alexharri

Build your own schema language with TypeScript's infer keyword

The `infer` keyword in TypeScript, especially when combined with recursive types, is incredibly powerful. It enables you to transform types in complex and intricate ways that feel like they should be impossible.

In this article, we'll solve the problems I encountered in building a schema builder library that makes extensive use of `infer` and recursive types.

By reading this article, I offer you an understanding of the `infer` keyword which will make you better suited to tackling new and harder TypeScript problems.

Introduction

A few years ago when I was learning Go, I learned about struct tags.

```
type Payload struct {
    Email string `json:"email,omitempty"`
}
```

Struct tags allow you to attach "meta information" to fields. They can then be read by other parts of your program to modify its behavior.

In other languages that do not have this feature, metadata can be attached to fields by wrapping the field in some type.

```
interface FieldWithMetadata<T> {
  field: T;
```

```
    alexharri
}
```

Schema builders need to solve this problem. To create useful schemas you will want to add constraints to fields, such as requiring that an input string is a valid email address. Using Yup, a JavaScript schema builder, you might write code like this:

```
import { object, string } from "yup";

const schema = object({
  email: string().email(),
});
```

This syntax is nice. However, there are some aspects of the syntax that are more verbose than I would like.

Let's take some examples. To mark a field as required, we add `.required()`.

```
const schema = object({
  email: string().email().required(),
});
```

To add a default value, we add `.default()`.

```
const schema = object({
  email: string().email().default("user@example.com"),
});
```

And for list of values, we wrap the type in `array()`.

```
  emails: array(string().email()).required(),
});
```

This is all fine, but I would love to be able to use a more terse, "native-feeling", syntax to describe the schema.

Here's what I would like. To start, I want to denote the optionality of a field using the same syntax as TypeScript:

```
const schema = object({
  email?: string().email(),
});
```

Then, we would use the same syntax for default values as in JavaScript destructuring assignments:

```
const schema = object({
  email?: string().email() = "user@example.com",
});
```

For lists of values, I would like to use an array literal syntax like so:

```
const schema = object({
  contacts: string[],
});
```

Finally, let's invent some custom syntax around constraints such as `.email()`.

```
const schema = object({
```

```
    alexharri
});
```

I like this, but it won't compile. This is just not valid JavaScript (or TypeScript) syntax.

But why should we let JavaScript dictate what we can and cannot do? CSS-in-JS is a thing, and they use template literals to embed CSS in JavaScript.

```
const Button = styled.button`
  display: inline-block;
  border-radius: 50%;
`;
```

Well, we can do the same thing.

```
const schema = compileSchema(`{
  email?: string <email> = "user@example.com";
}`);
```

Don't we lose all type information?

It certainly feels like we should, but no!

Right now, our schema builder's interface can be described like so.

```
function compileSchema<T extends string>(template: T): Schema<???>;
```

We need to fill in the blanks. We somehow want to take the string template `T` and convert it to the type that it represents.

alexharri

there.

Parsing primitives

The `extends` keyword enables conditional types in TypeScript.

```
type Test<T> = T extends "yes" ? 0 : 1;

Test<"yes">; // 0
Test<"no">; // 1
```

We can use that to get the primitive represented by an input string:

```
type ParsePrimitive<T> = T extends "string"
  ? string
  : T extends "number"
  ? number
  : T extends "boolean"
  ? boolean
  : never;

ParsePrimitive<"boolean">; // boolean
```

The repetitive `? : ? :` is somewhat unwieldy and hard to read, but it allows us to create a chain of conditionals.

Parsing objects

The `ParsePrimitive` type is useful, but it's insufficient to tackle a more complex string such as `"{value:number}"`. We can't create an infinite number of cases to match every possible key-value combination.

To convert an object string `T` into a real type we need to

2. extract the key from `T`,
3. extract the value from `T`,
4. create an object using the key-value pair

We can test whether a string `T` matches the key-value object pattern using Template Literal Types.

```
type IsObjectString<T> = T extends `{${string}:${string}}`
  ? true
  : false;
```

This alone is not very useful. If we want to transform the string `T` into the actual object type it represents, we need to extract information about the key and value from `T`.

```
type ParseObject<T> = T extends `{${string}:${string}}`
  ? something // We need to extract information from 'T'
  : never;
```

