TypeScript type-level programming

Type level arithmetics

Table of contents

1. TypeScript type-level programming
2. Table of contents
    1. What is Javascript
    2. TypeScript
3. Types as a functional programming language
    1. Values
    2. Operations
    3. Control structures
    4. Value vs type
    5. Lazy evaluated
4. Type level binary arithmetics
    1. Binary arithmetics with logic gates
    2. Half adder
    3. Full adder
    4. 4-bit adder
    5. Merge sort

What is Javascript

- Scheme + Self + C + Java = JavaScript

TypeScript

Scheme + Self + C + Java = JavaScript

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

JavaScript + Types

Types as a functional programming language

TypeScript = JavaScript + Types

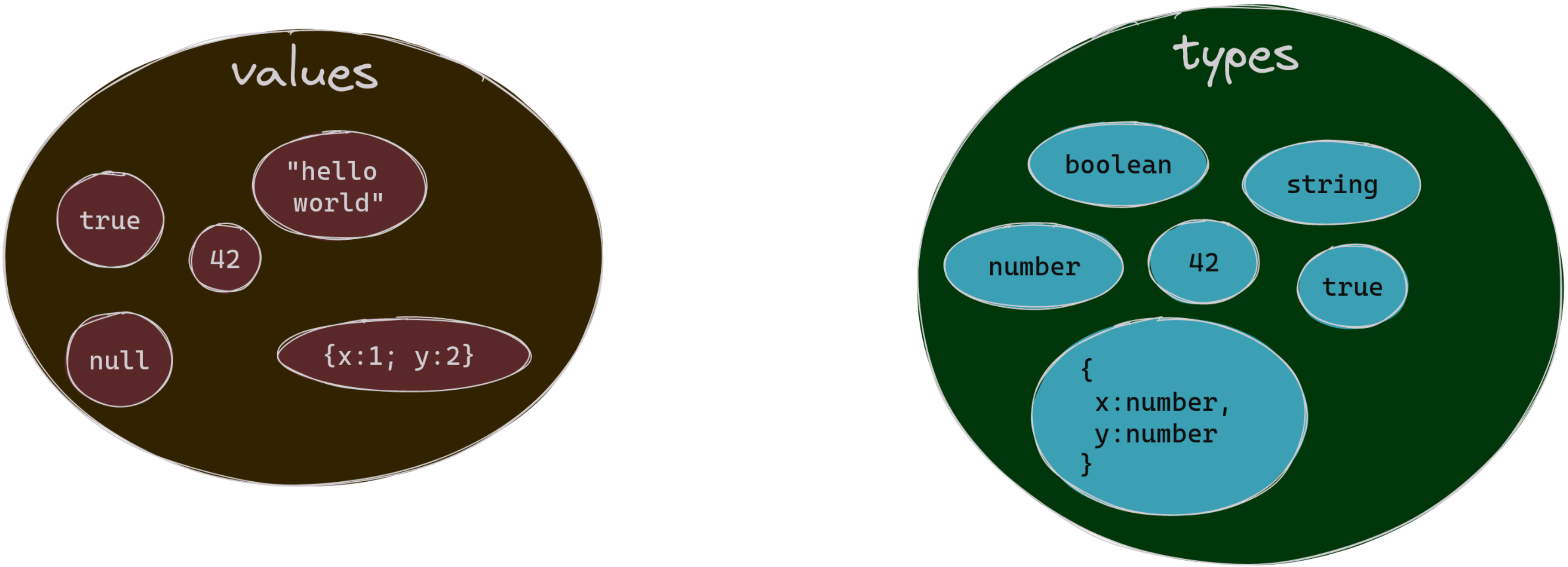
It *is* JavaScript, just with added Types. And Types is a language also. Two for the price of one!

Type Level Type Script = ~~JavaScript~~ + Types

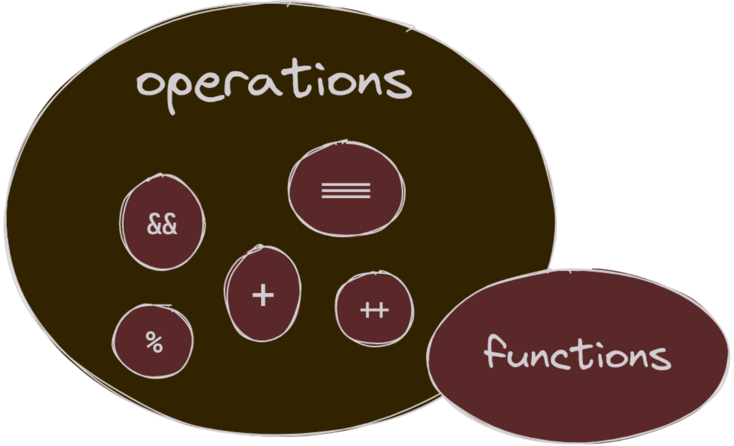
TLTS is a immutable, pure, lazy, functional programming language.

