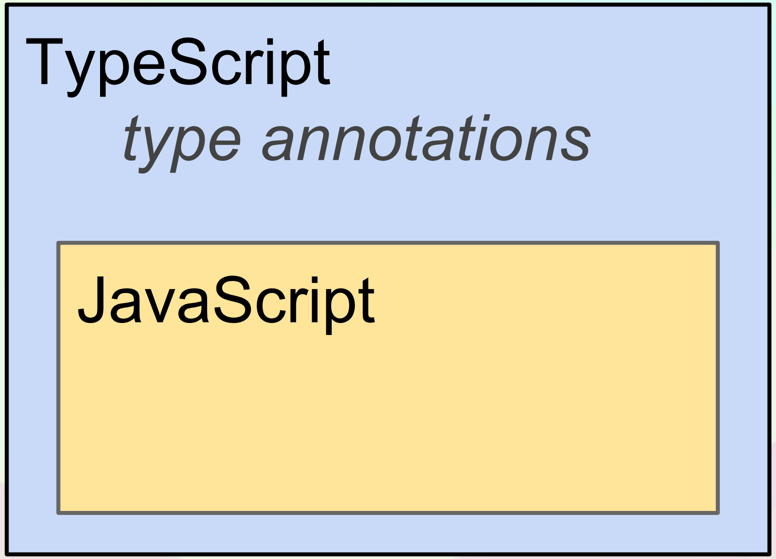
TypeScript & Dependent Typing

Ruben Pieters

TypeScript

TypeScript



TypeScript

Goals:

Static error detection as minimal layer on top of JavaScript

(Disclaimer: my personal summary from
https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals)

TypeScript

Goals:

Static error detection as minimal layer on top of JavaScript

Non-Goals:

Do not alter the way JavaScript code is organized due to addition of type system

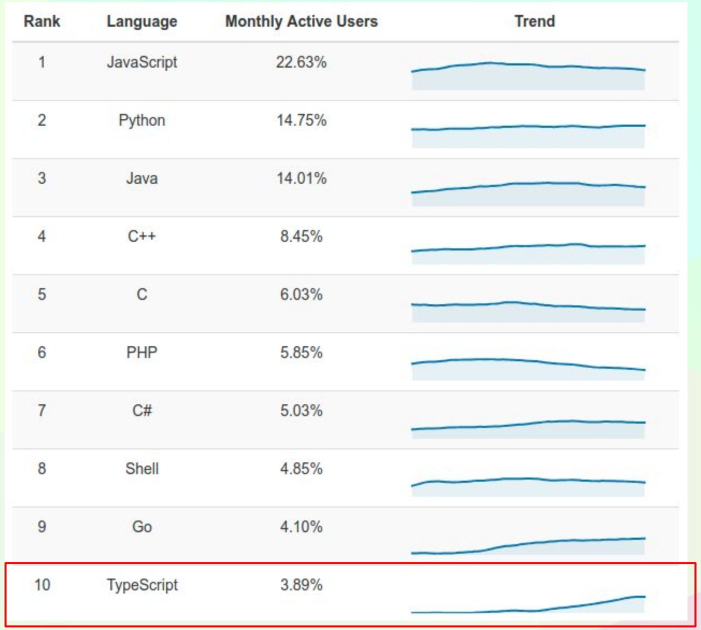
(Disclaimer: my personal summary from
https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals)

TypeScript

Type-level features:

- Literal Types
- Union/Intersection Types
- Indexed Access Types (2.1, December 2016)
- Conditional Types (2.8, March 2018)
- ...

