The Unreasonable Effectiveness of TypeScript

TypeScript type-level programming

Running example: Type-level arithmetics

Setting expectations

- Practicality: 0%
- Time waste: 100%

```
1. Will learn something new? "Yes"
2. Is this something useful? "No"
_____________________________________________

∴  Conclusion: We will learn something useless.
```

If we learn something useless does it count as learning?

Is it fun? "For some"

Why? "Because we can!"

Disclaimer: You can ask me anything about TypeScript but don't expect an answer. I'm not an expert.

Setting ambitions

Topics to cover

- **What** - What is Type-Level Programming?
- **How** - How is it done?
- **Why** - Why TypeScript has it?

Table of contents

To hide the lack of real content I'll try to cover it with random unrelated memes 🤦

What is type-level programming?

Normal types

Add two numbers

```
declare function sumStr(a: string, b: string): number;

const x = sumStr("23", "32");
```

Sort array of number

```
declare function sortStr(xs: string[]): number[];

const xs = ["3", "1", "2", "0"];
const sortedXs = sortStr(xs);
```

What is type-level programming?

Make type-system calculate the result

But it's clear `x` could must be `55` and `sortedXs` must be `[1, 2, 3]`.
Can we make type-system to be a little bit more precise?
Calculate the result and tell us what `x` and `sortedXs` should be ?
All *before* running the program, purely relying on a type-checker? Yes, we can!

```
const x = sumStr("23", "32");
//    ^? const x:55
//
//
```

```
declare function sumStr<A extends string, B extends string>(
  a: A,
  b: B
): AddStr<A, B>;
```

```
const xs = ["3", "1", "2"] as const;
const sortedXs = sortStr(xs);
//    ^? sortedXs = [1, 2, 3];
```

```
declare function sortStr<A extends readonly string[]>(xs: A): SortNumberStr<A>;
//
//
```

Without ever running the program we got the results. This might just make TypeScript the fastest language on Earth, with a runtime of precisely zero. ☺

How `SortNumberStr<A>` type look like?

```
type SortNumberStr<Xs extends readonly string[]> = MapToNumbers<
  MergeSort<MapFromStrings<Mutable<Xs>>>
>;
```

```
type MergeSplit<
  T extends BinaryNumber[],
  Xs extends BinaryNumber[] = [],
  Ys extends BinaryNumber[] = [],
> = T extends [infer X]
  ? [[...Xs, X], Ys]
  : T extends [
        infer X extends BinaryNumber,
        infer Y extends BinaryNumber,
        ...infer Rest extends BinaryNumber[],
      ]
    ? MergeSplit<Rest, [...Xs, X], [...Ys, Y]>
    : [Xs, Ys];

type Merge<Xs extends BinaryNumber[], Ys extends BinaryNumber[]> = Xs extends []
  ? Ys
  : Ys extends []
    ? Xs
```

How is it done?

TypeScript = JavaScript + Types

TypeScript *is* JavaScript.

> TypeScript is JavaScript with syntax for types. (ref https://www.typescriptlang.org/)

TypeScript is designed to

- type-check JavaScript programs
- provide better language tooling for JavaScript programs
- ensure correctness of JavaScript programs

TypeScript *is not (meant to be)*

- a new language
- a compiler
- a JavaScript replacer

Types as a functional programming language

JavaScript + Type System = TypeScript
~~JavaScript~~ + Type System = Type-level TypeScript

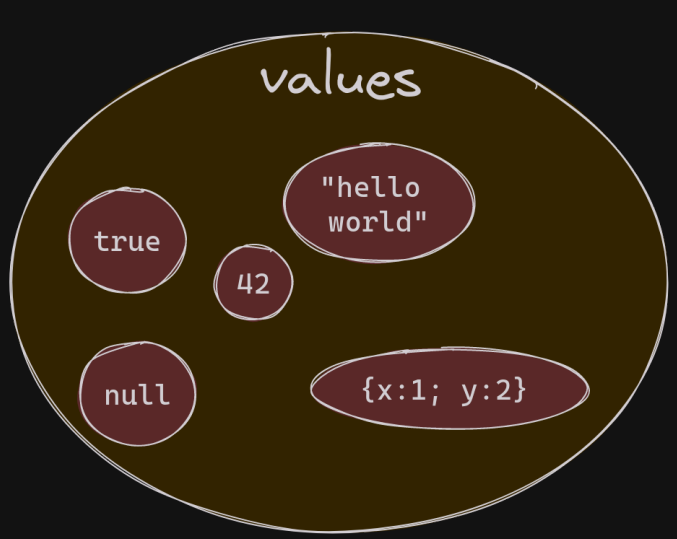
TypeLang - the language of types.

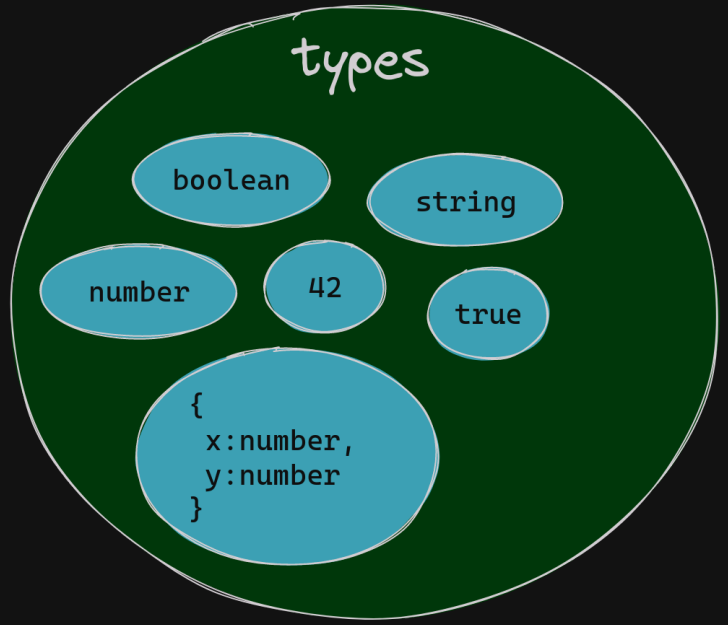
TypeLang is an immutable, pure, lazy, functional programming language.

