No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first"

Andy Hunt

Programming Safely With Types

Ian Thomas | @anatomic

Does JavaScript have types?

(And can they be safe?)

JavaScript's types

JavaScript's types

- Undefined

JavaScript's types

- Undefined
- Null

JavaScript's types

- Undefined
- Null
- Boolean

JavaScript's types

- Undefined
- Null
- Boolean
- String

JavaScript's types

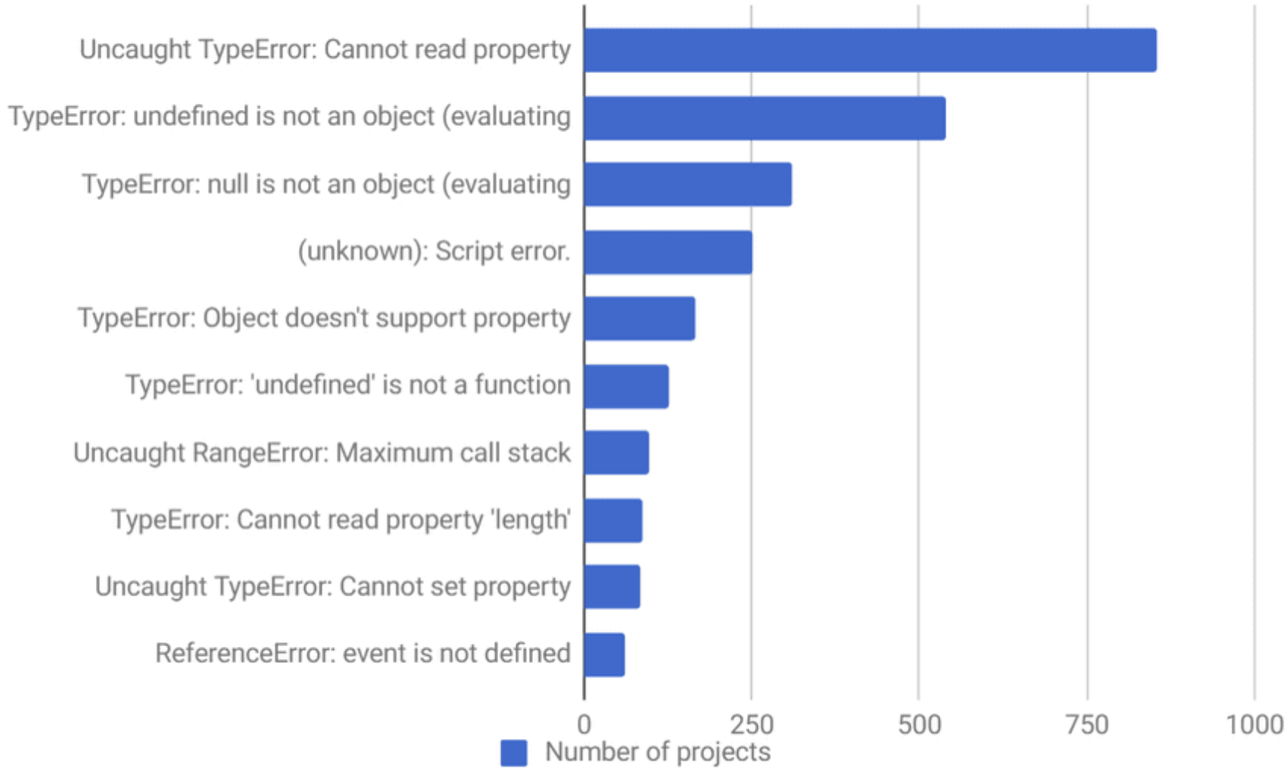
- Undefined
- Null
- Boolean
- String
- Number

JavaScript's types

- Undefined
- Null
- Boolean
- String
- Number
- Object

What is safety?