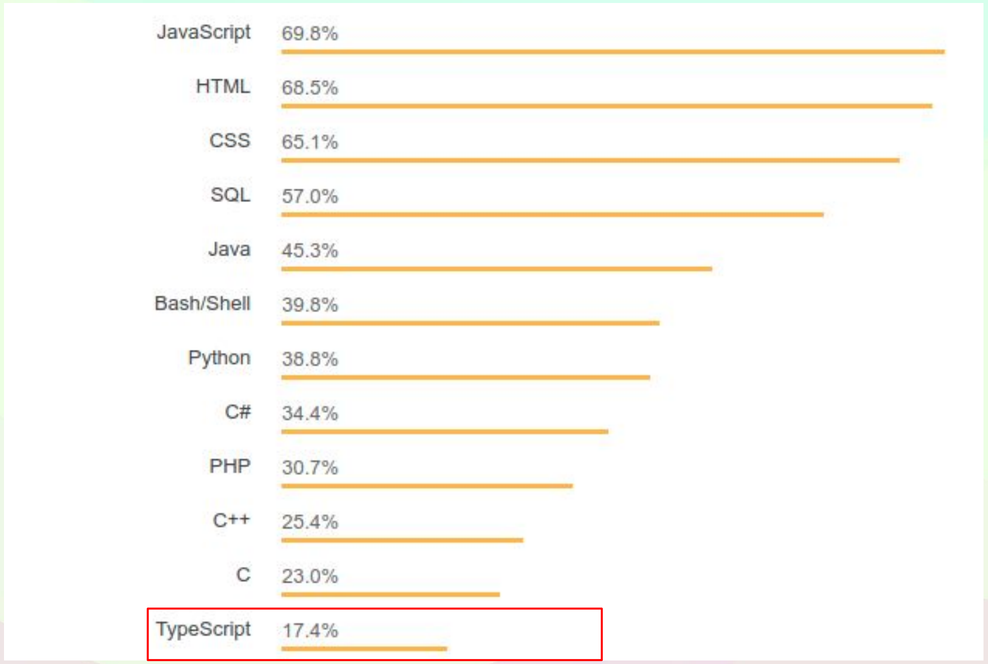
TypeScript



Rank	Language	Monthly Active Users	Trend
1	JavaScript	22.63%	
2	Python	14.75%	
3	Java	14.01%	
4	C++	8.45%	
5	C	6.03%	
6	PHP	5.85%	
7	C#	5.03%	
8	Shell	4.85%	
9	Go	4.10%	
10	TypeScript	3.89%	

http://www.benfrederickson.com/ranking-programming-languages-by-github-users

TypeScript



https://insights.stackoverflow.com/survey/2018

TypeScript

Type-level features:

- Literal Types
- Union/Intersection Types
- Indexed Access Types (2.1, December 2016)
- Conditional Types (2.8, March 2018)
- ...

Illustrate some shortcomings

Improve consistently with TS (non-)goals?

Simple Function

Simple Function

```
function add(
  x,
  y,
) {
  return x + y
}
```

Simple Function

```
add("1", 2)
```

Simple Function

```
function add(
  x: number,
  y: number,
): number {
  return x + y
}
```

Simple Function

```
add("1", 2)

"1" is not a ‘number’
```

Generics

```
function id<A>(
  a: A,
): A {
  return a
}
```

Generics

```
function id<A>(
  a: A,
): A {
  return a
}

id(1)
// id<number>(a: number): number
```

More Complex

More Complex

```
doAction("parseNumber", string): number
```

```
doAction("toString", number): string
```

More Complex

```
doAction("parseNumber", "2")

// 2

doAction("toString", 2)

// "2"
```

More Complex

```
doAction("parseNumber", "2")

// 2

doAction("toString", 2)

// "2"
```

```
el.addEventListener("click", yourFunction)
```

```
function doAction(
  action,
  input,
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction(
  action: "parseNumber" | "toString",
  input,
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction(
  action: "parseNumber" | "toString",
  input: ?,
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input,
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input,
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
        default: throw "impossible" // default case needed :(
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
) {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
type InputMap = {
    "parseNumber": string,
    "toString": number,
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

More Complex

```
doAction("parseNumber", "2") // number

doAction("parseNumber", 2)

// doAction<"parseNumber">(action: "parseNumber", input: string):
number

doAction("toString", 2) // string

doAction("toString", "2")

// doAction<"toString">(action: "toString", input: number): string
```

More Complex

```
addEventListener<K extends keyof WindowEventMap>(
  type: K,
  listener: (this: Window, ev: WindowEventMap[K]) => any,
  options?: boolean | AddEventListenerOptions
): void
```

https://github.com/Microsoft/TypeScript/blob/master/lib/lib.dom.d.ts

More Complex

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

Finished?

