Phoomparin Mano - TypeScript Berlin Meetup #4

You might not need **advanced types.**

brikl

brikl

Phoomparin Mano (Poom)

Developer Advocate, BRIKL.

Bangkok, Thailand.

Organizes Young Creator's Camp,
The Stupid Hackathon Thailand,
BKK.JS, Hacktoberfest, CodePlearn, etc.

GitHub: @phoomparin

Get Started
Handbook
Handbook Reference
Advanced Types
Utility Types
Decorators
Declaration Merging
Iterators and Generators
JSX
Mixins
Modules
Module Resolution
Namespaces
Namespaces and Modules
Symbols
Triple-Slash Directives
Type Compatibility
Type Inference
Variable Declaration

Conditional Types

A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;

// Type is 'string | number'
let x = f(Math.random() < 0.5);
//  ^ = let x: string | number
```

Try

Another example would be the `TypeName` type alias, which uses nested conditional types:

```
type TypeName<T> = T extends string
  ? "string"
  : T extends number
  ? "number"
  : T extends boolean
  ? "boolean"
```

On this page

Type Guards and Differentiati...
User-Defined Type Guards
Using the in operator
typeof type guards
instanceof type guards
Nullable types
Optional parameters and prop...
Type guards and type assertio...
Type Aliases
Interfaces vs. Type Aliases
Enum Member Types
Polymorphic this types
Index types
Index types and index signatu...
Mapped types
Inference from mapped types
Conditional Types
Distributive conditional types
Type inference in conditional t...
Predefined conditional types

Is this page helpful?

Yes No

When do I need to use **advanced types?** 🤔

Let's **build** a

Schema Builder 💡

Let's make an **app** for

Hackathon Grouping 🎟️


SOMEWHERE SOMEONE
Sellsuki
SCB
cus
Tho Works
APPMAN

Let's look at the

API Surface 🏝️

```
const Member = t.type('member', {
  id: t.id(),
  name: t.text(),
  age: t.number(),
})
```

```
const Team = t.type('team', {
  name: t.text(),
  lead: t.of(Member),
  members: t.many(t.of(Member))
})
```

```
const poom: Member = {
  id: "AAAA-BBBB",
  name: "Poom",
  age: 19
}
```

```
const team: Team = {
  name: '🐰',
  lead: poom,
  members: [poom]
}
```

```
const h = createHandler<Team>(() => ({
  name,
  lead,
  members: [member]
}))
```

How would we write this

Just for the types?

```
const Member = t.type('member', {
  id: t.id(),
  name: t.text(),
  age: t.number(),
})
```

```
const t = {
  text: () => '',
  number: () => 0,
  many: <T>(input: T) => [input],
  of: <T>(type: T) => type,
  optional: <T>(type: T): T | null => type,
}
```

```
  many: <T>(input: T) =>
  of:                                                  p
  opti                                                 :
}

const Person = {
  name: t.text(),
  age: t.number(),
}
```

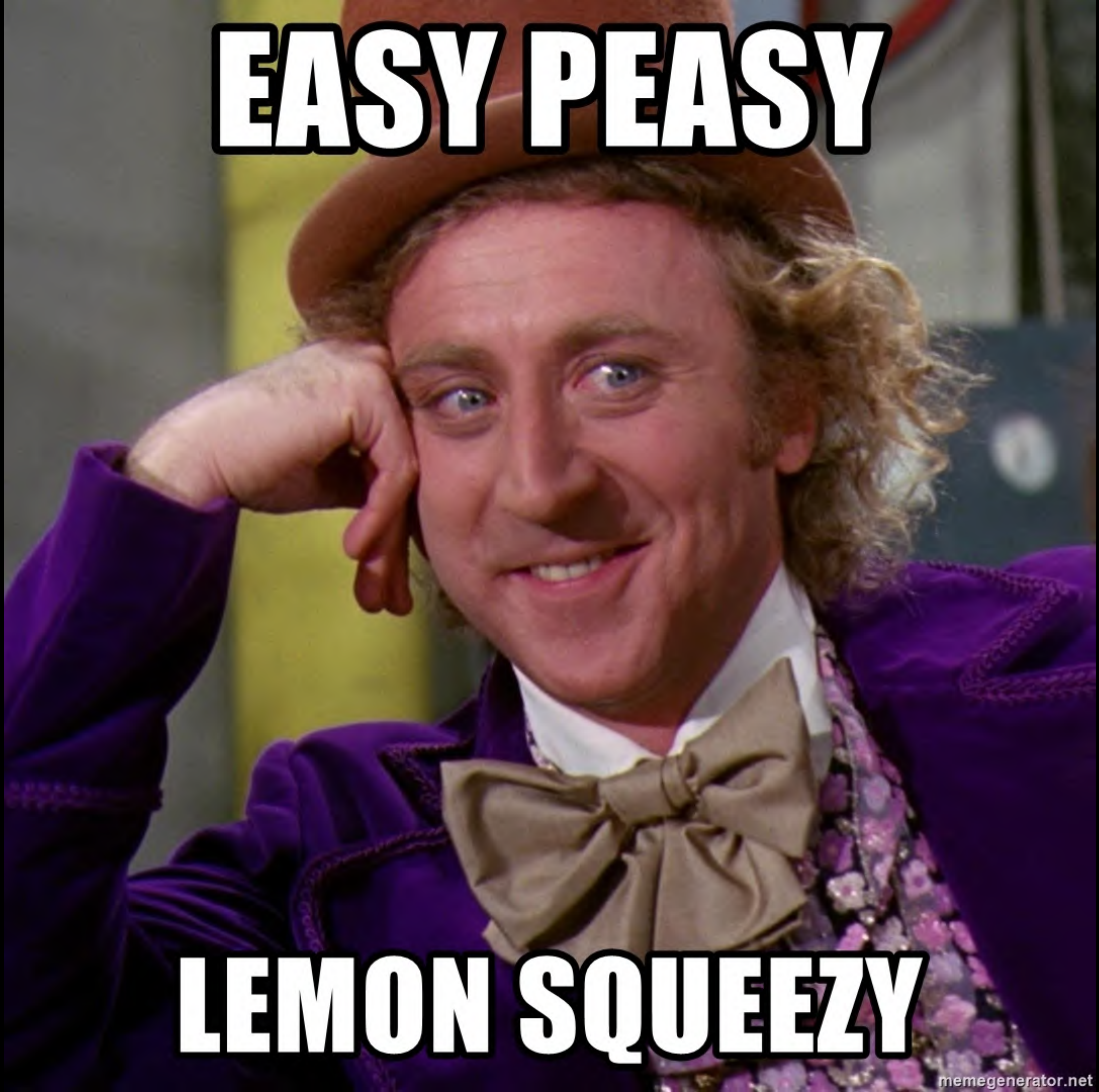
```
const Person: {
    name: string;
    age: number;
}
```