- functional - functions all we have, recursion is our friend (no higher order functions though)
- pure - no input/output
- immutable - we can not re-assign types once they already declared
- lazy - types are evaluated as needed. Opposed to JS eager evaluation.

TypeLang Values

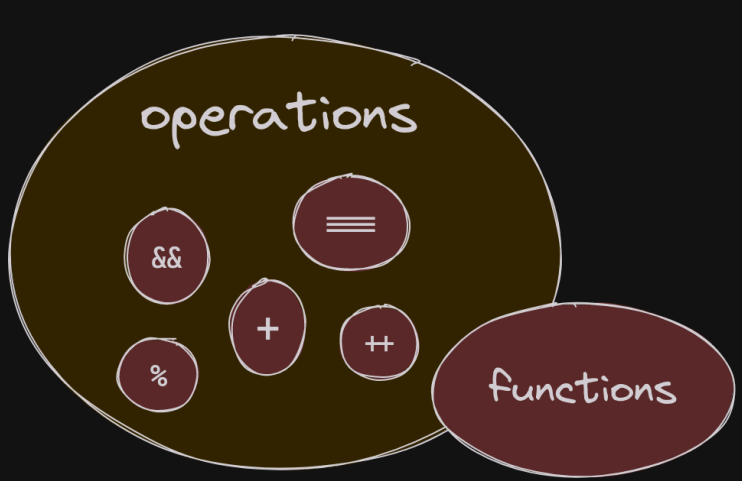
Things the language can make use of

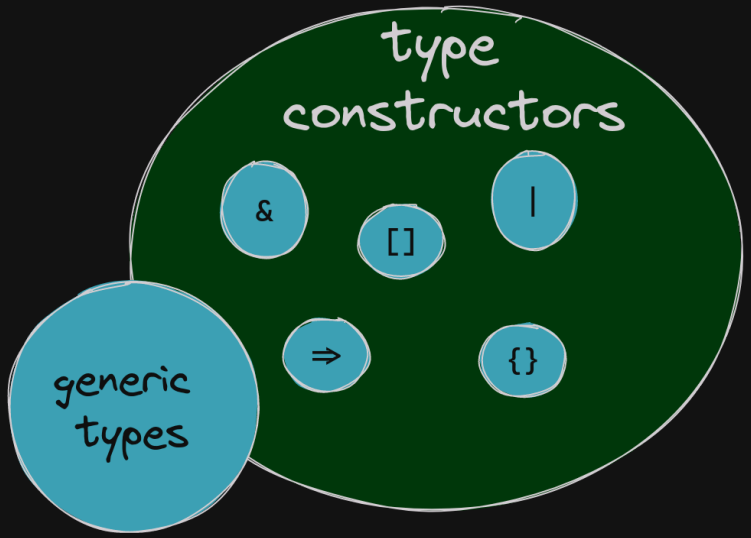




TypeLang Operations

A set of available operations defines what we can do with values



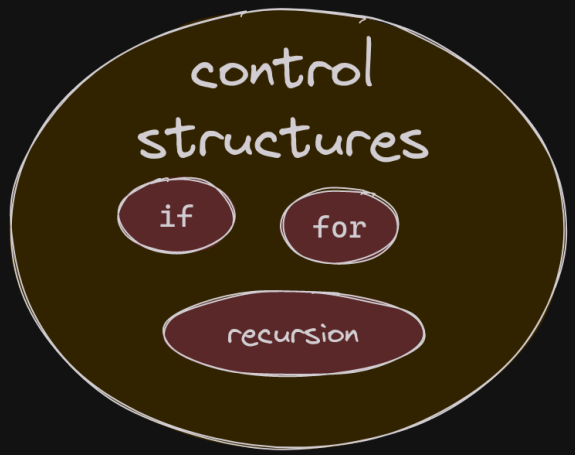


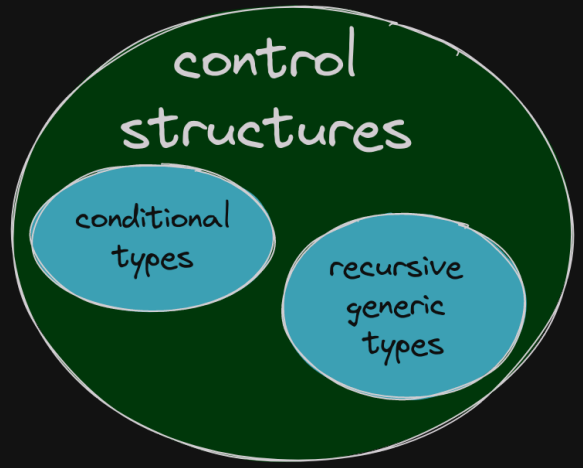
On value-level there are values and operations which operate on those values - take values as parameters and return values.

On type-level there are types and type constructors which operate on those types - take types as parameters and return types.