The `infer` keyword allows us to create variables during pattern matching.

```
type ParseObject<T> = T extends `{${infer K}:${infer V}}`
  ?
    // We have access to 'K' and 'V' here
    something
  :
    // 'K' and 'V' are not accessible here
    never;
```

The variables `K` and `V` are **only** created if `T` matches the pattern.

alexharri

```
type ParseObject<T> = T extends `{${infer K}:${infer V}}`
  ? { [key in K]: ParsePrimitive<V> }
  : never;

ParseObject<`{value:number}`>; // { value: number }
```

Objects with multiple properties

The first big hurdle we encounter is objects with multiple properties.

```
ParseObject<`{a:string;b:number}`>; // { a: never }
```

The naive approach would be to match create multiple conditionals for each number of properties:

```
type ParseObject<T> =
  T extends `{${infer K0}:${infer V0}}`
  ? { [key in K0]: ParsePrimitive<V0> }
  : T extends `{${infer K0}:${infer V0};${infer K1}:${infer V1}}`
  ? { [key in K0]: ParsePrimitive<V0> } & { [key in K1]: ParsePrimitive<V1> }
  : T extends `{${infer K0}:${infer V0};${infer K1}:${infer V1}};${infer K2}:${
  ? { [key in K0]: ParsePrimitive<V0> } & { [key in K1]: ParsePrimitive<V1> } &
  : never;
```

This is terrible and does not scale. We can divide and conquer instead.

Given that we have the content of an object string `T` that contains N properties, we can split the content of `T` into N many strings that contain one property. We can then parse each property string individually.

alexharri

```
  ? MergeArrayOfObjects<ParseProperties<SplitProperties<Content>>>
  : never;
```

Let's start off with `SplitProperties`.

Split properties

Given a string `T`:

```
`a:string;b:number;c:boolean`
```

We want the output of `SplitProperties<T>` to be:

```
["a:string", "b:number", "c:boolean"];
```

We can somewhat trivially create a type that splits the string by `;`.

```
type SplitProperties<T> = T extends `${infer A};${infer B}`
  ? [A, B]
  : [T];

Equals<
  SplitProperties<`a:string;b:number;c:boolean`>,
  ["a:string", "b:number;c:boolean"],
>;
```

Note: `Equals` as used above asserts that the two arguments are equal.

But this only splits at the first instance of `;`.

alexharri

that contains no `;`.

```
type SplitProperties<T> = T extends `${infer A};${infer B}`
  ? [A, ...SplitProperties<B>]
  : [T];

Equals<
  SplitProperties<`a:string;b:number;c:boolean`>,
  ["a:string", "b:number", "c:boolean"],
>;
```

This is our first example of recursive types. To better visualize what is going on, we can break down what is happening step by step:

```
// Shortened to 'Split' for brevity
type Split<T> = T extends `${infer A};${infer B}`
  ? [A, ...Split<B>]
  : [T];

Split<`1;2;3;4`>; // [`1`, ...Split<`2;3;4`>]
Split<`2;3;4`>;   // [`2`, ...Split<`3;4`>]
Split<`3;4`>;     // [`3`, ...Split<`4`>]
Split<`4`>;       // [`4`]
```

At each iteration, we find a single substring and delegate the responsibility of finding the rest of the substrings by recursing. Once we reach the base case of a string with no `;`, we return the string and stop recursing.

We're going to be using recursive types **a lot** so take a moment to deeply understand what is going on. Things will only get more complex from here.

Parsing and merging the list of properties

emitted by `SplitProperties<T>`.

```
type ParseProperties<T extends string[]> = {
  [K in keyof T]: ParseProperty<T[K]>;
};
```

We'll implement `ParseProperty` later. For now, we'll assume that `ParseProperty<T>` returns an object type that looks like so:

```
{ [key in K]: V };
```

So for our example string, we have an output of:

```
Equals<
  ParseProperties<SplitProperties<`a:string;b:number;c:boolean`>>,
  [{ a: string }, { b: number }, { c: boolean }],
>;
```

We can merge an array of objects like so:

```
type MergeArrayOfObjects<T> = T extends [infer R, ...infer Rest]
  ? R & MergeArrayOfObjects<Rest>
  : {};

Equals<
  MergeArrayOfObjects<[{ a: string }, { b: number }, { c: boolean }]>,
  { a: string } & { b: number } & { c: boolean },
>;
```

We're using recursive types again. Let's break this down a bit.

alexharri

element.