```
const
  name
  age:
}
```

```
const Team: {
    name: string;
    people: {
        name: string;
        age: number;
    }[];
```

```
const Team = {
  name: t.text(),
  people: t.many(t.of(Person)),
}
```

```
const createHandler = <T>(result: () => T) => result

const h = createHandler<typeof Project>(() => ({
  name: 'Candy Machine',
  team: {
    name: 'Lollipop Gang',
    people: [{name: 'Poom', age: 19}],
  },
  tagline: 'Hello',
}))
```


EASY PEASY
LEMON SQUEEZY
memegenerator.net

Are we done here?


Well yes, but actually no

Requirements ✅

1. Type **schema** as **JSON structure**
2. TypeScript Types

Let's use **plain JS objects** and **pure functions!**

```
const t = {
  type: (name, schema) => ({type: 'schema', name, schema}),
  text: () => ({type: 'string'}),
  number: () => ({type: 'number'}),
  many: (input) => ({type: 'array', item: input}),
  of: (type) => ({
    type: 'ref',
    item: type,
  }),
  optional: (type) => ({
    type: 'optional',
    item: type,
  }),
}
```

Compose it together! 🍻

```
const Project = t.type('project', {
  name: t.text(),
  team: t.of(Team),
  status: t.of(ProjectStatus),
  tagline: t.optional(t.text()),
})
```

```
{
  "name": {"type": "string"},
  "team": {
    "type": "ref",
    "item": {
      "type": "type",
      "name": "team",
      "schema": {
        "name": {"type": "string"},
        "lead": {⋯
        },
        "people": {
          "type": "array",
          "item": {
            "type": "ref",
            "item": {
              "type": "type",
              "name": "person",
              "schema": {
                "id": {"type": "id"},
                "name": {"type": "string"},
                "age": {"type": "number"}
              }
```

It generates

a **JSON structure**

(first requirement)

That looks good! **BUT...**

Requirements

1. Type schema as JSON structure
2. **TypeScript Types**

```
: (parameter) input: any }
(input) => ({type: 'array
```

What we need to do ✅

Generate a structure in the type land.

```
const text = () => ({type: 'string'})
```

Type is widened to string 😞

```
const text = () => ({type: 'string'})
```

```
const text = () => ({type: 'string'})
```

```
const text: () => {
    type: string;
}
```

```
const text = (): {type: 'string'} =>
  ({type: 'string'})
```

Type is precisely 'string' 🙌

```
const text = (): {type: 'string'} =>
  ({ty
```

```
const text: () => {
    type: 'string';
}
```

The **mission** here 🚀

Preserve the information in types.

```
const t = {
  text: (): {type: 'string'} => ({type: 'string'}),
  number: (): {type: 'number'} => ({type: 'number'}),
  many: <T>(input: T) => ({type: 'array', item: input}),

  of: <T>(type: T): {type: 'ref'; item: T} => ({
    type: 'ref',
    item: type,
  }),

  optional: <T>(type: T): {type: 'optional'; item: T} => ({
    type: 'optional',
    item: type,
  }),
}
```

Preserve the schema and its **name** in **type** level.

```
type: <T, N extends string>(
  name: N,
  schema: T
): {type: 'type'; name: N; schema: T} ⇒ ({
  type: 'type',
  name,
  schema,
}),
```

Still works! 😎

```
const Project = t.type('project', {
  name: t.text(),
  team: t.of(Team),
  status: t.of(ProjectStatus),
  tagline: t.optional(t.text()),
})
```

```
})

const
  'for
  'int
  'pro
)
```

```
const Project: {
    type: 'type';
    name: "project";
    schema: {
        name: {
            type: 'string';
        };
        team: {
```

```
const Project = t.type('project', {        You, 2 hours ago •
  name: t.text(),
  team: t.of(Team),
  status: t.of(ProjectStatus),
  tagline: t.optional(t.text()),
})
```

```
type ITeamSchema = {
    name: {
        type: 'string';
    };
    lead: {
        type: 'ref';
        item: {
```

```
type ITeamSchema = typeof Team.schema
```

```
const h = createHandler<Team>(() => ({
  name,
  lead,
  members: [member]
}))
```

Our next task: ✅

Generate the return type from our type.

Let's begin from Scalar types.