What are the most frequently reported errors in JavaScript applications?



top 10 javascript errors from 1000+ projects (and how to avoid them)

errorClass

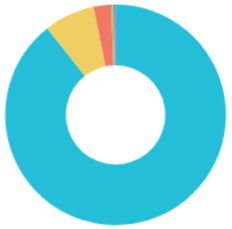
TypeError

7 days ago	26.92%	1.07 M	38.2 k
FIRST SEEN	OF TOTAL ERRORS	INSTANCES	INTERACTIONS AFFECTED

Overview | Error Instances

Top browsers where this error occurred

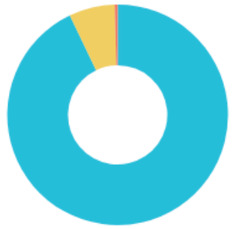
Based on the time window and filters you've selected



IE	954.946 k	89.36 %
Chrome	79.4 k	7.43 %
Safari	27.452 k	2.57 %
Firefox	3.908 k	0.37 %
IE Mobile	2.894 k	0.27 %

Top device types where this error occurred

Based on the time window and filters you've selected

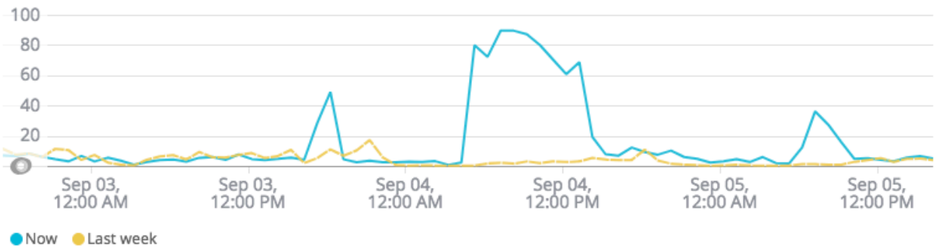


Desktop	993.077 k	92.87 %
Mobile	71.743 k	6.71 %
Tablet	4.474 k	0.42 %
Unknown	7	0.00065 %

This group of errors is 26.92% of your overall error rate

Based on the time window and filters you've selected

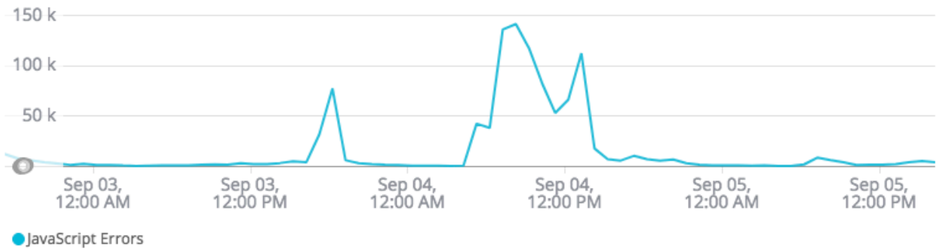
Error rate over time



This group of errors has 1.07 M instances

Based on the time window and filters you've selected

Page views with error (ppm)



In computer science, type safety is the extent to which a programming language discourages or prevents type errors.