On a successful match, a variable `R` will be created that contains the first element in the array. The rest of the elements will be placed into an array `Rest`.

The `...` in `...Rest` indicates that we want the rest of the elements, from zero to infinity.

```
type Example<T> = T extends [infer R, ...infer Rest]
  ? { R: R, Rest: Rest }
  : never;

Example<[1, 2, 3]> // { R: 1, Rest: [2, 3] }
Example<[2, 3]>    // { R: 2, Rest: [3] }
Example<[3]>       // { R: 3, Rest: [] }
Example<[]>        // never -- there are no elements for 'R' to match
```

Let's break down the `MergeArrayOfObjects` in the same way:

```
// Shortened to 'Merge' for brevity
type Merge<T> = T extends [infer R, ...infer Rest]
  ? R & Merge<Rest>
  : {};

Merge<[{ a: 1 }, { b: 2 }, { c: 3 }]> // { a: 1 } & Merge<[{ b: 2 }, { c: 3 }]>
Merge<[{ b: 2 }, { c: 3 }]>           // { b: 2 } & Merge<[{ c: 3 }]>
Merge<[{ c: 3 }]>                     // { c: 3 } & Merge<[]>
Merge<[]>                             // {}
```

We keep recursing until we reach a base case. For `SplitProperties`, the base case was a string `T` without a `;`. For `MergeArrayOfObjects`, the base case is an empty array.

Object properties

A schema language that only supports primitives would produce large flat objects:

```
  bookName: string;
  bookDescription: string;
  authorName: string;
  authorAge: number;
}`);
```

I would much rather write this as two object properties:

```
const schema = schema(`{
  book: {
    name: string;
    description: string;
  };
  author: {
    name: string;
    age: number;
  };
}`);
```

In supporting object properties, we run into our first edge case.

```
type SplitProperties<T> = T extends `${infer A};${infer B}`
  ? [A, ...SplitProperties<B>]
  : [T];

Equal<
  SplitProperties<`a:{b:string;c:number};d:boolean`>,
  ["a:{b:string", "c:number}", "d:boolean"],
>;
```

The pattern matching in `${infer A};${infer B}` is greedy so it matches the first instance of `;` that it encounters. This splits object properties with multiple sub-properties.

alexharri

```
type SplitProperties<T> =
  T extends `${infer A}{${infer Content}};${infer B}`
  ? [`${A}{${Content}}`, ...SplitProperties<B>]
  : T extends `${infer A};${infer B}`
  ? [A, ...SplitProperties<B>]
  : [T];

Equals<
  SplitProperties<`a:{b:string;c:number};d:boolean`>,
  ["a:{b:string;c:number}", "d:boolean"],
>;
```

And, well, this seems to produce the correct result. However, this is easily broken by introducing one more level of nesting.

```
Equals<
  SplitProperties<`a:{b:{c:string};d:number};e:boolean`>,
  ["a:{b:{c:string}", "d:number}", "e:boolean"],
>;
```

This just moves the problem one level down.

Additionally, specifically matching `;` after `{}` is a problem when the object is the last property.

```
type SplitProperties<T> =
  T extends `${infer A}{${infer Content}};${infer B}`
  //                                      ^
  ? [`${A}{${Content}}`, ...SplitProperties<B>]
  : // ...;

type T1 = Equals<
  SplitProperties<`a:string;b:{c:number;d:boolean}`>,
```

alexharri

We need a more robust way to deal with object properties.

Balancing brackets

The solutions we've used to split the list of properties all have the same problem. They split up object properties.

If we take a look at an incorrectly split-up property, such as `a:{b:string` or `a:{b:{c:string}`, we can observe that the number of opening and closing brackets (`{` and `}`) are unequal. In a well-formed object property, the number of opening and closing brackets will always be equal.

This observation leads to a different solution. Instead of preventing object properties from being split in the first place, we can fix them after the fact by balancing brackets.

Balancing brackets can be done with a relatively simple algorithm. Starting at the first element.

1. If the number of `{` and the number of `}` in the current element are not equal, then the string is unbalanced.
2. If the string is unbalanced, merge the current element with the next element and repeat step 1 again.
3. If the string is balanced, move to the next element.