```
interface ScalarTypeMapping {        You,
  id: string
  string: string
  number: number
}


type Scalar = keyof ScalarTypeMapping
```

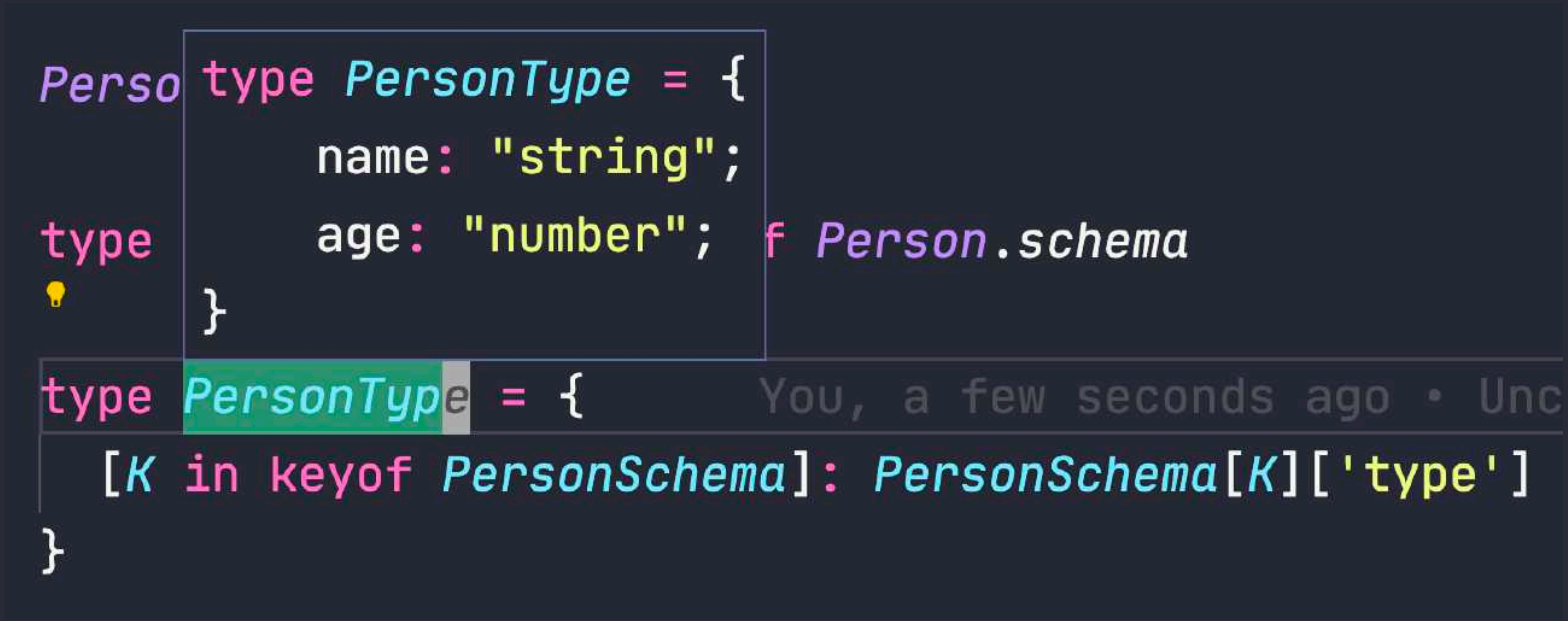
```
type Scalar = "string" | "number" |
"id"
{type: Scalar}
```

```
type PersonSchema = {
    name: {
        type: 'string';
    };
    age: {
        type: 'number';
    };
```

◁ We want the **type** field

```
type PersonSchema = typeof Person.schema

type PersonType = {
  [K in keyof PersonSchema]: PersonSchema[K]['type']
}
```

Similar to mapValues(p ⇒ p.type)



```typescript
type MapSchemaToReturnType<T> = {
  [K in keyof T]: ScalarTypeMapping[T[K]['type']]
}
```

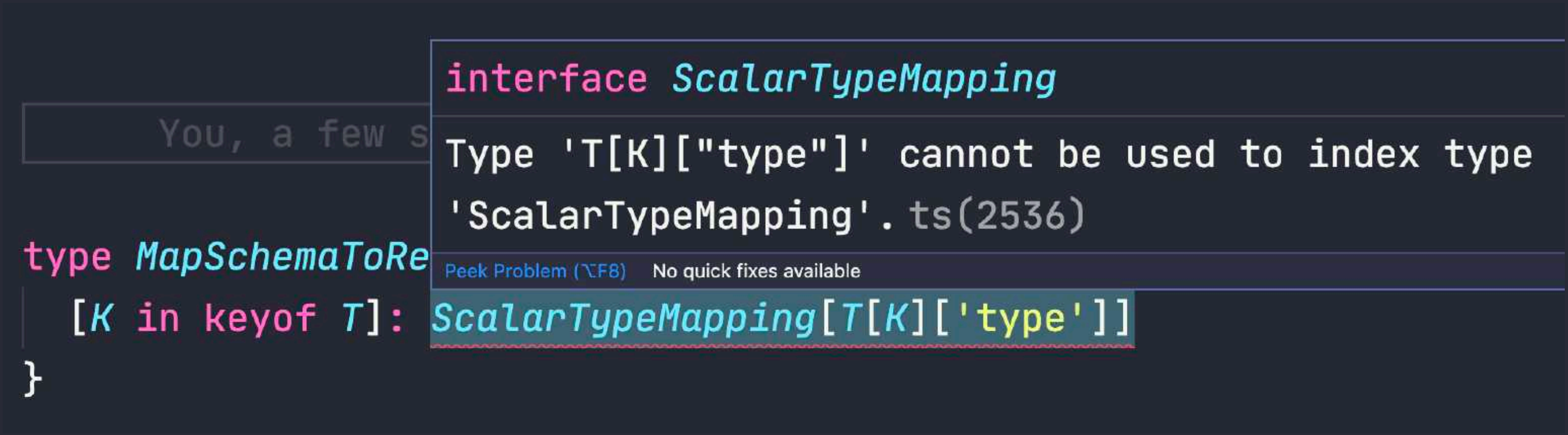

```
You, a few s

type MapSchemaToRe
  [K in keyof T]: ScalarTypeMapping[T[K]['type']]
}
```

interface ScalarTypeMapping

Type 'T[K]["type"]' cannot be used to index type 'ScalarTypeMapping'. ts(2536)

Peek Problem (⌥F8) No quick fixes available

Why does this happen?



The compiler **cannot ensure** that the **"type"** key is a **Scalar** type!

(i.e. it looks like `{type: ...}`)