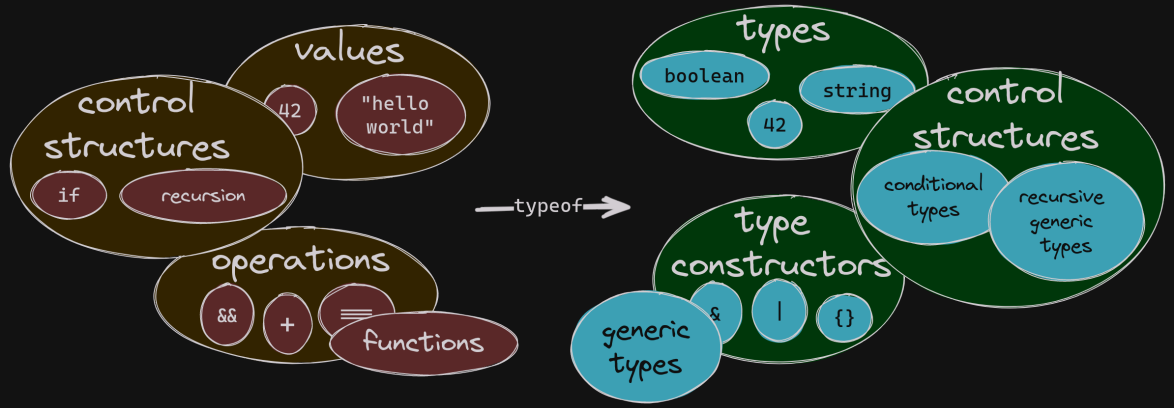
Operations on types are different from the ones we have in JS. We only have type operations which can make sense on types.

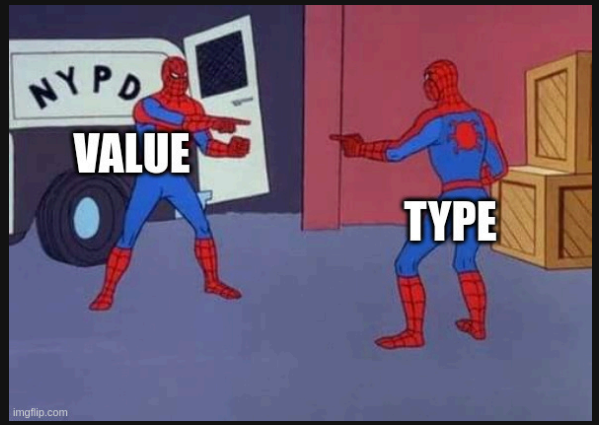
TypeLang Control structures





Value-level (JavaScript) *vs* Type-level (TypeLang)





TypeLang values - types

- Primitive types

```
type B = boolean;
type S = string;
type X = Array<number>;
```

- *Literal types*. TypeScript unique ?

```
// literal values
const someNum = 37;
const s = "hello";

// literal types
type SomeNum = 37;
type S = "hello";
```

TypeLang Operations - Type constructors

- union `|`
- intersection `&`
- arrays `number[]`
- tuples `[string, boolean]`
- objects `{key1: string; key2: boolean}`
- records `{[key: string]: number}`
- function type `⇒`

  Could think about type operations as data structures for types. We can put other types in and extract them out.

Union

Constructor:

```
type T = boolean | string;
```

Intersection

Constructor:

```
type Signal = "red" | "yellow" | "green";
type Ok = "green" | "white";

type T = Signal & Ok;
```

Tuples

Constructor:

```
type T = [boolean, string];

type T = [first: boolean, second: string]; //tuple elements could have names
type T = [boolean, string?]; //tuple elements could be optional
```

Access:

```
type E = T[0 | 1]; // extracting type of individual elements
type E = T[number]; // of all elements
type E = T["length"]; // get tuple length
```

Operations on tuples: Concatenate:

```
type E = [ ...T1, ...T2]; // merge tuples

//Tuples and arrays can be combined in the same constructor.
type T = [number, ...number[]]; // non-empty array of numbers
```

Array

Constructor:

```
type T = boolean[];
```

Access:

```
type E = T[number]; //extract type of an array values
type E = T[0]; // same
```

Objects

Constructor:

```
type T = { key1: string; key2: boolean };
```

Access:

```
type T = { key1: string; key2: boolean };

type E1 = T["key1" | "key2"]; // get property types
type E2 = keyof T; // get property keys type
type E3 = T[keyof T]; // shortcut to get all possible property keys type
```

Records

Constructor:

```
type T = { [key: string]: boolean }

type T = Record<Keys, Type>
type T = Record<K, V> = { [Key in K]: V }
```

Access:

```
type E = T[string]; // get property value type. Not possible Object constructor
type E = keyof T; // get property keys type
```

Function type

Constructor:

```
type T = (input: number) => boolean;
```

Access:

```
type P = Parameters<T>;
type R = ReturnType<T>;
```

Template type literals

Constructor:

```
type A = "hello";
type B = 37;
type T = `${A} foo ${B}`;
```

Access by pattern matching

```
type E = `a foo 37` extends `${infer A} foo ${infer B extends number}`
  ? [A, B]
  : never;
```

Absolute unique. Together with conditional recursive types we can iterate over strings.

```
type MakeMeKebab<S> = S extends `${infer First} ${infer Rest}`
  ? `${First}-${MakeMeKebab<Rest>}`
  : S;

type E = MakeMeKebab<"tomato cheese anchovy cheese">;
```

be amazed: https://github.com/codemix/ts-sql

TypeLang Control structures

- Functions
- Conditionals (if-then expressions)
- Pattern matching
- Recursion (loops)

TypeLang Functions

Generic types are equivalent of function calls.

TypeLang functions are generic types. They take types as inputs and produce types as outputs.

```
// value level
const myFunc = (a, b) ⇒ a + b;

const x = myFunc(1, 2);
```

```
// type level
type MyFn<A, B> = A | B;

type X = MyFn<1, 2>;
```

TypeLang Functions

Defaults.

```
type Fn<A, B = null> = [A, B];
```

Type constraints

```
type Fn<A extends string, B = null> = { [Key in A]: B };
```

It's possible to specify a type of an input. A type of a type?!

Here `A extends string` defines what `A` could be - `A` should be assignable to `string`.
(Plays similar role to classes constraints in Haskell.)