```
let x: "parseNumber" | "toString"

doAction(x, "2")

doAction(x, 2)

// doAction<"parseNumber" | "toString">(action: "parseNumber" |
"toString", input: string | number): string | number
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
InputMap["parseNumber" | "toString"]

= number | string
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": return Number.parseInt(input);
        case "toString": ...
    }
}
```

```
input : number | string
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": return 2; // typechecks :(
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
            // result type : string | number
        case "toString": return 2;
            // result type : string | number
    }
}
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: InputMap[K],
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

Can we fix it?

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: UnionToIntersection<InputMap[K]>,
): OutputMap[K] {
    switch (action) {
        case "parseNumber": ...
        case "toString": ...
    }
}
```

```
UnionToIntersection<string | number> = string & number
```

```
type UnionToIntersection<U> =
  (U extends any ? (k: U)=>void : never)
    extends ((k: infer I)=>void) ? I : never
```

https://stackoverflow.com/questions/50374908/transform-union-type-to-intersection-type

```
type UnionToIntersection<U> =
  (U extends any ? (k: U)=>void : never)
    extends ((k: infer I)=>void) ? I : never
```

```
UnionToIntersection<bool> = true & false
```

https://stackoverflow.com/questions/50374908/transform-union-type-to-intersection-type

```
let x: "parseNumber" | "toString";

doAction(x, "2")

// doAction<"parseNumber" | "toString">(action: "parseNumber" |
"toString", input: string & number): string | number
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: UnionToIntersection<InputMap[K]>,
): OutputMap[K] {
    switch (action) {
        case "parseNumber": return Number.parseInt(input);
        case "toString": ...
    }
}
```

```
input : UnionToIntersection<InputMap[K]>
```

```
function doAction<K extends "parseNumber" | "toString">(
  action: K,
  input: UnionToIntersection<InputMap[K]>,
): OutputMap[K] {
    switch (action) {
        case "parseNumber": return Number.parseInt(<any>input);
        case "toString": ...
    }
}
```

… (Don’t) Exactly mimic the design of existing languages ...

… (Don’t) Exactly mimic the design of existing languages ...

Dependent Types?

… (Don’t) Exactly mimic the design of existing languages ...

Dependent Types?

… Produce a language that is composable and easy to reason about ...

Type-level Computation

Type-level Computation

```
type Nat = Z | S
type Z = { tag: "Z", }
type S = { tag: "S", pred: Nat, }
```

Type-level Computation

```
type Nat = Z | S
type Z = { tag: "Z", }
type S = { tag: "S", pred: Nat, }
```

```
const one: Nat = { tag: "S", pred: { tag: "Z", }, }
```

Type-level Computation

```
type Nat = Z | S
type Z = { tag: "Z", }
type S = { tag: "S", pred: Nat, }



const one: Nat = { tag: "S", pred: { tag: "Z", }, }
type One = { tag: "S", pred: Z, }
```

Type-level Computation

```
function add(n: Nat, m: Nat): Nat {
  switch (n.tag) {
    case "Z": return m;
    case "S": return { tag: "S", pred: add(n.pred, m), };
  }
}


const two = add(one, one)
// { tag: "S", pred: { tag: "S", pred: { tag: "Z", }, }, }
```

Type-level Computation

```
type Add<N extends Nat, M extends Nat> =
  N extends Z ? M :
  { tag: "S", pred: Add<N["pred"], M> }
```

Type-level Computation

