6. Loops with Recursive Types

In our journey to learn the language of types, we've already gone pretty far. In the previous chapter, we started **computing** types from other types using conditional logic. We used the type system as a programming language for the first time and in this chapter, I want us to explore this idea even further. After code branching, it's time to learn how to write type-level **loops**!

Loops make programming **feel like magic** sometimes. It's a wonder that computers can solve such complex problems so quickly given the right algorithm. **Computers just excel at repetitive tasks** and loops let us tap into their power. There is no reason not to take advantage of this at the type level too!

In this chapter, we will use **recursion** to loop over tuple types. If you aren't already well-versed in recursive algorithms, the code I'm going to show may look unfamiliar. Bear with me though, and remember that we are not only learning a new programming language but a **functional** one! It takes time for all of these new concepts to sink in, and being open to some discomfort is the key to making progress.

Let's dive into it!

Two styles of loops

Most programming languages we use today are spiritual children of the **C** language. They have a similar syntax and are grounded in an **imperative** programming style. In the imperative style, we tell the computer what to do in a series of instructions that should be executed sequentially, in the right order.

Imperative programs look a bit like this:

```
// imaginary imperative language 🤖
do that;
while (condition is true) {
  do this repetitive thing;
}
// and don't ask questions, computer!
```

To tell the computer that a block of code **should be executed several times**, we have **keywords** like `while` or `for`. This programming style is definitely the most common. Python, JavaScript, Ruby, Go, and many, many other languages support this.

Type-level TypeScript, however, does **not** support imperative programming. Just like other functional languages, it tends to use **functions for everything**, including **loops!** If we want to perform a repetitive task, we just call the same function over and over again.

Here is a **functional loop** written in JavaScript:

```
// Using JS as a functional language 🌈
const doRepetitiveTask = (some, input) ⇒
  condition === true
    ? doRepetitiveTask(again, withSomeOtherInput) // ← recursion
    : something;
```

Our `doRepetitiveTask` function calls itself from the **inside** in order to keep looping. This is called **recursion** and this is what we will learn about today!

Here is our previous example again, but this time written in the language of types:

```
type DoRepetitiveTask<Some, Input> =
  Condition extends true
    ? DoRepetitiveTask<Again, WithSomeOtherInput>
    : Something
```

Notice that **we did *not* need any new syntax** to write a loop. We already covered all the necessary building blocks in previous chapters:

- Defining and calling functions with **Generic Types**.
- Code branching with **Conditional Types**.

A recursive loop always contains a condition. If it did not, the loop would never stop and the type-checker wouldn't be happy about your code. That's why the official documentation refers to these types as *Recursive Conditional Types*.

Using recursion instead of a for loop may feel like a constraint at first, but know that **every imperative loop can be rewritten as a recursive one**, without exception. Mutating temporary variables is replaced by passing different arguments to our recursive function, and breaking the loop is just returning a value (or a type!)

Enough with pseudo code. Let's see a real example!

Looping over Tuples

First, we need a data structure to loop over. We will start with tuple types which, as you know, are **the type-level equivalent of arrays**. Recursively looping on an array consists of **three steps**:

- **Splitting** the list into two parts: the **first** element and the **rest** of the list.
- **Computing** something using the **first** element.
- **Recursively calling** the function on the **rest** of the list.

Let's illustrate this with a concrete case: **finding an element in a list**.

A type-level `find` function

Let's say our app needs to store some **tabular** data. We need a `Table` type that contains **several columns**. Each column has a name and a list of values of potentially different types:

```ts
// Columns contain lists of values
type Column = {
  name: string;
  values: unknown[];
};

// A table is a non-empty list of columns
type Table = [Column, ...Column[]];

// `UserTable` is a subtype of `Table`:
type UserTable = [
  { name: "firstName"; values: string[] },
  { name: "age"; values: number[] },
  { name: "isAdmin"; values: boolean[] },
];
```

Now, let's say we want to write a `getColumn(table, columnName)` function that takes a **table**, a **column name**, and returns **the values** inside this column. We also would like the return type of `getColumn` to be inferred based on the provided column name:

```ts
const users: UserTable = [
  { name: "firstName", values: ["Gabriel", "Bob", "Alice"] },
  { name: "age", values: [29, 43, 31] },
  { name: "isAdmin", values: [true, false, false] },
];

const firstNames = getColumn(users, "firstName");
// We would like `firstNames` to be inferred as a `string[]`

const isAdmins = getColumn(users, "isAdmin");
// and `isAdmins` to be inferred as a `boolean[]`
```

💡 Press the "Edit" button at the top-right corner of any code block to see type-checking in action.