In JavaScript, a recursive version of this algorithm looks like so:

```
function areBracketsBalanced(s: string) {
  return numberOf("{").in(s) === numberOf("}").in(s);
}

function balanceBrackets(items: string[]) {
  if (items.length < 2) return items;
```

```
    return [items[0], ...balanceBrackets(items.slice(1))];
  }

  const merged = items[0] + items[1];
  return balanceBrackets([merged, ...items.slice(2)]);
}
```

Note: This recursive solution is terrible for memory usage. We're constructing a new array in every iteration. An iterative approach with a while loop would be optimal.

We can apply the same recursive pattern to balance brackets using types:

```
type BalanceBrackets<T extends string[]> =
  T extends [
    infer Curr extends string,
    infer Next extends string,
    ...infer Rest extends string[]
  ]
    ? AreBracketsBalanced<Curr> extends true
      ? // Process next item
        [Curr, ...BalanceBrackets<[Next, ...Rest]>]
      : // Merge the next item with the current item
        // and recursively process the merged item
        BalanceBrackets<[`${Curr};${Next}`, ...Rest]>
    : T;
```

This implements the same algorithm as the JavaScript example above, just for types.

The base case occurs when there are less than two elements in the array (we can't merge 0 or 1 elements), which we implement in JavaScript with:

```
if (items.length < 2) return items;
```

In TypeScript, we do that with:

alexharri

This pattern requires that both `Curr` and `Next` match a specific element in the array. If there are not at least two elements in the array `Curr` and `Next` can't be assigned, so that pattern is not matched. This implements the base case for our recursion.

Note: Remember that `...Rest` can be assigned zero to infinite elements.

However, we need to define `AreBracketsBalanced`. We want that type to return `true` if the string `T` contains the same number of `{` and `}`, and false otherwise.

Counting the number of characters in string type

To be able to check if a string contains an equal number of `{` and `}`, we first need to be able to count the number of those characters in the string.

We can access the number of elements in a tuple by reading its `length` property:

```
[string, string, string]["length"]; // 3
```

However, TypeScript just returns `number` for the length of string constants.

```
"abc"["length"]; // number
```

So what this problem boils down to is:

1. converting an input string `T` into a tuple of characters
2. filtering the tuple to only contain the character we're counting
3. reading the `length` of the tuple

We can convert a string into a tuple by recursively inferring one character at a time.

alexharri

```
  T extends `${infer Char}${infer Rest}`
    ? [Char, ...StringToTuple<Rest>]
    : [];

StringToTuple<"abc">; // ["a", "b", "c"]
```

We can filter that tuple with some more recursive inference.

```
type FilterTuple<T extends any[], Include> =
  T extends [infer Item, ...infer Rest]
    ? Item extends Include
      ? [Item, ...FilterTuple<Rest, Include>]
      : FilterTuple<Rest, Include>
    : [];

FilterTuple<[3, 2, 3, 3, 4, 5], 3>; // [3, 3, 3]
```

Combining these, we can count the instances of a character in a string.

```
type InstancesInString<T extends string, Char> =
  FilterTuple<StringToTuple<T>, Char>["length"];

InstancesInString<`a:{b:{c:string}`, "{">; // 2
InstancesInString<`a:{b:{c:string}`, "}">; // 1
```

With that, we can create a type that checks whether the brackets are balanced.

```
type AreBracketsBalanced<T extends string> =
  InstancesInString<T, "{"> extends InstancesInString<T, "}">
    ? true
    : false;

AreBracketsBalanced<`a:{b:{c:string}`>; // false
```

alexharri

Putting all of this together, we can now split properties correctly:

```
type SplitProperties<T extends string> =
  BalanceBrackets<SplitString<T, ";">>;

Equals<
  SplitProperties<`a:{b:string;c:number};d:boolean`>,
  ["a:{b:string;c:number}", "d:boolean"],
>;
```

Parsing a property

`SplitProperties` is now producing an array of strings representing properties for us to process. Let's now get to implementing `ParseProperty`, which I promised earlier.