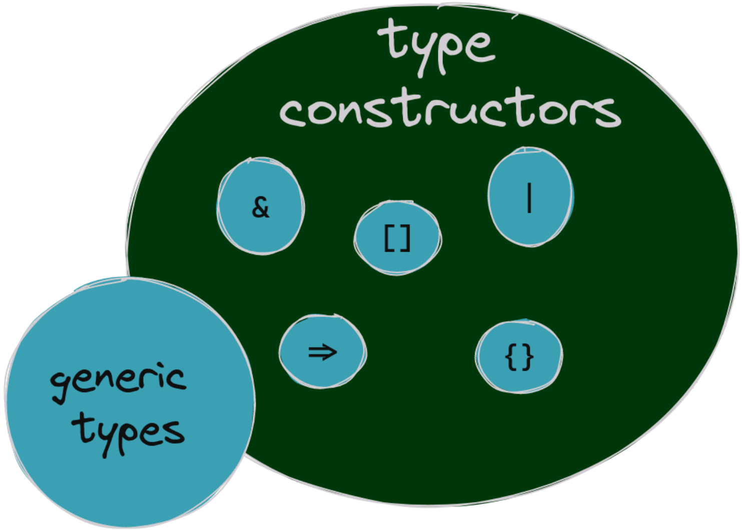
- functional - functions all we have, recursion is our friend (no higher order functions though)
- pure - no input/output
- immutable - we can not re-assign types once they already declared
- lazy - types are evaluated as needed. Opposed to JS eager evaluation.

Values


values
true
"hello world"
42
null
{x:1; y:2}
types
boolean
string
number
42
true
{
x:number,
y:number
}

Operations



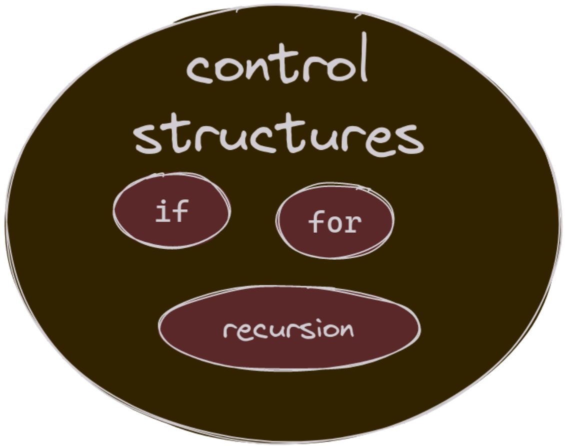


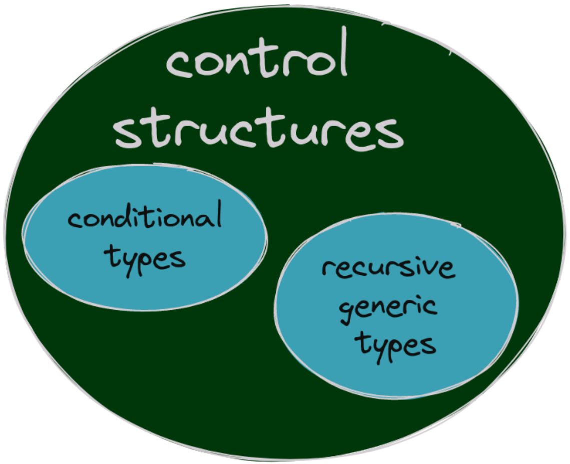
On value-level there are values and operations which operate on those values - take values as parameters and return values.