```
You, a few s
interface ScalarTypeMapping
Type 'T[K]["type"]' cannot be used to index type
'ScalarTypeMapping'. ts(2536)
Peek Problem (⌥F8)   No quick fixes available

type MapSchemaToRe
  [K in keyof T]: ScalarTypeMapping[T[K]['type']]
}
```

Enter conditional types! 🤪

```
type IsScalar<T> = T extends {type: Scalar}
  ? 'yes'
  : 'no'

type A = IsScalar<{type: 'string'}>
type B = IsScalar<{type: 'cthulhu'}>
```

```
type Scalar = "string" | "number" |
```

```
type A = "yes"
```

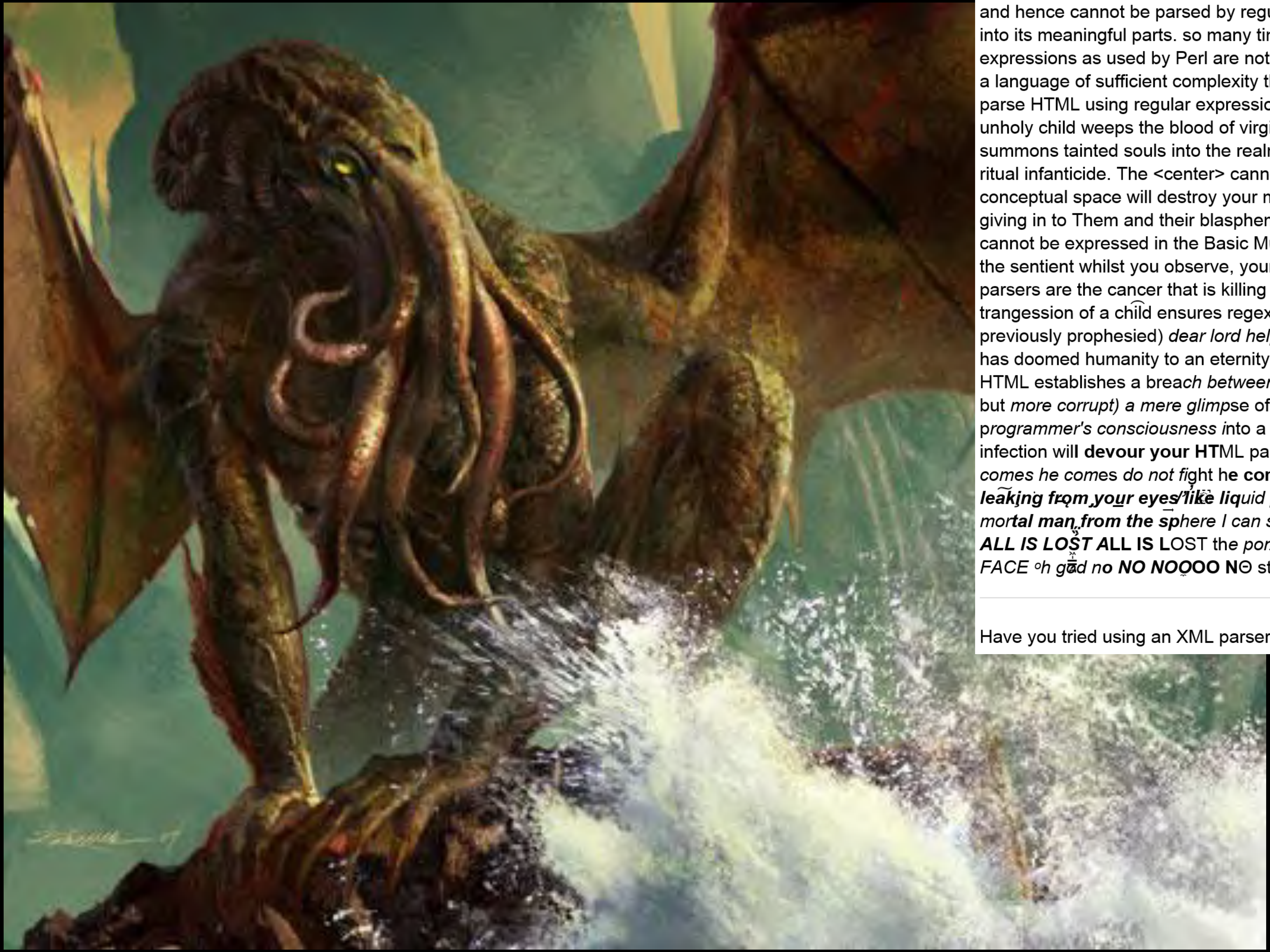
✅

```
type A = IsScalar<{type: 'string'}>
```

```
type  type B = "no"
```

✅

```
type B = IsScalar<{type: 'cthulhu'}>
```



You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a brea*ch between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a world of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiance de*stroying all enlightenment, HTML tags* ***leaking from your eyes like liq***uid pain, the song of regular expre~~ssion parsing~~ will exti*nguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie***s of Man ALL IS LOST ALL IS L**OST the pon*y he comes* he c~~omes he comes~~ ***the* ich**or permeates al*l MY FACE MY FACE* ᵒh god n*o* ***NO NOOOO N*Θ** stop t*he an*gles `are` **not real ZALGO IS TONY THE PONY HE COMES**

Have you tried using an XML parser instead?

```
interface ScalarTypeMapping {      You,
  id: string
  string: string
  number: number
}

type Scalar = keyof ScalarTypeMapping
```

```
type GetReturnType<T> = T extends {type: Scalar}
  ? ScalarTypeMapping[T['type']]
  : T
```

```
type C = number
```

✅

```
type C = GetReturnType<{type: 'number'}>
```

Okay. How about references?

```
const Team = create('team', {
  name: t.text(),
  lead: t.of(Person),
})
```

```
const lead: {
    type: 'ref';
    item: {
        type: string;
        name: string;
        schema: {
            id: {
```

```
const lead = t.of(Person)
```

How do we get the item generic inside the ref?