Currently, properties take the form of

- a primitive property, such as `a:string`
- an object property containing sub-properties, such as `a:{b:string}`

The commonality between these is that both start with a key and a colon, allowing us to create a common `KeyValue` type.

```
type KeyValue<T extends string> = T extends `${infer K}:${infer V}`
  ? {
      key: K;
      value: ParseValue<V>;
    }
  : never;
```

and a primitive property.

```
type ParseValue<T> = T extends `{${string}}`
  ? ParseObject<T>
  : ParsePrimitive<T>;
```

With these, we can create a `ParseProperty` type.

```
type ParseProperty<T extends string> = KeyValue<T> extends {
  key: infer K extends string;
  value: infer V;
}
  ? { [key in K]: V }
  : never;
```

Putting this together, we have now implemented a somewhat basic parser.

```
type ParseObject<T> = T extends `{${infer Content}}`
  ? MergeArrayOfObjects<ParseProperties<SplitProperties<Content>>>
  : never;

Equals<
  ParseObject<`{a:{b:string;c:number};d:boolean}`>,
  {
    a: { b: string; c: number };
    d: boolean;
  },
>
```

Array properties

As mentioned earlier, I would like to support array properties using an array literal syntax.

```
  values: number[];
}`);
```

Arrays of objects should be supported, and arrays should be able to be multi-dimensional.

```
const schema = compileSchema(`{
  matrix: { value: number }[][];
}`);
```

To support this, we can augment `FindValue` to check for array notation.

```
type ParseValue<T> =
  // Match array notation
  T extends `${infer Before}[]`
  ? ParseValue<Before>[]
  :
  // Match object
  T extends `{${string}}`
  ? ParseObject<T>
  :
  // Default to primitives if neither array nor object
  ParsePrimitive<T>;

ParseValue<`{a:string[]}[][]`>; // { a: string[]; }[][]
```

Optional values

I would like to be able to denote optional properties using `?:` like in TypeScript:

```
const schema = compileSchema(`{
  value?: number;
}`);
```

alexharri

We can update `KeyValue` to check for the presence of `?:`.

```
type KeyValue<T extends string> =
  // Optional property
  T extends `${infer K}?:${infer V}`
  ? {
      key: K;
      value: ParseValue<V> | null;
    }
  :
  // Required property
  T extends `${infer K}:${infer V}`
  ? {
      key: K;
      value: ParseValue<V>;
    }
  : never;

ParseValue<`{a?:number}`>; // { a: number | null }
```

This looks sensible, but there's a subtle bug.

If a non-optional object property contains an optional property, then the `?:` in `${infer K}?:${infer V}` matches the property inside of the object.

```
type KeyValue<T extends string> =
  T extends `${infer K}?:${infer V}`
    ? [K, V]
    : never;

KeyValue<`a:{b?:string}`>; // ["a:{b", "string}"]
```

We can resolve this by always matching the first `:` and then checking whether `K` ends with a `?`.

alexharri

```
  T extends `${infer K}:${infer V}`
    ? K extends `${infer KeyWithoutQuestionmark}?`
      ? {
          key: KeyWithoutQuestionmark;
          value: ParseValue<V> | null;
        }
      : {
          key: K;
          value: ParseValue<V>;
        }
    : never;
```

Whitespace

You may have noticed the lack of whitespace in the examples above. However, that doesn't seem to square with how we intend for templates to be written by users.

```
const schema = compileSchema(`{
  name: string;
  email: string;
}`);
```

TypeScript template literals are whitespace sensitive, which we can sidestep by stripping out all whitespace from the input string before processing it.

We do that, of course, using `infer` and recursion.

```
type RemoveSpaces<T extends string> = T extends `${infer L} ${infer R}`
  ? RemoveSpaces<`${L}${R}`>
  : T;

type RemoveTabs<T extends string> = T extends `${infer L}\t${infer R}`
  ? RemoveTabs<`${L}${R}`>
  : T;
```

```
  ? RemoveNewlines<`${L}${R}`>
  : T;

type RemoveWhitespace<T extends string> =
  RemoveSpaces<RemoveTabs<RemoveNewlines<T>>>;

Equals<
  RemoveWhitespace<`{\n  hello: { world: string;\n}`>,
  `{hello:{world:string;}`,
>;
```

We can apply this by wrapping the input string to the top-level parsing type with `RemoveWhitespace`.

```
type Parse<T extends string> = ParseObject<RemoveWhitespace<T>>;
```

Outro

I hope I was successful in showing how powerful and versatile the `infer` keyword and recursive types are in TypeScript.

The source code for this article is available for you to take a look at. Feel free to tinker, extend the code to support new features, or change up the syntax entirely!

If you would like to take a look at the open-source library I wrote, check out `strema` on GitHub. It's a more mature version of what we implemented in this article.

It implements:

- Hash maps
- Rules (such as `<email>`)
- Default values

alexharri

- Custom type errors at compile-time
- A runtime template parser and data validator

Anyways, thanks for reading!