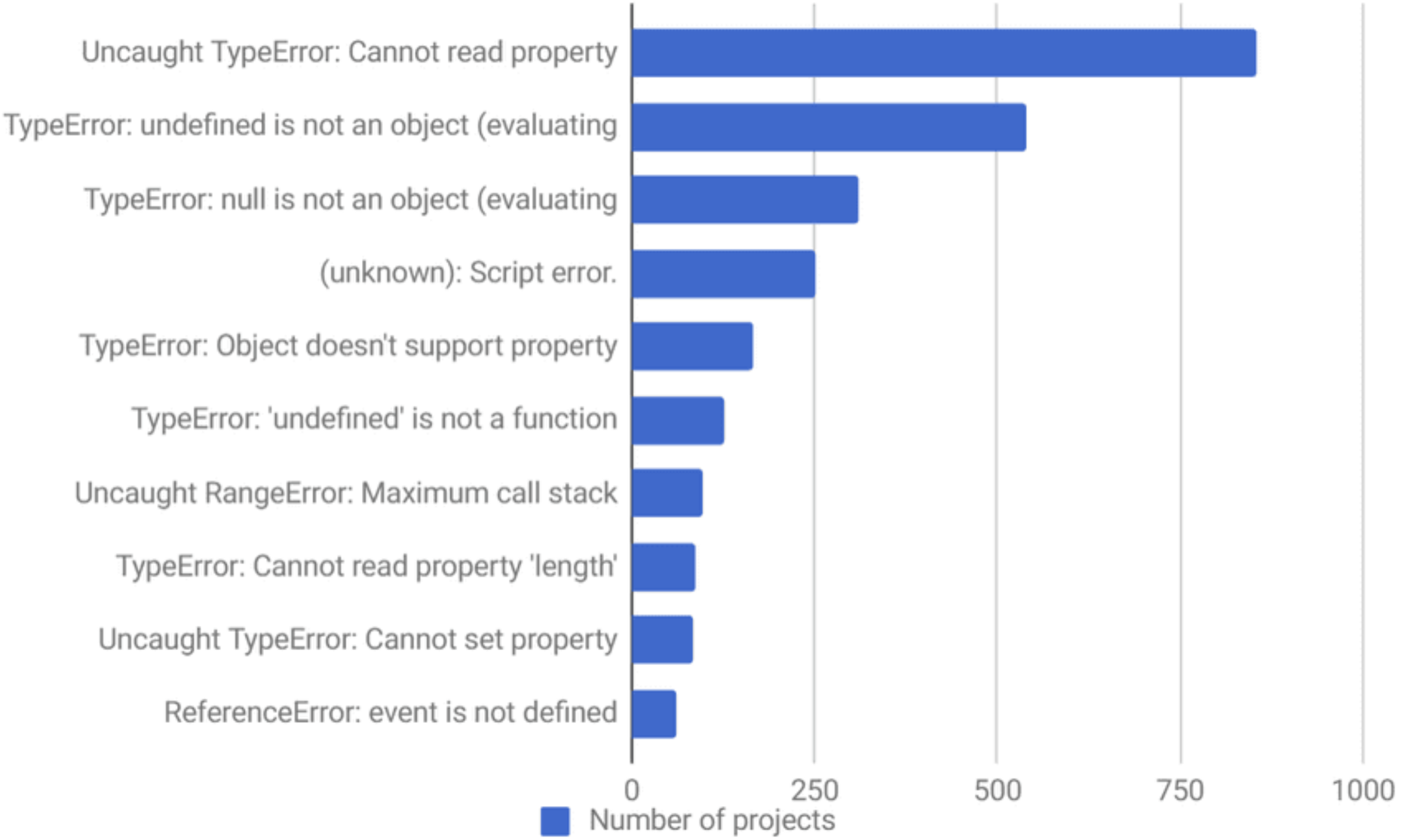
★

Wikipedia

The behaviors classified as type errors [..] are usually those that result from attempts to perform operations on values that are not of the appropriate data type.

★

Wikipedia


Uncaught TypeError: Cannot read property
TypeError: undefined is not an object (evaluating
TypeError: null is not an object (evaluating
(unknown): Script error.
TypeError: Object doesn't support property
TypeError: 'undefined' is not a function
Uncaught RangeError: Maximum call stack
TypeError: Cannot read property 'length'
Uncaught TypeError: Cannot set property
ReferenceError: event is not defined
0
250
500
750
1000
Number of projects

Static vs Dynamic; Strong vs Weak.[1]

[1] http://2ality.com/2013/09/types.html

Spot the difference

```
const a = null;
a.prop; // 1

const b = {};
b.prop; // 2
```

What happens at 1 and 2?

Spot the difference

```
const a = null;
a.prop; // 1

const b = {};
b.prop; // 2
```

Spot the difference

```
const a = null;
a.prop; // 1

const b = {};
b.prop; // 2
```

1. 💥 `TypeError: Cannot read property 'prop' of null` 💥

Spot the difference

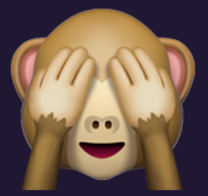
```
const a = null;
a.prop; // 1

const b = {};
b.prop; // 2
```

1. 💥 `TypeError: Cannot read property 'prop' of null` 💥
2. Silently fails, returning `undefined`

Which is interesting because...

```
> typeof null
'object'
```



Getting the first item in an array

What should we return?

What should we return?

- When the input is not an array?

What should we return?

- When the input is not an array?
- When the input is an empty array?

What should we return?

- When the input is not an array?
- When the input is an empty array?
- When the input is an array of > 0 items?

Null, undefined or false?

Accessing data nested in objects

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const sumC = data => {
  // what goes in here?
};
```

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const sumC = data => {
  // what goes in here?
};
```

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const sumC = data => {
  // what goes in here?
};
```

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const sumC = data => {
  // what goes in here?
};
```

A naive approach

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const add = (a, b) => a + b;
const sumC = data => data.a.b.c.reduce(add, 0);

sumC(a); // 6
sumC(b); // ?
```

TypeError: Cannot read property 'reduce' of null

How can we improve this code?

Add a guard clause

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const add = (a, b) => a + b;
const sumC = data => {
  if (data.a.b.c && Array.isArray(data.a.b.c)) {
    return data.a.b.c.reduce(add, 0);
  }

  // what do we return here?
};
```