```
of: <T>(type: T): {type: 'ref'; item: T} => ({
  type: 'ref',
  item: type,
}),
```

```
type GetRefItem<T> = T extends {
  type: 'ref'
  item: {schema: infer Schema}
}
  ? Schema
  : 'nope'
```

```
type GetRefItem<T> = T extends {
  type: 'ref'
  item: {schema: infer Schema}
}
  ? Schema
  : 'nope'
```

The **infer keyword** infers the **type contained inside the generic.**

```
type RefTest = {
    id: 'id';
    name: 'string';
}
```

```
type RefTest = GetRefItem<{
  type: 'ref'
  item: {
    type: 'type'
    name: 'person'
    schema: {id: 'id'; name: 'string'}
  }
}>
```


Let's put them together!

Ready? Here we go.

Happy Halloween! 🎃 👻

```typescript
type MapSchemaToReturnType<T> = {
  -readonly [K in keyof T]: T[K] extends {type: Scalar}
    ? ReturnTypeMapping[T[K]['type']]
    : T[K] extends {type: 'ref'; item: {schema: infer Schema}}
    ? MapSchemaToReturnType<Schema>
    : 'none'
}
```


Let's refactor and sprinkle some comments!

Much sweeter 🍭

```
/** Map an unboxed input type (e.g. scalar, constructed type) to the return type. */
type GetReturnType<Input> =
  // Is a scalar type? (string, number)
  Input extends {type: Scalar}
    ? ReturnTypeMapping[Input['type']]
    : // Is a constructed type with schema? (created by t.type() function)
    Input extends {type: 'type'; schema: infer Schema}
    ? MapSchemaToReturnType<Schema>
    : Input
```

Let's add support for Arrays!

```
const Team = t.type('team', {
  name: t.text(),
  lead: t.of(Member),
  members: t.many(t.of(Member))
})
```

Arrays can contain **refs** and scalars, so we **apply it recursively.**

```
// Is an array type?
Input extends {
  type: 'array'
  item: infer Item
}
  ? MapBoxedInputToReturnType<Item>[]
```

```
type ArrayOfRefs = MapBoxedInputToReturnType<{
  type: 'array'
  item: {
    type: 'ref'
    item: {
      type: 'type'
      name: 'people'
      schema: {name: {type: 'string'}}
    }
  }
}>
```

```
const arrRef: ArrayOfRefs = [{name: 'Poom'}]
```

```
/** Map a boxed Input type (e.g. array, ref) to the native return type. */
type MapBoxedInputToReturnType<Input> =
  // Is an array type?
  Input extends {
    type: 'array'
    item: infer Item
  }
    ? MapBoxedInputToReturnType<Item>[]
    : // Is a reference type? (can reference constructed type or enums)
    Input extends {type: 'ref'; item: infer Item}
    ? GetReturnType<Item>
    : // Otherwise, the typed is not wrapped in array of ref.
      GetReturnType<Input>
```

The first time

you use **nested & recursively applied conditional** types,

it can be difficult to **wrap your head around** 🤯

Finally, we **apply** this to every **field** of the schema structure! 🥂

```
/** Map a schema definition `Record<string, Input>` to the na
type MapSchemaToReturnType<T> = {
  -readonly [K in keyof T]: MapBoxedInputToReturnType<T[K]>
}
```

```typescript
/** Map a boxed Input type (e.g. array, ref) to the native return type. */
type MapBoxedInputToReturnType<Input> =
  // Is an array type?
  Input extends {
    type: 'array'
    item: infer Item
  }
    ? MapBoxedInputToReturnType<Item>[]
    : // Is a reference type? (can reference constructed type or enums)
    Input extends {type: 'ref'; item: infer Item}
    ? GetReturnType<Item>
    : // Otherwise, the typed is not wrapped in array of ref.
      GetReturnType<Input>
```

```typescript
/** Map an unboxed input type (e.g. scalar, constructed type) to the return type. */
type GetReturnType<Input> =
  // Is a scalar type? (string, number)
  Input extends {type: Scalar}
    ? ReturnTypeMapping[Input['type']]
    : // Is a constructed type with schema? (created by t.type() function)
    Input extends {type: 'type'; schema: infer Schema}
    ? MapSchemaToReturnType<Schema>
    : Input
```

```
type PersonSchema = MapSchemaToReturnType<typeof Person.schema>
type TeamSchema = MapSchemaToReturnType<typeof Team.schema>
type ProjectSchema = MapSchemaToReturnType<typeof Project.schema>

const poom: PersonSchema = {        You, a few seconds ago • Uncommi
  id: 'lab-member-001',
  name: 'Poom',
  age: 19,
}

const teamRed: TeamSchema = {
  name: 'Team Red',
  lead: poom,
  people: [poom],
}

const project: ProjectSchema = {
  name: 'Team Red',
  team: teamRed,
  status: 'forming team',
}
```

That works perfectly! 🎉

Requirements

1. ~~Type schema as JSON structure ✅~~
2. ~~TypeScript Types ✅~~

We did it! 🎉 😎

<u>We now have ✅:</u>

Scalars, Types, Refs, Arrays

Let's create **Enums** next!

```
const ProjectStatus = t.enum(
    'forming team',
    'interviewing users',
    'prototyping'
)
```

How do we turn a string array into union?