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

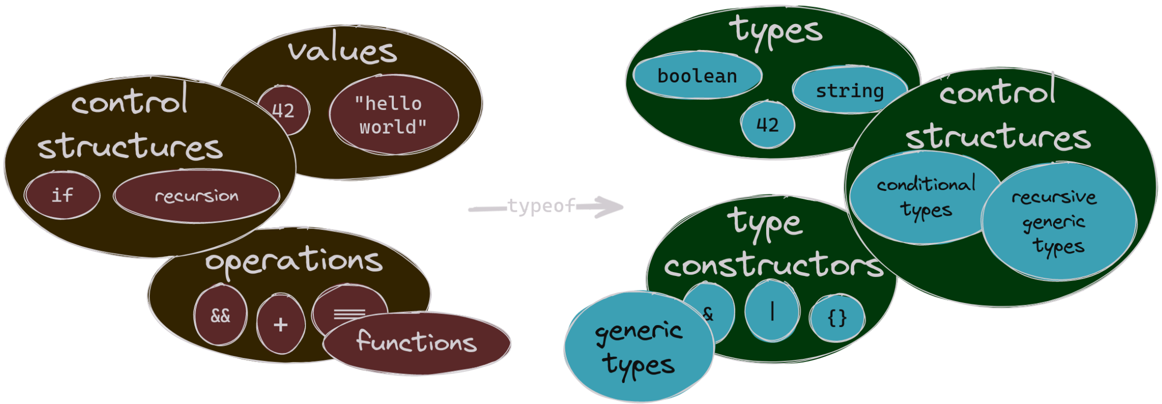
Operations on types are different from the ones we have in JS. We only have type operations which can make sense on types.

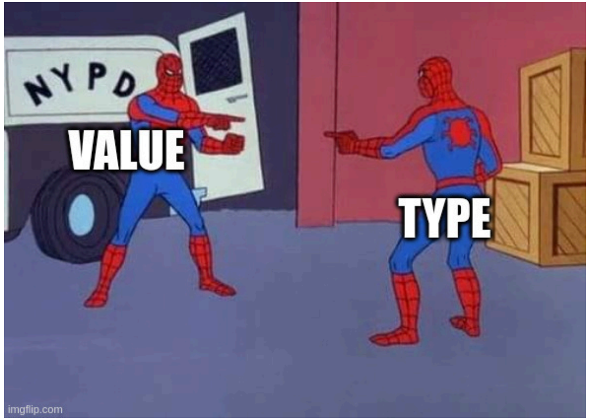
Control structures





Value vs type





Primitive

- 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";
```

Type constructors

Makes types out of other types. If in type world types are values, then type constuctions are operations on types.

Value level:

- values: `42`, `true`, `hello world`
- operations: `+`, `&&`, `===` Type level:
- values: `number`, `boolean`, `string`, `true`, `42`, `hello world`
- operations: create tuple, array, record, object, union, intersection

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

Access: `???` What does that even mean? Iterate over elements ? Think as a set of types

Intersection

Constructor:

```
type T = ... & ...
```

Access: `???` What does it even mean? Take original parts ?

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: Concat:

```
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 Access: `type E = T[0]` same

Objects

Constructor: `type T = { key1: string; key2: boolean }` Access: `type E = T['key1' | 'key1'] // string | boolean` // get property types Access: `type E = keyof T // 'key1' | 'key2'` get property keys type Access: `type E = T[keyof T]` shortcut to get all possible property keys type

Records

Constructor: `type T = { [key: string]: boolean }` , `Record<Keys, Type>` , `Record<K, V> = { [Key in K]: V }` Access: `type E = T[string]` // get property value type. Not possible Object constructor Access: `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. Togeter 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 amaized: https://github.com/codemix/ts-sql

Functions

Generic types are equivalent of function calls.

TLTS functions are generic types. They take types as inputs and values as outputs.

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

```
type Fn<A, B> = A | B;

// functions might have defaults
type Fn<A, B = null> = [A, B];
```

Type constraints. It’s possible to specify a type of an input. A type of a type?!

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

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

Conditionals (if-then expressions)

Conditional type: `type X = A extends B ? T : F` example:`type T = A extends 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.

Equivalent JS conditional expression: `let x = A ⩽ B ? true : false`

The `extends` keyword defins A and B subtype relationship - A is a subtype of B. We can imagine it as `type T = A is a subtype of B ? true : false` or `type T = A is a subset of B ? true : false` or `type T = A is assignable to B ? true : false` or anyother synomym to it A is a subtype of B (B is a supertype of A) <- 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 can be used in place of B <- a formal meaning of a formal definition A <: B, A ⊑ B, A ≤: B, A ⊆ B, A -> B

A caveat: In `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.

Pattern matching

I told you, TLTS is a functional programming language.

TLTS 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.