How do we represent the failure branch in this function?

```
// option 1
return false;

// option 2;
return null;

// option 3;
throw new Error("An array is required to sum");

// option 3a
throw new CannotSumError("C should be an array");
```

```
// option 1
return false;

// option 2;
return null;

// option 3;
throw new Error("An array is required to sum");

// option 3a
throw new CannotSumError("C should be an array");
```

```
// option 1
return false;

// option 2;
return null;

// option 3;
throw new Error("An array is required to sum");

// option 3a
throw new CannotSumError("C should be an array");
```

```
// option 1
return false;

// option 2;
return null;

// option 3;
throw new Error("An array is required to sum");

// option 3a
throw new CannotSumError("C should be an array");
```

```
// option 1
return false;

// option 2;
return null;

// option 3;
throw new Error("An array is required to sum");

// option 3a
throw new CannotSumError("C should be an array");
```

If we are regularly accessing nested properties, how can we formalise this approach and make it reusable?

There has to be a

better way?

Pure functions

Consistent return types

Immutable data (structures)

Introducing

ADTs

Crocks

A collection of well known Algebraic Data Types for your utter enjoyment.

Get Started

The data types provided in Crocks allow you to remove large swaths of imperative boilerplate, allowing you to think of your code in terms of what it does and not how it does it.

Crocks

Sum Types to the rescue

First, a quick intro to Haskell-like type signatures

User Defined Types

```
data Bool = True | False
```

Type Alias / Synonyms

```
type EventId = Int
```

Function Signatures

```
add :: Int -> Int -> Int
```

~~Call~~ Get Me Maybe?

```
data Maybe a = Just a | Nothing
```

A solution using ADTs

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const add = (a, b) => a + b;
const sumC = data => {
  if (data.a.b.c && Array.isArray(data.a.b.c)) {
    return Just(data.a.b.c.reduce(add, 0));
  }

  Nothing();
};
```

A solution using ADTs

```
const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const add = (a, b) => a + b;
const sumC = data => {
  if (data.a.b.c && Array.isArray(data.a.b.c)) {
    return Just(data.a.b.c.reduce(add, 0));
  }

  Nothing();
};
```

Safely getting properties from an object

```
const prop = require("crocks/Maybe/prop");
const propPath = require("crocks/Maybe/propPath");

const data = { a: { b: { c: [1, 1, 2, 3, 5] } } };

const a = prop("a", data); // Just { b: { c: [1, 1, 2, 3, 5] } }
const b = prop("b", data); // Nothing

const c = propPath(["a", "b", "c"], data); // Just [1, 1, 2, 3, 5]
const d = propPath(["a", "b", "d"], data); // Nothing
```

Safely getting properties from an object

```
const prop = require("crocks/Maybe/prop");
const propPath = require("crocks/Maybe/propPath");

const data = { a: { b: { c: [1, 1, 2, 3, 5] } } };

const a = prop("a", data); // Just { b: { c: [1, 1, 2, 3, 5] } }
const b = prop("b", data); // Nothing

const c = propPath(["a", "b", "c"], data); // Just [1, 1, 2, 3, 5]
const d = propPath(["a", "b", "d"], data); // Nothing
```

Safely getting properties from an object

```
const prop = require("crocks/Maybe/prop");
const propPath = require("crocks/Maybe/propPath");

const data = { a: { b: { c: [1, 1, 2, 3, 5] } } };

const a = prop("a", data); // Just { b: { c: [1, 1, 2, 3, 5] } }
const b = prop("b", data); // Nothing

const c = propPath(["a", "b", "c"], data); // Just [1, 1, 2, 3, 5]
const d = propPath(["a", "b", "d"], data); // Nothing
```

Safely getting properties from an object

```
const prop = require("crocks/Maybe/prop");
const propPath = require("crocks/Maybe/propPath");

const data = { a: { b: { c: [1, 1, 2, 3, 5] } } };

const a = prop("a", data); // Just { b: { c: [1, 1, 2, 3, 5] } }
const b = prop("b", data); // Nothing

const c = propPath(["a", "b", "c"], data); // Just [1, 1, 2, 3, 5]
const d = propPath(["a", "b", "d"], data); // Nothing
```

Maybe is a Functor

Maybe is a Functor

A value which has a Functor must provide a `map` method.
The `map` method takes one argument

— Fantasyland Specification

```
map :: Functor f => f a ~> (a -> b) -> f b
```

Maybe is a Functor

```
const sumC = data =>
    propPath(["a", "b", "c"], data)
    .map(vals => vals.reduce(add, 0));
```

What Happens If We Get A Nothing?

What Happens If We Get A Nothing?