Constraints are also helpful to force TS to infer subtypes of a type parameter - forcing type narrowing.

```
//without constraint
const asObj = <S>(value: S) => ({ value });
const obj = asObj("hello");
```

```
// with constraint
const asObj = <S extends string>(value: S) => ({ value });
const obj = asObj("hello");
```

TypeLang Conditionals (if-then expressions)

Conditional type:

```
type X = A extends B ? T : F;
```

example:

```
type A = "hello";
type B = string;

type X = A extends B ? true : false;
```

Equivalent JS conditional expression:

```
const x = A ⩽ B ? true : false;
```

Since type-level typescript is a functional programming language, there are no conditional statements, only conditional type expressions - each type constructor returns a type.

TypeLang Condition checks for a subtype relationship

```
type X = A extends B ? T : F;
```

The `A extends B` defines A and B subtype relationship - A is a subtype of B.

```
// NOT REAL SYNTAX. MEANING OF THE `extends` KEYWORD
type T = A is a subtype of B ? true : false
type T = A is a subset of B ? true : false
type T = A is assignable to B ? true : false
```

What is a subtype

- `A` is a subtype of `B` (`B` is a supertype of `A`) *A formal definition*
- `A` can be used in place of `B` *a formal meaning of a formal definition*
- `A` is a subset of `B` (a set of values of `A` is a subset of values of `B` . `B` includes all values of `A`)
- `A` is "smaller" then `B`
- `A` implements `B`
- `A` extends `B`
- `A` can be assigned to `B`
- `A` can be used where B is expected
- `A <: B` , `A ⊑ B` , `A ≤: B` , `A ⊆ B` , `A → B`

Special cases

A caveat:

```
type T = never extends number ? true : false;
```

conditional type will be evaluated to `never`. Even though `never` is a subtype of `number` it is treated specially, and conditional with `never` on the left of `extends` will always be evaluated to `never`.

Example:

```
type If<A extends boolean, T, F> = A extends true ? T : F;
type E = If<true, 42, string>; // 42
```

IsEqual

`A extends B` define subtype relationship. Not Equality. TypeScript doesn't provide a way to check if two types are equal. To check if two types are equal we need to check if

`A` is a subtype of `B`
and
`B` is a subtype for `A`

```
type Extends<A, B> = A extends B ? true : false;

type Equal<A, B> = [Extends<A, B>, Extends<B, A>] extends [true, true]
  ? true
  : false;
```

Approximation, still wrong on some cases.

TypeLang Pattern matching

TypeLang has pattern matching. Ironically JS itself still doesn't.

Pattern matching: Checking if type conforms to some pattern (another type) and deconstructing the type according to the pattern. Allows to extract the values from type constructors.

Patter match object

```
// Pattern match `T` against pattern with shape `{ id: string, status: _something_ }`
type GetStatus<T> = T extends { id: string; status: infer S } ? S : never;

type E = GetStatus<{ id: "abc"; status: "ok" }>;
```

TypeLang Pattern matching

Patter match tuple

```
// Pattern match `T` against a tuple.
type Head<T> = T extends [infer H, ...unknown[]] ? H : never;
type Last<T> = T extends [...unknown[], infer Last] ? Last : never;

type E1 = Head<[1, 2]>;
type E2 = Last<[1, 0, 2]>;
```

TypeLang Pattern matching

Patter match function

Functions are type constructors also, so pattern match works

```
type FromFunc<T> = T extends (arg: infer A) => infer R
  ? { input: A; output: R }
  : never;

type E = FromFunc<(i: string) => number>;
```

TypeLang Pattern matching

Patter match user-defined type constructor

Works with user-defined type constructors aka Generic Types

```
type Thing<Id, Name> = { id: Id; name: Name }

// Pattern matching against Thing generic type
type GetId<T> = T extends Thing<infer Id, infer Name> ? Id : never;

type E = GetId<Thing<string, number> // string
```

TypeLang Pattern matching

A trick to get intermediate results from a computation

```
// instead of
type Fn<I> = DoSomething<Calc<I>> & // calling Calc twice here
  DoSomethingElse<Calc<I>>; // and here

// Optimizing type level performance 🤯
// we can use "assign" calculation to intermediate variable
type Fn<X> =
  Calc<X> extends infer Y // get Y from calculation
    ? DoSomething<Y> & DoSomethingElse<Y> // use Result
    : never;
```

Equivalent of Haskell

```
f x =
   let y = ... x ...
   in  ... y ...
```

TypeLang Pattern matching

Trick to constrain an inferred type.

```
type FirstIfString<T> = T extends [infer S extends string, ...unknown[]]
  ? S
  : never;
```

`infer S extends string` matches against `S`, it also ensures that `S` has to be a string. If `S` isn't a string, it takes the false path, which in these cases is never.

TypeLang Recursion (loops)

Recursive generic types + conditional types (aka Recursive Conditional Types)

```
type FnRec<Input> = // recursive function
  Input extends Some // condition
    ? FnRec<DoSomething<Input>> // recursive case
    : BaseCase; // base case
```

Side note: Following functional language tradition TypeScript performs Tail-Call Optimization on Recursive Conditional Types 🤯

JavaScript itself still don’t

TypeLang Recursion (loops)

Looping over lists

```
type Fn<List> = List extends [infer First, ...infer Rest] ? Fn<Rest> : BaseCase;
```

Example

```
type Contains<List, X> = List extends [infer First, ...infer Rest]
  ? First extends X
    ? true
    : Contains<Rest, X>
  : false;

type E1 = Contains<[1, "two", false, 4], 4>;
type E2 = Contains<[1, "two", false, 4], boolean>;
type E3 = Contains<[1, "two", false, 4], 37>;
```