```
const enums = ['A', 'B', 'C'] as const
type Enum = typeof enums[number]
```

But I don't want to use `as const` in user code! 😞

This can construct a **readonly string array!**

```
enum: <Choices extends string[]>(
  ...choices: Choices
): {type: 'enum'; choices: Choices} => ({
  type: 'enum',
  choices,
}),
```

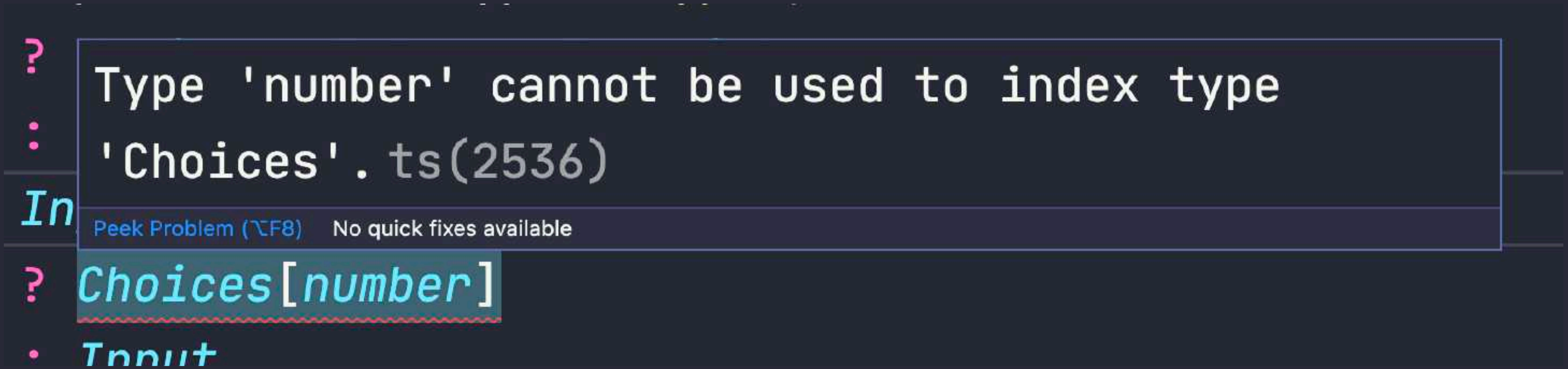
```
const ProjectStatus: {
    type: 'enum';
    choices: ["forming team", "interviewing user
s", "prototyping"];
}
```

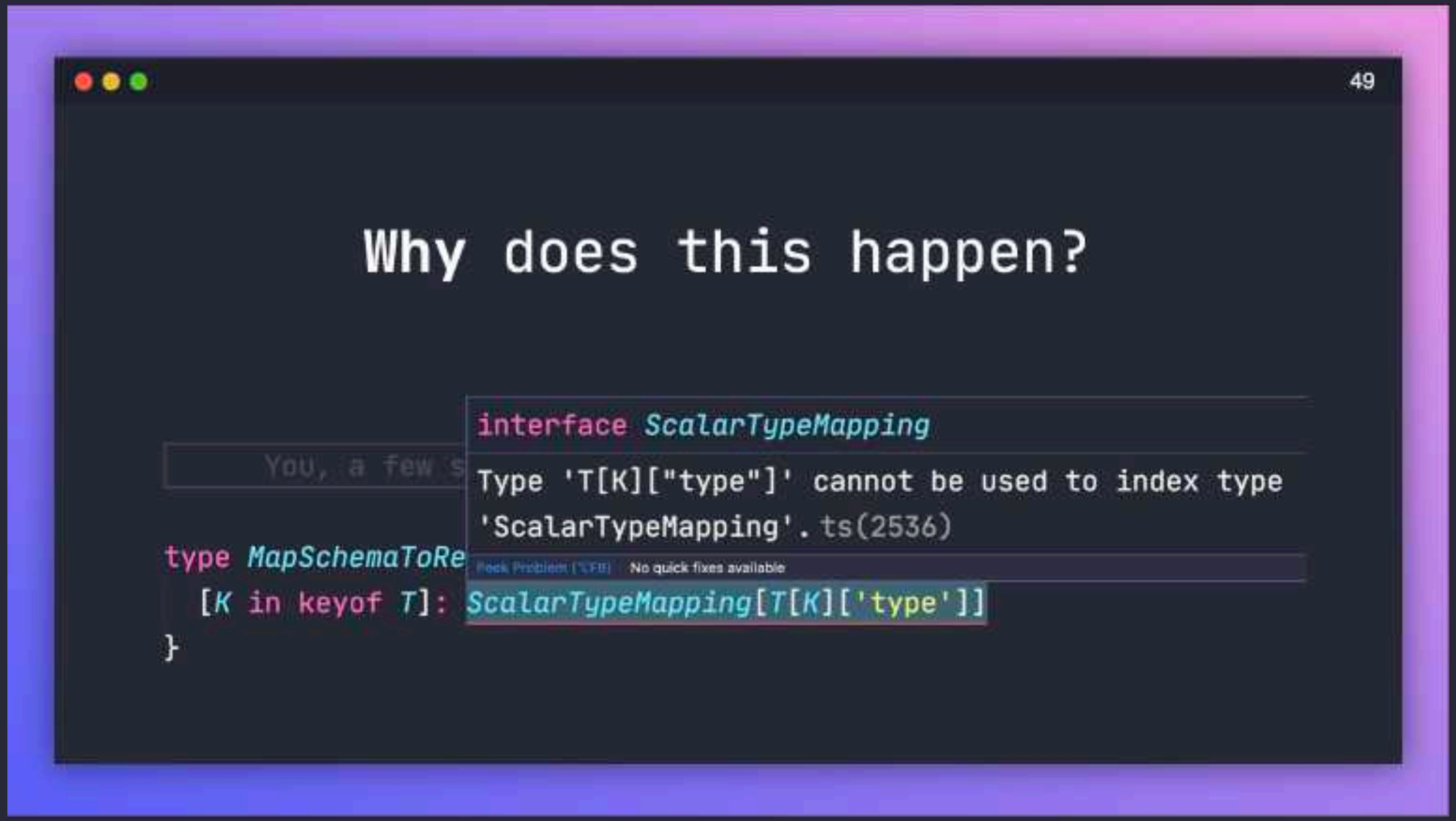
```
const ProjectStatus = t.enum(
  'forming team',
  'interviewing users',
  'prototyping'
)
```

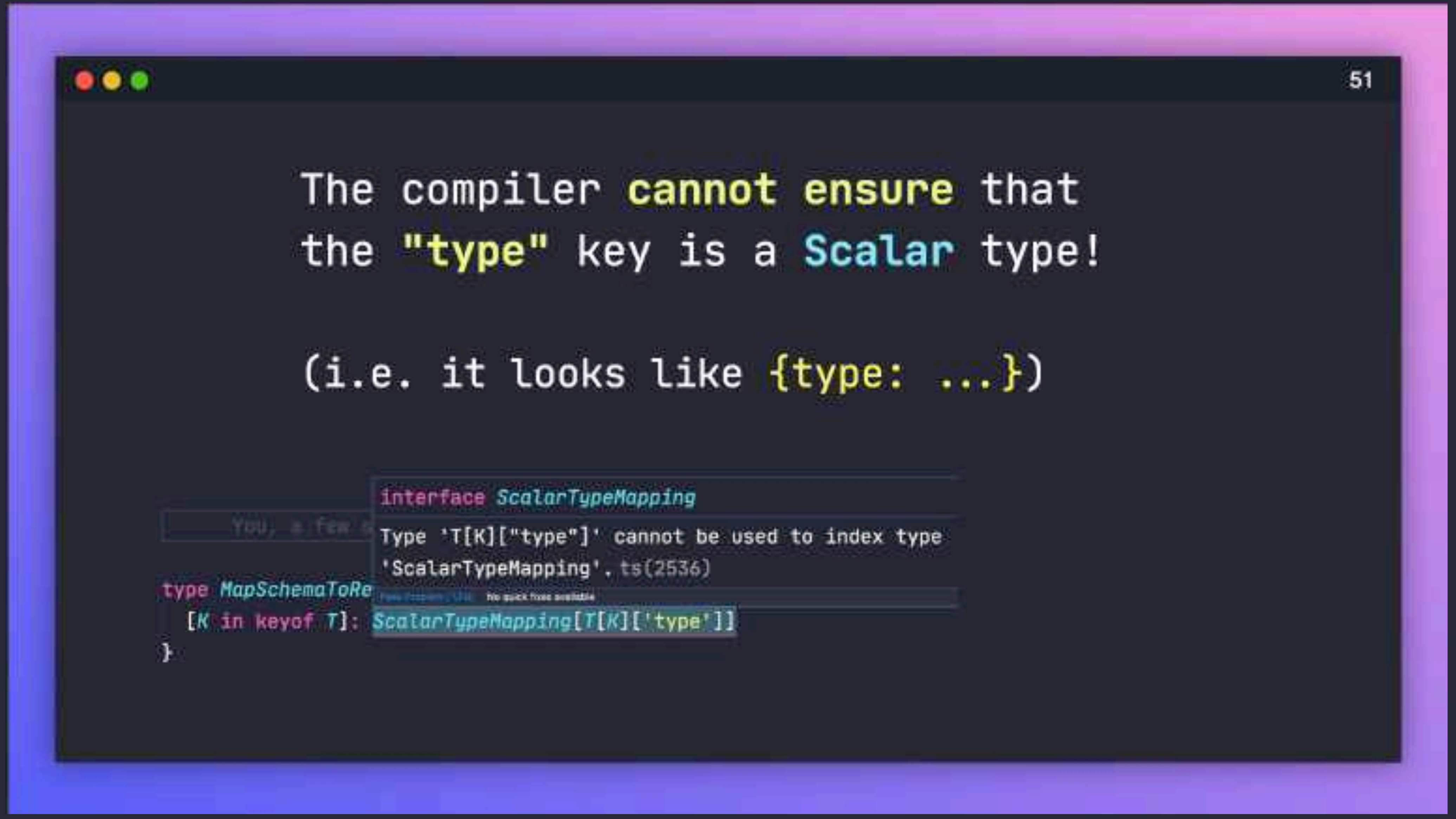
We should be able to extract the choices from the generic, then **Choices[number]**