What should the type *signature* of this function look like? Probably something like this:

```ts
declare function getColumn<
  T extends Table,
  N extends string
>(table: T, columnName: N): GetColumn<T, N>;
/*                           ~~~~~~~~~~~~~~
 *                                ^ 🤔
 */
```

We need to define a type-level `GetColumn` function that finds the right column in the list:

```ts
type Result1 = GetColumn<UserTable, "firstName">;
//         ^?  string[]

type Result2 = GetColumn<UserTable, "isAdmin">;
//         ^?  boolean[]
```

Sounds good. So what does this `GetColumn` generic look like? 👺

```ts
type GetColumn<List, Name> = // 😇
  List extends [infer First, ...infer Rest]
    ? First extends { name: Name, values: infer Values }
      ? Values
      : GetColumn<Rest, Name>
    : undefined;
```

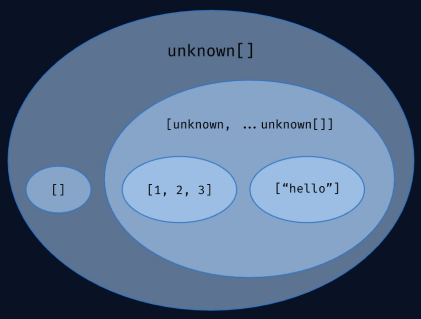
Wow. This is by far the most complex type we've seen in this course! Let's decompose it into smaller chunks that are easier to grasp.

As we've discussed, to loop on a list we first need to split it into two pieces — the **first element** and the **rest** of the list. For this, we use a conditional type in combination with the `infer` keyword:

```ts
type GetColumn<List, Name> =
  List extends [infer First, ...infer Rest]
    ? ... //   ^ we pattern-match on the list!
    : ...
```

Remember `infer`? It lets us assign a variable by **pattern matching** on the type on the left-hand side of `extends`. Using `infer`, we can assign the first element to a `First` variable, and the rest of the list to a `Rest` variable. We now have access to these two variables in the *truthy* code branch.

But what about the *falsy* code branch? It can only be reached if the `List` variable is not assignable to `[unknown, ...unknown[]]`. If we assume `GetColumn` is always given a well-formed tuple type, this can only happen if the list is **empty** – if `List` contains the `[]` type.



`unknown[]` includes the empty list type, but `[unknown, ...unknown[]]` does not.

If the list is empty, we return the `undefined` type:

```ts
type GetColumn<List, Name> =
  List extends [infer First, ...infer Rest]
    ? ...
    : undefined;
    /* ^ We choose to return undefined if the
         list is empty.                      */
```

So far so good? Great. Now, what do we do with the `First` and the `Rest` variables?

Since we are trying to find the column with the correct name, we'd better check if our `First` variable isn't this very column we are looking for! We need to check if its `"name"` property is assignable to our `Name` type parameter. Let's use another conditional type:

```ts
type GetColumn<List, Name> =
  List extends [infer First, ...infer Rest]
    ? First extends { name: Name, values: infer Values }
      /*                    👆
        if the name is the right one
```

```
        we return the `Values` property!
          👇                              */
      ? Values
      : ...
  : undefined;
```

if `First` is assignable to `{name: Name, values: infer Values}`, it means the name of this column is assignable to our `Name` parameter. In this case, we found our column! We can just return its `Values` property. Otherwise, though, what should we do?

Nothing!

Well, almost. We only need to **recurse** on the **rest** of the list:

```
type GetColumn<List, Name> =
  List extends [infer First, ...infer Rest]
    ? First extends { name: Name, values: infer Values }
      ? Values
      : GetColumn<Rest, Name>
      /*           👆
         Call the same function with
            the rest of the list!     */
    : undefined;
```

That's it! At some point, `GetColumn<Rest, Name>` will either return the right column or `undefined` because all *edge cases* are already covered. Now we can use this function with any table:

```
declare const products: [
  { name: "productName"; values: string[] },
  { name: "price"; values: number[] },
  { name: "location"; values: [lat: number, long: number][] },
];

const locations = getColumn(products, "location");
//            ^?  [lat: number, long: number][]

const result2 = getColumn(products, "oops");
//          ^?  undefined
```

Awesome!

Want to know the best part of the story? All recursive loops use this code structure. If you understood this example, you are all set to understand any recursive algorithm!

