

Phase 1: Collect and resolve types

Find user-defined classes and interfaces.

What are the properties on each class?

```
/** @constructor */
function Foo() { this.a = 'a'; }

Foo.prototype.mymethod = function () { this.b = 'b'; };
```

Resolve type aliases.

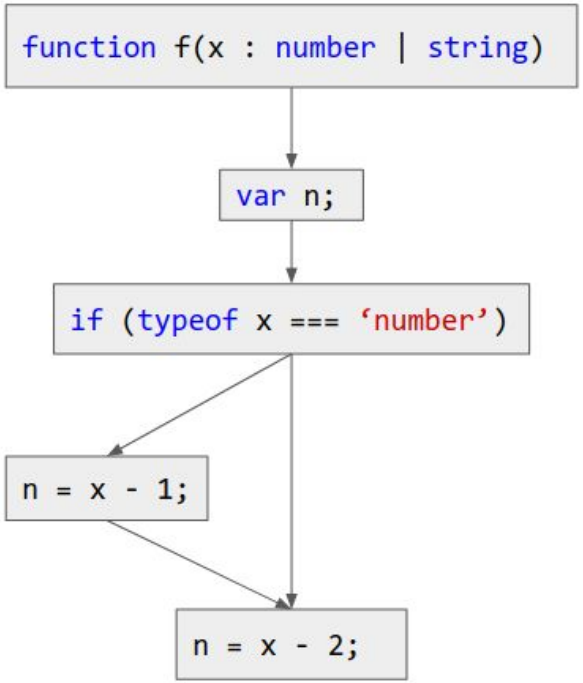
```
type FooBar = Foo | Bar;
```

Create the inheritance hierarchy and detect cycles.

Phase 2: Typecheck each function in isolation

```
function f(x: number | string) {
  var n : number;
  if (typeof x === 'number') {
    n = x - 1; // no warning, x is number
  }
  n = x - 2; // warning, x is number|string
}
```

Phase 2: Typecheck each function in isolation



Phase 2: Typecheck each function in isolation

Caveat: does not handle aliasing correctly.

```
function f(x) {
  var y = x;
  if (y !== null) {
    // The type checker thinks x is still nullable here
  }
}
```

Won't cover:
- Backwards analysis: inferring signatures of unannotated functions
- Analyzing loops
- Implementation choices for fast type checking

Evaluating the quality of a JS type checker

If it is not sound, what guarantees does it provide?

Empirical evaluation based on user feedback.
Does it find bugs people expect it to find?
Does it have many false positives?
Err towards reducing false positives, even if it means finding fewer bugs.

In practice, soundness is a continuum, not a binary property.
Insisting on full soundness for an optional type system is not productive.

Consider a ML classifier that recognizes email spam. It is useful even if it misses some spam and classifies some non-spam as spam.
Similarly, unsound static analyses could be evaluated using precision and recall.

How to write JavaScript that typechecks well

Data definitions should not be too dynamic: the type checker should be able to find all properties on each type, and to construct the inheritance hierarchy.

Whatever is used across scopes should be annotated; do not rely on type inference:
- class and interface properties
- variables used across scopes
- function parameters

```
function f() {
    var x;
    function g() { x = 'asdf'; }
    g();
    var y: number = x; // assigning a string to a number
}
```

Use strict flags provided by your type checker, e.g., to detect unknown types.

So, what’s next for JS types?

The three type systems inform each other.
Non-nullable types, unions, and type aliases now in TypeScript.
Structural interfaces now in Closure Compiler.
TODOs in Closure Compiler: bounded generics, function overloading, tuples.

But: no types in browser; ecosystem fragmentation.

Closure Compiler, TypeScript and Flow have a common core of useful features.
A future standardized JS type system would likely derive from this core.

Type systems for other dynamic languages

The design requirements are the same in other languages.

- Handle the native style of programming
- Find many bugs with few false positives
- Let users gradually type their code
- Fast

Therefore, the high-level design should be transferable too.

- Unsoundness and the any type
- Two-phase analysis
- Best practices: keep data static and annotate types across scopes

Prediction: in the coming years, most dynamic languages will get optional type systems (PHP, Python, Ruby, Lua, etc. Some of these already in progress.)

Thank you!

Relevant info

The implicit-conversion examples were taken from Gary Bernhardt's WAT talk.

Early history of JS types.

Doc listing the differences between the TypeScript and Closure type systems.