```
type toUnion = ['hello', 'world'][number]

const a: toUnion = 'hello'
```

```
/** Map an unboxed input type (e.g. scal
type GetReturnType<Input> =
  // Is a scalar type? (string, number)
  Input extends {type: Scalar}
    ? ReturnTypeMapping[Input['type']]
    : // Is a constructed type with sche
    Input extends {type: 'type'; schema: infer Schema}
    ? MapSchemaToReturnType<Schema>
    : // Is an enum type? (created by t.enum() function)
    Input extends {type: 'enum', choices: infer Choices}
    ? Choices[number]
    : Input
```


Type 'number' cannot be used to index type 'Choices'. ts(2536)
Peek Problem (⌥F8)
No quick fixes available
Choices[number]


49
Why does this happen?
interface ScalarTypeMapping
Type 'T[K]["type"]' cannot be used to index type 'ScalarTypeMapping'. ts(2536)
type MapSchemaToRe
[K in keyof T]: ScalarTypeMapping[T[K]['type']]
}


51
The compiler cannot ensure that the "type" key is a Scalar type!
(i.e. it looks like {type: ...})
interface ScalarTypeMapping
Type 'T[K]["type"]' cannot be used to index type 'ScalarTypeMapping'. ts(2536)
type MapSchemaToRe
[K in keyof T]: ScalarTypeMapping[T[K]['type']]
}

Let's hint the compiler a bit 🤪

```
type EnumType<T extends string[] = []> = {type: 'enum'; choices: T}
```

```
/** Map an unboxed input type (e.g. scalar, constructed type) to the
type GetReturnType<Input> =
  // Is a scalar type? (string, number)
  Input extends {type: Scalar}
    ? ReturnTypeMapping[Input['type']]
    : // Is a constructed type with schema? (created by t.type() func
    Input extends {type: 'type'; schema: infer Schema}
    ? MapSchemaToReturnType<Schema>
    : // Is an enum type? (created by t.enum() function)
    Input extends EnumType<infer Choices>
    ? Choices[number]
    : Input
```

Now we can use **enums** as a **union type.**

```
type TypeEnum = "hello" | "world"
```

```
type TypeEnum = GetReturnType<{
  type: 'enum'
  choices: ['hello', 'world']
}>

const typeEnumTest: TypeEnum = 'hello'
```

We get an **autocompletion** for all possible **enum values!**

```
const project: ProjectSchema = {
  name: 'Team Red',
  team: teamRed,
  status: '',          You, a few seconds
}                   forming te…    forming …
                    interviewing users
                    prototyping
```

It's time for the **final boss!**

Now, time for the **final boss.**

Optionals.

How **hard** can it be?