TypeLang is Lazy

```
halt = (c) => (c ? 42 : halt(c));
halt(true); // evaluates to ‘42’
halt(false); // goes infinite

ifFunc = (c, t, f) => (c ? t : f);
ifFunc(true, "yes", "no"); // return yes
ifFunc(true, "yes", halt(false)); // the answer is 'yes' but we never get it

type Halt<C> = C extends true ? 42 : Halt<C>;

type If<C, T, F> = C extends true ? T : F;

type Eval = If<true, "yes", Halt<false>>; // this evaluates to 'yes', cause TS types are lazy evaluated
```

Type level binary arithmetics

Binary arithmetics with logic gates

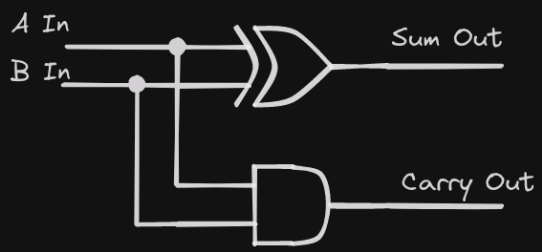
```
A   B   Sum
0 + 0 = 0 0
0 + 1 = 0 1
1 + 0 = 0 1
1 + 1 = 1 1
```

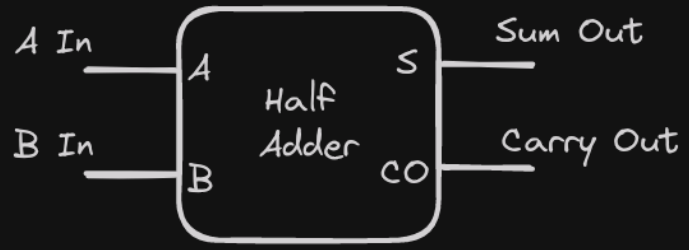
```
(A)    1 1 1 1
(B)   +1 1 1 0
      ---------
(S) 1 1 1 0 1
```

```
(CO)    1 1
(A)     1 1 1 1
(B)   + 1 1 1 0
      ---------
(S)   1 1 1 0 1
```

Half adder

A In	B In	Sum	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

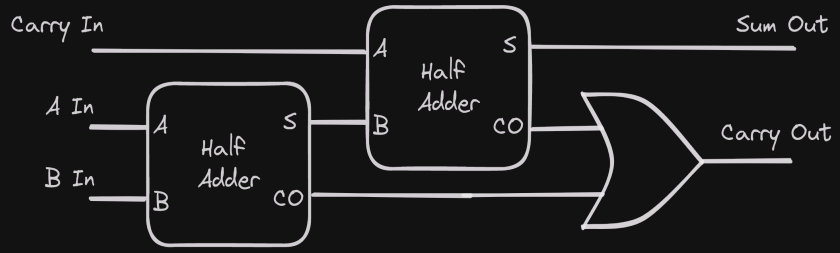


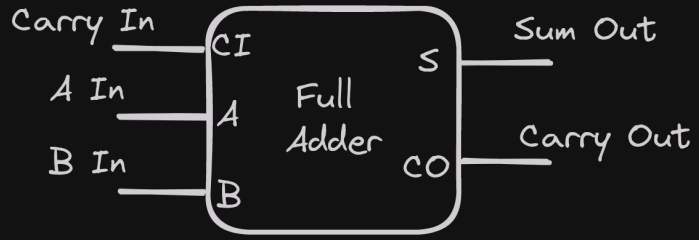



```ts
// Gates
type Bit = 0 | 1;
type Not<A> = A extends 0 ? 1 : 0;
type Or<A, B> = [A, B] extends [0, 0] ? 0 : 1;
type And<A, B> = [A, B] extends [1, 1] ? 1 : 0;
type Xor<A, B> = [A, B] extends [1, 0] | [0, 1] ? 1 : 0;
type NAnd<A, B> = Not<And<A, B>>;
```

```ts
// half adder
type HalfAdder<A, B> = [S: Xor<A, B>, CO: And<A,B>]
```

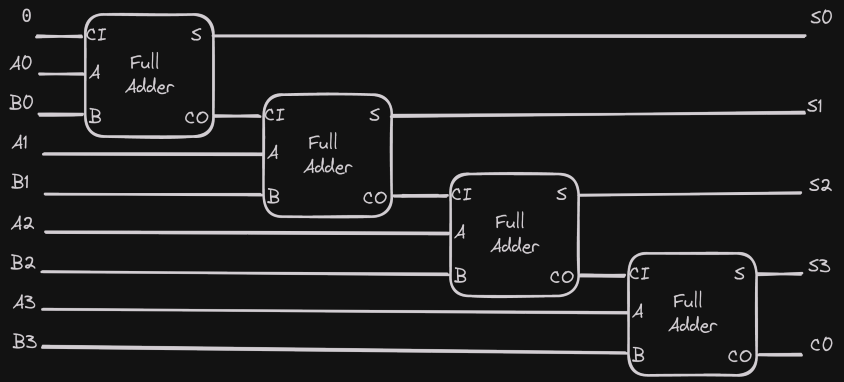
Full adder

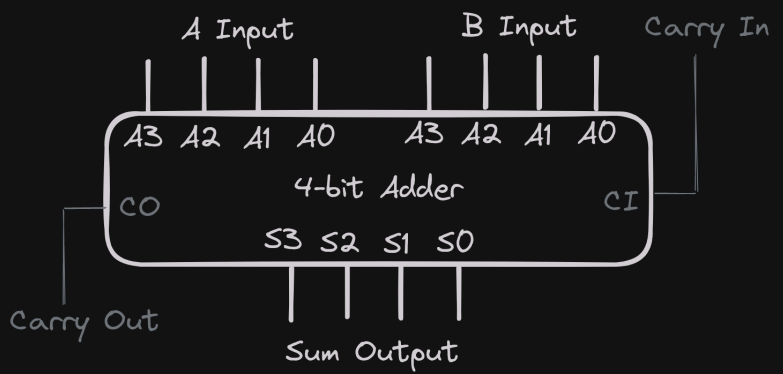





```
type FullAdder<A, B, CI> =
  HalfAdder<A, B> extends [infer HA1S, infer HA1CO]
    ? HalfAdder<CI, HA1S> extends [infer HA2S, infer HA2CO]
      ? [S: HA2S, CO: Or<HA1CO, HA2CO>]
      : never
    : never;
```