```
type Add<N extends Nat, M extends Nat> =
  N extends Z ? M :
  { tag: "S", pred: Add<N["pred"], M> }
```

```
1 extends 1 | 2 ? true : false

// true

{ tag: "S" } extends { tag: "Z" } ? true : false

// false
```

Type-level Computation

```
type Add<N extends Nat, M extends Nat> =
  N extends Z ? M :
  { tag: "S", pred: Add<N["pred"], M> }
```

Type-level Computation

```
type Add<N extends Nat, M extends Nat> =
  N extends Z ? M :
  { tag: "S", pred: Add<N["pred"], M> }



type Two = Add<One, One>
// { tag: "S", pred: { tag: "S", pred: Z } }
```

Type-level Computation

```
type VNil = {
  tag: "VNil",
}

type VCons<A, N extends Nat> = {
  tag: "VCons",
  a: A,
  tail: Vec<A, N>,
}

type Vec<A, N extends Nat> = VNil | VCons<A, N>
```

Type-level Computation

```
function vnil<A>(): Vec<A, Z> {
  return { tag: "VNil", }
}

function vcons<A, N extends Nat>(
  a: A,
  tail: Vec<A, N>,
): Vec<A, { tag: "S", pred: N }> {
  return { tag: "VCons", a, tail, }
}
```

Type-level Computation

```
function concat<A, N extends Nat, M extends Nat>(
  v1: Vec<A, N>,
  v2: Vec<A, M>,
): Vec<A, Add<N, M>> { ... }
```

Type-level Computation

```
function concat<A, N extends Nat, M extends Nat>(
  v1: Vec<A, N>,
  v2: Vec<A, M>,
): Vec<A, Add<N, M>> { ... }

const vec1 = vcons(1, vnil())
// Vec<number, One>
const vec2 = concat(vec1, vec1)
// Vec<number, Two>
```

Type-level Computation

```
function concat<A, N extends Nat, M extends Nat>(
  v1: Vec<A, N>,
  v2: Vec<A, M>,
): Vec<A, Add<N, M>> {
  switch (v1.tag) {
    case "VNil": return v2
    case "VCons": ...
  }
}
```

Type-level Computation

```
function concat<A, N extends Nat, M extends Nat>(
  v1: Vec<A, N>,
  v2: Vec<A, M>,
): Vec<A, Add<N, M>> {
  switch (v1.tag) {
    case "VNil": return v1 // typechecks :(
    case "VCons": ...
  }
}
```

Type-level Computation

```
function head<A, N extends { tag: "S", pred: Nat }>(
  v1: Vec<A, N>,
): A {
  switch (v1.tag) {
    case "VNil": throw "impossible" // still necessary :(
    case "VCons": return v1.a
  }
}
```

Type-level Computation

```
function head<A, N extends { tag: "S", pred: Nat }>(
  v1: Vec<A, N>,
): A { ... }



const x = head(vnil())
// typechecks :(
```

Type-level Computation

```
function head<A, N extends { tag: "S", pred: Nat }>(
  v1: Vec<A, N>,
): A { ... }


const x = head(vnil())
// typechecks :(
```

Where is the problem?

Type-level Computation

```
function vnil<A>(): Vec<A, Z> {
  return { tag: "VNil", }
}

type Vec<A, N extends Nat>
  = VNil | VCons<A, N>
```

```
data Vect : (len : Nat)
              -> (elem : Type)
              -> Type where
 Nil  : Vect Z elem
 (::) : (x : elem)
     -> (xs : Vect len elem)
     -> Vect (S len) elem
```

Type-level Computation

```
function add(n: Nat, m: Nat): Nat {
  switch (n.tag) {
    case "Z": return m;
    case "S": return { tag: "S", pred: add(n.pred, m), };
  }
}
```

```
type Add<N extends Nat, M extends Nat> =
  N extends Z ? M :
  { tag: "S", pred: Add<N["pred"], M> }
```