```
const Team = t.type('team', {
  name: t.text(),
  lead: t.of(Member),
  tagline: t.optional(t.text())
})
```

```
optional: <T>(type: T): {type: 'optional'; item: T} ⇒ ({
  type: 'optional',
  item: type,
}),
```

What could **possibly** go wrong?

```
type ToOptionalField<T> = T | null | undefined

type Input = {name: string, age: number}

type B = {
  [K in keyof Input]: ToOptionalField<Input[K]>
}
```

as it turns out... **everything.**

```
const
type
type
type B = {
  [K in keyof Input]: ToOptionalField<Input[K]>
}
```

```
type B = {
    name: string;
    age: number;
}
'B' is declared but never used. ts(6196)
Quick Fix... (⌘.)
```

It's not very effective...

It hurt itself in its confusion!

Let's take a **step** back.

```
type PersonType = {
  name: {type: 'string'}
  tagline: {type: 'optional'; item: {type: 'string'}}
  age: {type: 'optional'; item: {type: 'number'}}
}
```

```typescript
export type Optional<T> = {
  value: T
  readonly __tag: unique symbol
}

export function toType<
  T extends { readonly __tag: symbol; value: any } = {
    readonly __tag: unique symbol
    value: never
  }
>(value: T['value']): T {
  return (value as any) as T
}
```

Bonus:

Newtype Pattern

You can wrap **anything** into a **new type!**

```
const hello: Optional<string>
const hello = toType<Optional<string>>('hello')
```

but that doesn't help in this case,
so i keep searching...

Given a **list of keys**, it makes them **optional!**

```
type NullablePartial<
  T,
  NK extends keyof T = {
    [K in keyof T]: null extends T[K] ? K : never
  }[keyof T],
  NP = Partial<Pick<T, NK>> & Pick<T, Exclude<keyof T, NK>>
> = { [K in keyof NP]: NP[K] }
```


Let's get the **game plan** ready!

Game Plan 🏀

1. Create `{type: 'optional'}`
2. Unbox the optional, default to `never`.
3. Get list of optional keys
4. Use `NullablePartial` to make those optional
5. Join optional and non-optional back.

Step 1 -- create the **optional** type 🏀

```
type PersonType = {
  name: {type: 'string'}
  tagline: {type: 'optional'; item: {type: 'string'}}
  age: {type: 'optional'; item: {type: 'number'}}
}
```

Step 2 -- unbox the **optional** type 🏀

```
/** Extract the value of an optional field,
 *  otherwise return never if not found. */
type ExtractOptionalFields<T> = {
  [K in keyof T]: T[K] extends {type: 'optional'; item: infer Item}
    ? Item
    : never
}
```

tagline? is unboxed, name becomes never

```
type OptionalFields = {
    name: never;
    tagline: {
        type: 'string';
    };
    age: {
type OptionalFields = ExtractOptionalFields<PersonType>
```

Step 3 -- get list of optional keys 🏀

```
/** Create a union out of field names
 *  in which the field type is not never. */
export type FilterKeys<T> = {
  [K in keyof T]: T[K] extends never ? never : K
}[keyof T]
```

tagline? & age? is optional

→ become a union.

```
type OptionalKeys = "tagline" | "age"
type OptionalKeys = FilterKeys<OptionalFields>
```

Step 4: Make em optional!

Given a **list of keys**, it makes them **optional.**

```
type NullablePartial<
  T,
  NK extends keyof T = {
    [K in keyof T]: null extends T[K] ? K : never
  }[keyof T],
  NP = Partial<Pick<T, NK>> & Pick<T, Exclude<keyof T, NK>>
> = { [K in keyof NP]: NP[K] }
```

✅

```
type OptionalFields = {
    name: never;
    tagline: {
        type: 'string';
    };
    age: {
type OptionalFields = ExtractOptionalFields<PersonType>
```

✅

```
type OptionalKeys = "tagline" | "age"
type OptionalKeys = FilterKeys<OptionalFields>
```

Finally, we got them to be optional!

```
type Optionals = {
    tagline?: {
        type: 'string';
    };
    age?: {
        type: 'number';
    };
```

Uncommitted changes

```
type Optionals = NullablePartial<OptionalFields, OptionalKeys>
```

Step 5 - Join optional and non-optional back.

We did it! 🎉

```
type Result = Pick<Optionals, OptionalKeys> &
  Pick<PersonType, Exclude<keyof PersonType, OptionalKeys>>

const typeOutput: MapSchemaToReturnType<Result> = {
  name: 'Hello!',
  ag          You, a few seconds ago • Uncommitted changes
}   age?                (property) ag…
```

Also, Extract<> and Exclude<> is my most **favourite utility type!**

```ts
/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;

/**
 * Extract from T those types that are assignable to U
 */
type Extract<T, U> = T extends U ? T : never;
```

and we're (almost) done here! yay :)

oh, about the talk title...

You might **not** need **advanced types**

If you can **simplify** your **API** surface.

(maybe it's better if the end user wrote their own types? depends.)

That's it.

Thank you!

oh. wait.

one more thing...

brikl

we're hiring.

brikl.com/careers

That's it.

Thank you!