```
Just.prototype.map = function(fn) { return Just(fn(this.value)) };
Nothing.prototype.map = function(fn) { return this; };
```

What Happens If We Get A Nothing?

```
Just.prototype.map = function(fn) { return Just(fn(this.value)) };
Nothing.prototype.map = function(fn) { return this; };
```

What Happens If We Get A Nothing?

```
Just.prototype.map = function(fn) { return Just(fn(this.value)) };
Nothing.prototype.map = function(fn) { return this; };
```

Currying, Composition & Pointfree style

```
// getC :: Object -> Maybe a
const getC = propPath(["a", "b", "c"]);

// sum :: Array Number -> Number
const sum = reduce(add, 0);

// sumC :: Object -> Maybe Number
const sumC = pipe(getC, map(sum));
```

Currying, Composition & Pointfree style

```
// getC :: Object -> Maybe a
const getC = propPath(["a", "b", "c"]);

// sum :: Array Number -> Number
const sum = reduce(add, 0);

// sumC :: Object -> Maybe Number
const sumC = pipe(getC, map(sum));
```

Currying, Composition & Pointfree style

```
// getC :: Object -> Maybe a
const getC = propPath(["a", "b", "c"]);

// sum :: Array Number -> Number
const sum = reduce(add, 0);

// sumC :: Object -> Maybe Number
const sumC = pipe(getC, map(sum));
```

Currying, Composition & Pointfree style

```
// getC :: Object -> Maybe a
const getC = propPath(["a", "b", "c"]);

// sum :: Array Number -> Number
const sum = reduce(add, 0);

// sumC :: Object -> Maybe Number
const sumC = pipe(getC, map(sum));
```

Using a Monoid

Using a Monoid

Using a Monoid

- Monoids allow us to represent binary operations and are usually locked down to a specific type

Using a Monoid

- Monoids allow us to represent binary operations and are usually locked down to a specific type
- They are great when you need to combine a list of values down to one value

Using a Monoid

- Monoids allow us to represent binary operations and are usually locked down to a specific type
- They are great when you need to combine a list of values down to one value
- Handily, Crocks provides a standard set of useful of Monoids!

Using a Monoid

One of the Monoids included with Crocks is the Sum type

```
// Instead of this
const add = (a, b) => a + b;
const sum1 = reduce(add, 0);

// We can use the built in behaviours of a Monoid
const sum2 = mreduce(Sum);
```

💡 Note that sum1 and sum2 are equivalent - both take an array of numbers and return a number

Updating our example

```
const { Sum, map, mreduce, pipe, propPath } = require("crocks");

const a = { a: { b: { c: [1, 2, 3] } } };
const b = { a: { b: { c: null } } };

const sumC = pipe(propPath(["a", "b", "c"]), map(mreduce(Sum)));

sumC(a); // Just 6
sumC(b); // Nothing
```

mreduce vs mconcat vs mreduceMap vs mconcatMap

Jumping back...

Getting the first item in an array

Getting the first item in an array

```
// head :: Array a -> Maybe a
const head = arr => {
  if (Array.isArray(arr) && arr.length > 1) {
    return Just(arr[0]);
  }

  return Nothing;
};
```

What if we need to represent more than just "Nothing"?

Maybe is just the start of this adventure...

```
data Either e a = Left a | Right b
```

```
data Result e a = Err e | Ok a
```

```
data Async e a = Rejected e | Result a
```

```
Data Pair a b = Pair a b
```

```
data RemoteData e a = NotAsked | Loading | Error e | Success a
```

This all sounds great, but why the crazy names?

The specifications in this list do not derive from goals such as trying to write rules for lists and maps. Instead, they start by noticing rules that apply in common to disparate structures.

★

https://github.com/fantasyland/fantasy-land/issues/127#issuecomment-192763058

Aside, why Fantasy Land?

Comment re: fantasy land

github issue

Where to learn more

- My Accompanying Workshop
- Professor Frisby's Mostly Adequate Guide
- Professor Frisby's Guide To Functional Programming
- Fantas Eel And Specification
- Elm
- Evilsoft's YouTube Channel

Programming safely with types

Ian Thomas | @anatomic