4-bit adder






```ts
type BinaryNumber = Array<Bit>;

type LastBit<T extends BinaryNumber> = T extends [...any, i
  ? Last
  : 0;
type Size<T extends BinaryNumber> = T["length"];

type Head<T extends any[]> = T extends [...infer Head, any]

// Adder
type Adder<
  A extends BinaryNumber,
  B extends BinaryNumber,
  CI extends Bit = 0,
> = [Size<A>, Size<B>] extends [0, 0]
  ? []
  : FullAdder<LastBit<A>, LastBit<B>, CI> extends [
      infer S,
      infer CO extends Bit,
    ]
    ? [...Adder<Head<A>, Head<B>, CO>, S]
    : never;
```

Merge sort

```
type MergeSort<Xs extends Array<BinaryNumber>> = Xs["length
  ? Xs
  : MergeSplit<Xs> extends [
       infer Ls extends Array<BinaryNumber>,
       infer Rs extends Array<BinaryNumber>,
     ]
    ? Merge<MergeSort<Ls>, MergeSort<Rs>>
    : never;
```

Haskell

```
mergeSort :: (Ord a) ⇒ [a] → [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs =
  let [ls, rs] = mergeSplit xs
  in merge (mergeSort ls) (mergeSort rs)
```

Merge split

```
type MergeSplit<
  T extends BinaryNumber[],
  Xs extends BinaryNumber[] = [],
  Ys extends BinaryNumber[] = [],
> = T extends [infer X]
  ? [[...Xs, X], Ys]
  : T extends [
      infer X extends BinaryNumber,
      infer Y extends BinaryNumber,
      ...infer Rest extends BinaryNumber[],
    ]
    ? MergeSplit<Rest, [...Xs, X], [...Ys, Y]>
    : [Xs, Ys];
```

Haskell

```
mergeSplitAcc [] xs ys = [xs, ys]
mergeSplitAcc [x] xs ys = [x:xs, ys]
mergeSplitAcc (x:y:rest) xs ys =
  mergeSplitAcc rest (x:xs) (y:ys)

mergeSplit xs = mergeSplitAcc xs [] []
```

Merge

```
type Merge<Xs extends BinaryNumber[], Ys extends BinaryNumb
  ? Ys
  : Ys extends []
    ? Xs
    : [Xs, Ys] extends [
          [infer X extends BinaryNumber, ...infer Xs extenc
          [infer Y extends BinaryNumber, ...infer Ys extenc
        ]
      ? Compare<X, Y> extends "LT"
        ? [X, ...Merge<Xs, [Y, ...Ys]>]
        : [Y, ...Merge<[X, ...Xs], Ys>]
      : never;
```

Haskell

```
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) =
   if x < y
   then x:merge xs (y:ys)
   else y:merge (x:xs) ys
```

Why TypeScript has it?

What is JavaScript ?

Scheme + Self + C + Java = JavaScript

When JavaScript joined programming language community

- Scheme - functions
- Self - prototype based
- C - syntax
- Java - name

New. Fun. Different. Fascinated. Flexible. Weird. Refreshing. Cool. Awkward. Easy-going.

Well suited to put some scripts on a web page

And... very hard to write large scale applications

TypeScript

Scheme + Self + C + Java = JavaScript

Two very dynamic languages. Both are very flexible and permissive. So as the offspring, JS. Good luck with putting type over it.

JavaScript + Types

We got you covered!

TypeScript = JavaScript + Types

TypeScript *is* JavaScript.

> TypeScript is JavaScript with syntax for types. (ref https://www.typescriptlang.org/)

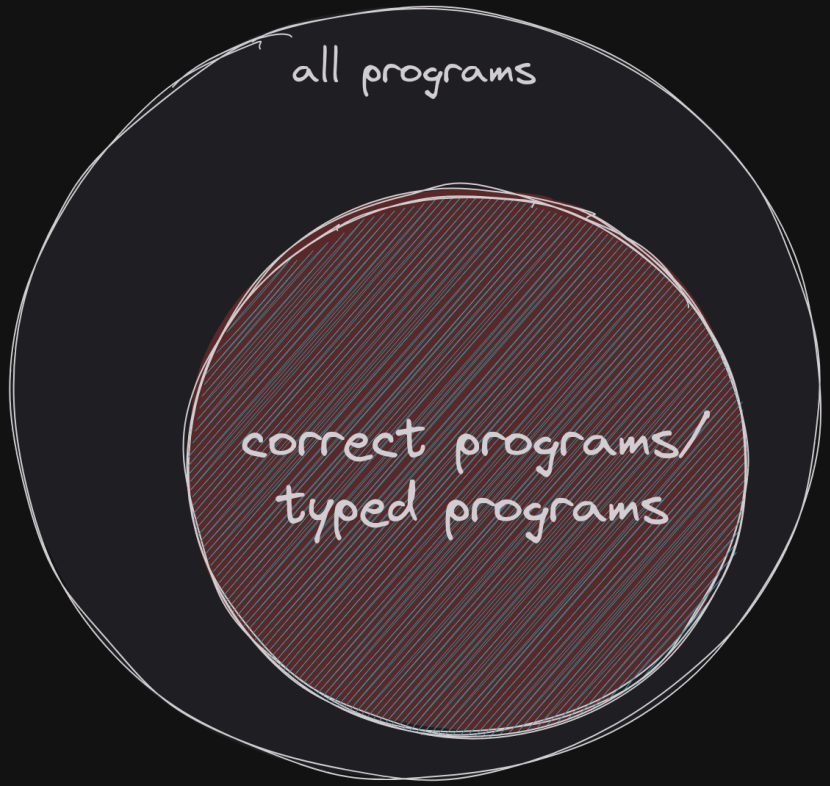
TypeScript is designed to

- type-check JavaScript programs
- provide better language tooling for JavaScript programs
- ensure correctness of JavaScript programs