The Structure of a recursive loop

All recursive loops share a similar structure. The pattern we have seen in the previous example will come up over and over again:

```
type SomeLoop<List /* ... other params */> =

  // 1. Split the list:
  List extends [infer First, ...infer Rest]

    // 2. Compute something using the first element.
    //    Maybe recurse on the `Rest`:
    ? SomeLoop<Rest /* ... modified params */>

    // 3. Return a default type if the list is empty:
    : SomeDefault;
```

If you have been writing JavaScript or TypeScript for a while, you probably know about `map`, `filter` and `reduce`. These are **native array methods** that *encapsulate* loops. Most imperative loops can be rewritten as a *composition* of those three methods.

```
return [1, 2, 3, 4]
  .map((x) => x * x)              // [1, 4, 9, 16]
  .filter((x) => x > 5)           // [9, 16]
  .reduce((sum, x) => sum + x, 0); // 25
```

`map`, `filter` and `reduce` are *Higher-Order Functions*. This barbarian name only means these functions take other functions as parameters. At the type level, it is unfortunately not possible to pass a function as a parameter to another function [1]. Because of this limitation, we won't be able to abstract over common loops the way we can in JavaScript, and we'll have to write recursive code by

hand whenever we want to loop through a data structure.

In the spirit of showing the **correspondence** between **value level** and **type level**, let's have a look at examples of loops to map, filter and reduce tuple types.

`map` loops

Mapping over a list means **transforming each element** into a different value. For example, transforming a list of **users** into a list of **names** is a "map" operation:

```
type Names = ToNames<[
  { id: 1; name: "Alice" },
  { id: 2; name: "Bob" }
]>;
// ⇒ ["Alice", "Bob"]

type ToNames<List> =
  List extends [infer First, ...infer Rest]
    ? [GetName<First>, ...ToNames<Rest>]
    : [];

type GetName<User> =
  User extends { name: infer Name } ? Name : "Anonymous";
```

If the list is not empty, **we transform its first element** using `GetName<First>` and we **prepend** it to the other names by calling `ToNames` on the rest of the list. If the list is empty, we simply return an empty list.

Calling `GetName<First>` is the only part of `ToNames` that is **specific** to our current problem of turning users into names. The rest of the code can be reused *as is* for any other map loop:

```
type SomeMapLoop<List> =
  List extends [infer First, ...infer Rest]
    ? [ /* ... 👈 your logic */ , ...SomeMapLoop<Rest>]
    : [];
```

All "map" loops share this **structure**!


```
  { id: 1; name: "Alice" },
  { id: 2; name: "Bob" }
]>;
// ⇒ ["Alice", "Bob"]

type ToNames<List> =
  List extends [infer First, ...infer Rest]
    ? [GetName<First>, ...ToNames<Rest>]
    : [];

type GetName<User> =
  User extends { name: infer Name } ? Name : "Anonymous";
```

If the list is not empty, **we transform its first element** using `GetName<First>` and we **prepend** it to the other names by calling `ToNames` on the rest of the list. If the list is empty, we simply return an empty list.

Calling `GetName<First>` is the only part of `ToNames` that is **specific** to our current problem of turning users into names. The rest of the code can be reused *as is* for any other map loop:

```
type SomeMapLoop<List> =
  List extends [infer First, ...infer Rest]
    ? [ /* ... 👈 your logic */ , ...SomeMapLoop<Rest>]
    : [];
```

All "map" loops share this **structure**!

```
  { id: 1; name: "Alice" },
  { id: 2; name: "Bob" }
]>;
// ⇒ ["Alice", "Bob"]

type ToNames<List> =
  List extends [infer First, ...infer Rest]
    ? [GetName<First>, ...ToNames<Rest>]
    : [];

type GetName<User> =
  User extends { name: infer Name } ? Name : "Anonymous";
```

If the list is not empty, **we transform its first element** using `GetName<First>` and we **prepend** it to the other names by calling `ToNames` on the rest of the list. If the list is empty, we simply return an empty list.

Calling `GetName<First>` is the only part of `ToNames` that is **specific** to our current problem of turning users into names. The rest of the code can be reused *as is* for any other map loop:

```
type SomeMapLoop<List> =
  List extends [infer First, ...infer Rest]
    ? [ /* ... 👈 your logic */ , ...SomeMapLoop<Rest>]
    : [];
```

All "map" loops share this **structure**!

```
  { id: 1; name: "Alice" },
  { id: 2; name: "Bob" }
]>;
// ⇒ ["Alice", "Bob"]

type ToNames<List> =
  List extends [infer First, ...infer Rest]
    ? [GetName<First>, ...ToNames<Rest>]
    : [];

type GetName<User> =
  User extends { name: infer Name } ? Name : "Anonymous";
```

If the list is not empty, **we transform its first element** using `GetName<First>` and we **prepend** it to the other names by calling `ToNames` on the rest of the list. If the list is empty, we simply return an empty list.