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" }>; // E = 'ok'
```

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

Functions are type constructors also, so pattern match works

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

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
```

Lazy evaluated

```
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

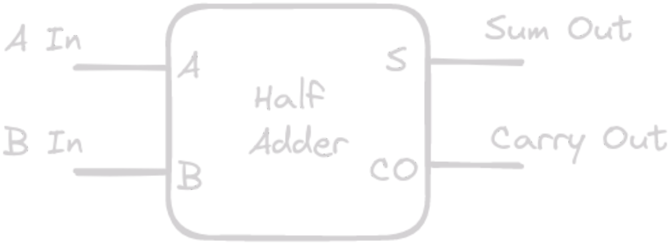
Binary arithmetics with logic gates

```
(A)    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

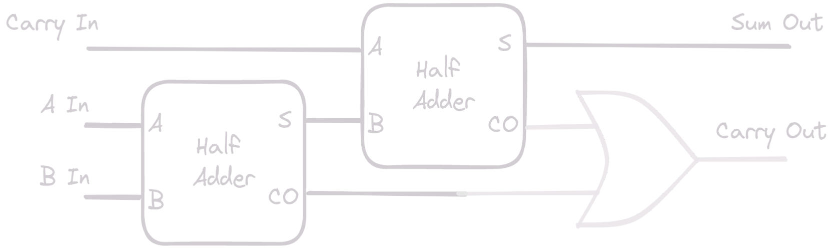
A In	B In	Sum	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

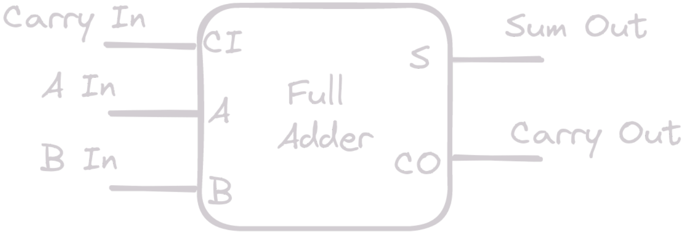



```
// Gates
type Bit = 0 | 1;
type Not<A> = A extends 0 ? 1 : 0;
type Or<A, B> = [A, B] extends [0, 0] ? 0 : 1;
type And<A, B> = [A, B] extends [1, 1] ? 1 : 0;
type Xor<A, B> = [A, B] extends [1, 0] | [0, 1] ? 1 : 0;
type NAnd<A, B> = Not<And<A, B>>;
```

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

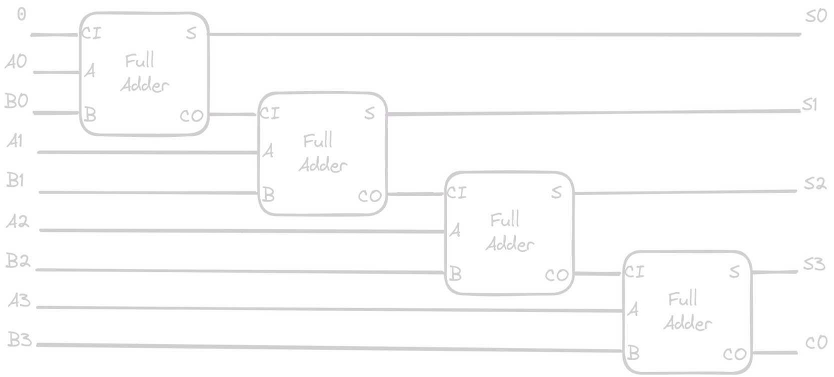
Full adder

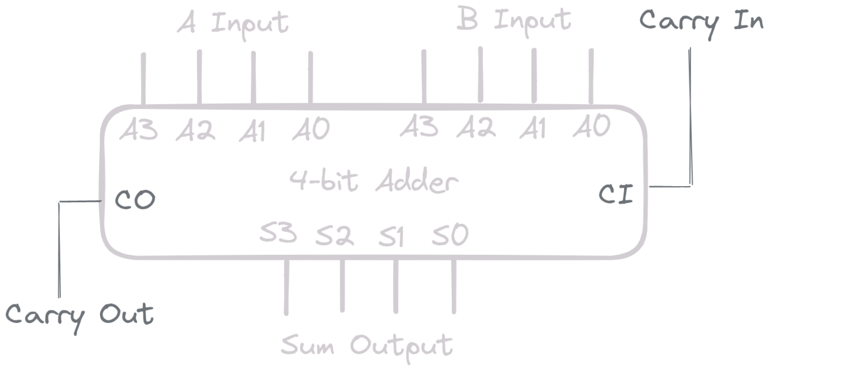





```
type FullAdder<A, B, CI> =
  HalfAdder<A, B> extends [infer HA1S, infer HA1CO]
    ? HalfAdder<CI, HA1S> extends [infer HA2S, infer HA2CO]
      ? [S: HA2S, CO: Or<HA1CO, HA2CO>]
      : never
    : never;
```

4-bit adder






```
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 extend
          [infer Y extends BinaryNumber, ...infer Ys extend
        ]
      ? 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
```