TypeScript *is not (meant to be)*

- a new language
- a compiler
- a JavaScript replacer

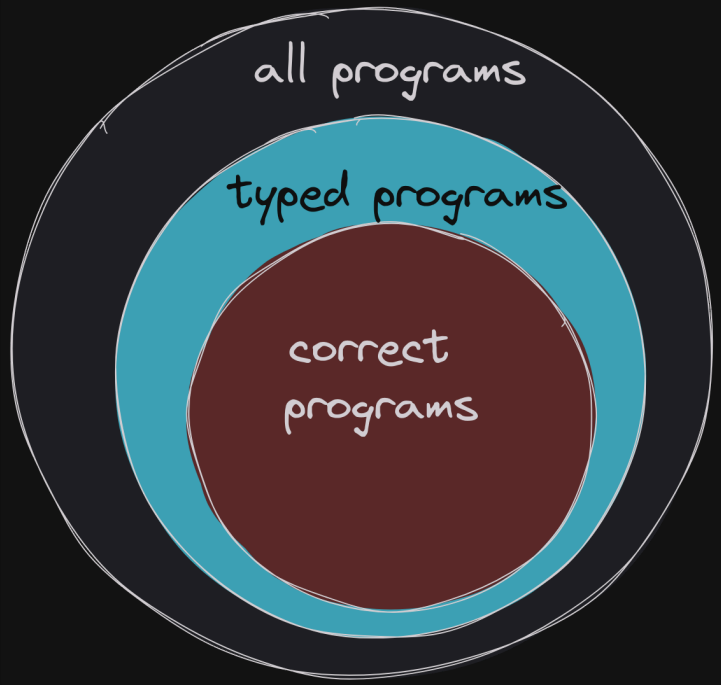
Any *other* statically typed languages (Think Java, Go, C#, Rust, Haskell, etc)

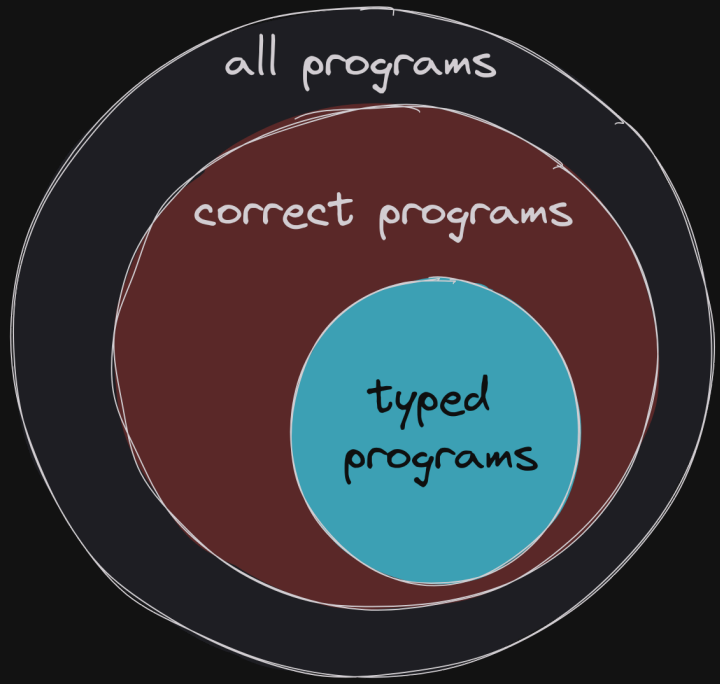


- Correct programs - can run and produce expected result
- Typed programs - can be type-checked and compiled

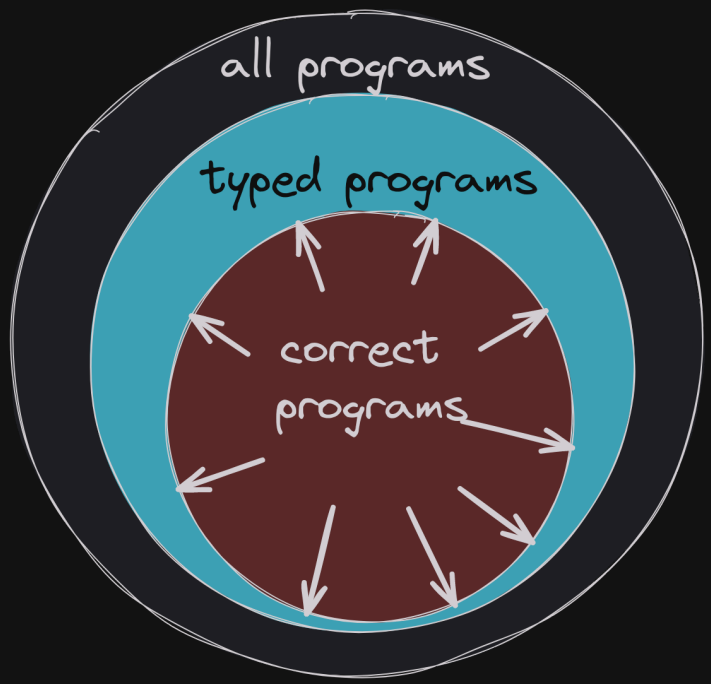
We only can write programs within the boundaries of type system. Type system limits what we can do.

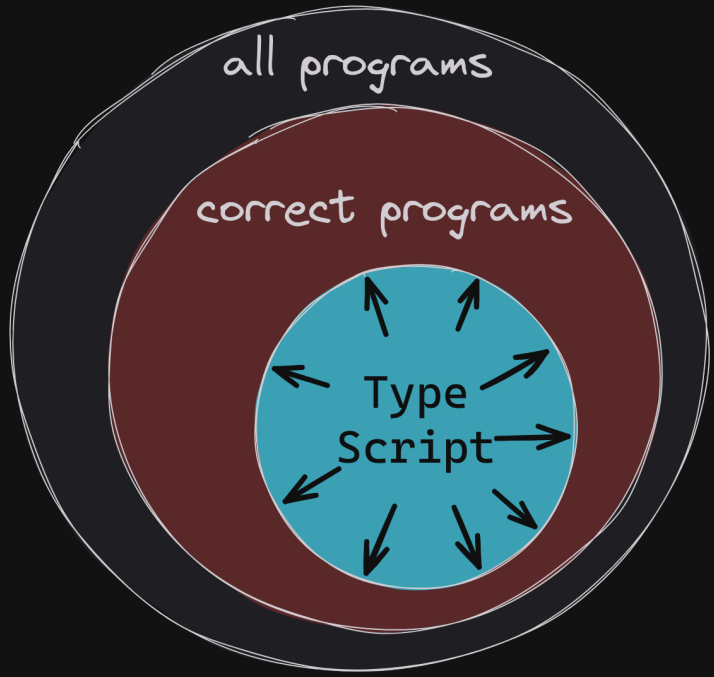
Static vs Optional type

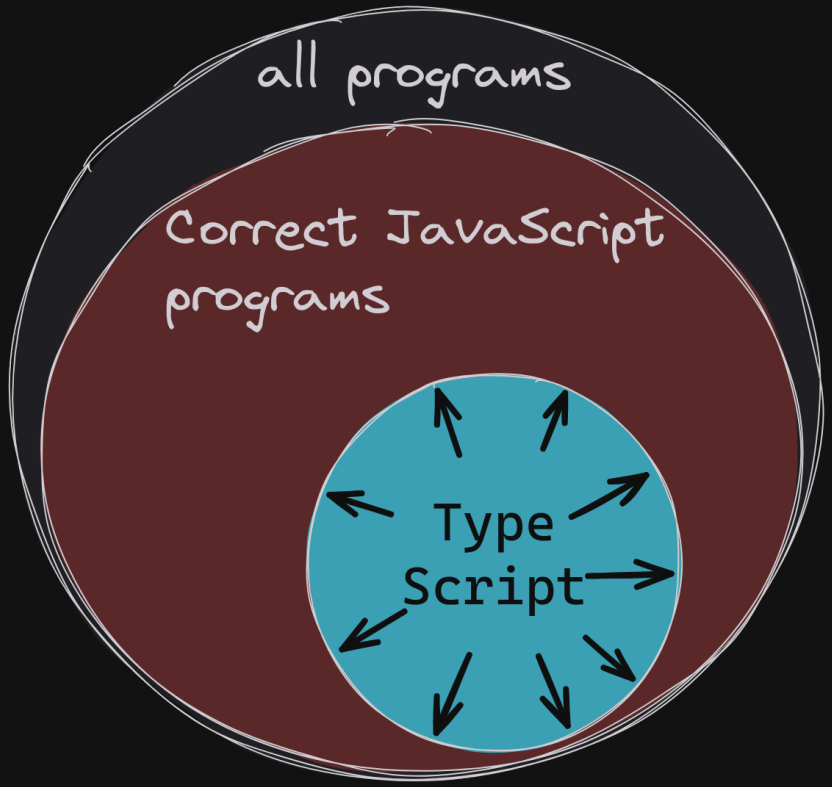




Static vs Optional type developer experience







JS is a highly dynamic, flexible language and people took advantage of that. There is an inherent tension between JS flexibility and type system rigidity.

For some *correct* JS programs we can *not* provide types.

In JS you can write all kind of programs, outside what is possible to type.
TS type system might not be able to capture all possible correct programs.

TS tries to push the boundaries of typed programs.
New TS versions allow to describe correctness of a richer set of correct programs.

TS is a set of compromises.
Sometimes when trying to expand in one direction you will contract in others.

Is it all roses?

It is great and unique

- Structural types
- Literal types
- Conditional types
- Recursive generics

Also complicated!

Are TypeScript types having it's C++ moment? It becomes unwieldy complicated and unmanageable?

Why they developed such a complex type system?

- The development stems from the JS usage patterns.
- It reflects the flexibility of JS.
- The problem TS is trying to solve - how to statically type a dynamically typed language with a great variety of usage patterns!

A couple of type definitions from Redux library.

```
export type ValidateSliceCaseReducers<
  S,
  ACR extends SliceCaseReducers<S>,
> = ACR & {
  [T in keyof ACR]: ACR[T] extends {
    reducer(s: S, action?: infer A): any;
  }
    ? {
        prepare( ...a: never[]): Omit<A, "type">;
      }
    : {};
};

type SliceDefinedCaseReducers<CaseReducers extends SliceCas
  [Type in keyof CaseReducers]: CaseReducers[Type] extends
    ? Definition extends AsyncThunkSliceReducerDefinition<a
      ? Id<
          Pick<
            Required<Definition>,
            "fulfilled" | "rejected" | "pending" | "settlec
          >
        >
      : Definition extends {
          reducer: infer Reducer;
        }
```

Libraries are using complex types (both external and internal)

and there are no signs it will become easier

We often have to deal with obscure type errors. And it is not fun!

Opinion

Advice

Don’t develop in type system. It might seem clever when written but it's the same trap every time

Me trying to pushing complex type to production code

It doesn’t look good

Don’t do it!

It can be fragile

`any` keyword, type predicates, assertions, function overloads are **all lies**.

The carefully constructed type having my back while I code.

You can use types to help yourself

Just don’t overuse it. Use common sense

Let the types be with you