Calling `GetName<First>` is the only part of `ToNames` that is **specific** to our current problem of turning users into names. The rest of the code can be reused *as is* for any other map loop:

```
type SomeMapLoop<List> =
  List extends [infer First, ...infer Rest]
    ? [ /* ... 👈 your logic */ , ...SomeMapLoop<Rest>]
    : [];
```

All "map" loops share this **structure**!

we do **not** update the result of the *recursive* call in any way before returning it:

```
// FromEntries is Tail-Recursive
type FromEntries<Tuple, Acc = {}> = ...
  ? FromEntries< ... > // ← Because we just return the recursive
  //                      result, without modifying it.
  : ...;
```

This is an important detail because TypeScript can perform tail-recursion elimination on tail-recursive types, a performance optimization we will learn about in the future. For now, just know tail-recursive functions can take larger inputs and have better type-checking performance than regular recursive functions.

Recursion and Type Constraints

In the previous chapter, we learned that putting type constraints on type parameters was a nice way of ensuring that **no unexpected inputs** could be given to one of our type-level functions. Constraints are filling the role of types for our types!

```
type Append<Tuple extends any[], Element> = [...Tuple, Element];
//                          👆
//       `tuple` has to be assignable to `any[]`

type T1 = Append<[1, 2], 3>; // ✅ [1, 2, 3]
type T2 = Append<{ oops: "!" }, 3>;
/*               ~~~~~~~~~~~~
                 ^ ❌ not assignable to any[]
```

As our types become more complex, this extra safety is even more relevant, so why haven't we been putting constraints on parameters in previous examples?

Well, because type constraints do not play nicely with conditional types. Variables declared with `infer` **do not inherit** the **constraint** of their parent type:

```ts
type User = { name: string };

type ToNames<List extends User[]> =
  // Expects a list of users 👆
  List extends [{ name: infer Name }, ...infer Rest]
    ? [Name, ...ToNames<Rest>]
    /*                  ~~~~
          ❌ Type 'Rest' does not satisfy
             the constraint 'User[]'.       */
    : [];
```

Even though we're 100% sure that `Rest` is assignable to `User[]` here, TypeScript loses track of this information. That's a real bummer because our recursion no longer type-checks![2]

Luckily, there are ways to circumvent this limitation of the type-checker. Let me introduce you to two of them.

1. Adding a constraint to our `infer` type variable

Since version 4.7, it's possible to add a type constraint when declaring a type variable with `infer`. This way, we can tell the type-checker that `Rest` is in fact assignable to `User[]`:

```ts
type User = { name: string };

type ToNames<List extends User[]> =
  List extends [
    { name: infer Name },
    ...infer Rest extends User[] // ← constraint
  ]
    ? [Name, ...ToNames<Rest>] // ✅
    : [];
```

This is a bit verbose and repetitive, but this works!

2. Inferring constraints with conditional types

Another way to convince TypeScript that `Rest` is assignable to `User[]` without resorting to new syntax is to use a conditional type **before** recursing:

```ts
type ToNames<List extends User[]> =
  List extends [{ name: infer Name }, ...infer Rest]
    ? [
        Name,
        ...ToNames<Rest extends User[] ? Rest : never> // ✅
      ]
    : [];

// ToNames still works as expected:
type Names = ToNames<[
  { id: 1; name: "Alice" },
  { id: 2; name: "Bob" }
]>;
// ⇒ ["Alice", "Bob"]
```

TypeScript **infers a type constraint** on `Rest` from the **conditional type**! To make our code a little cleaner, let's create an `As` generic that handles the "casting" for us:

```ts
type As<A, B> = A extends B ? A : never;

type ToNames<List extends User[]> =
  List extends [{ name: infer Name }, ...infer Rest]
    ? [Name, ...ToNames<As<Rest, User[]>>] // ✅
    : [];
```

All in all, both of these options make the code more verbose. Keep in mind that not using a constraint is often a viable option too. I personally tend to only add constraints when I need to use some syntax that isn't available for all types like the `...` spread syntax, which requires an `any[]` constraint.

Now you are well equipped to unblock yourself when you'll face this limitation!

Summary 📚

I find Recursive Types super exciting because they are so **versatile**. Once you start wrapping your head around recursive algorithms, you can design your types in a way that captures more code ***invariants*** and provide **friendlier** and **safer APIs**.

Let's summarize the most important notions of this chapter:

1. In Type-Level TypeScript loops are defined **recursively** and not imperatively.
2. Any imperative loop can be **rewritten** as a recursive one.
3. To loop over a tuple type, you always perform the **3 same steps**: **split** the list, **use** the first element and **recurse** on the rest of the list.
4. Type constraints can be challenging to use with recursion, but there are ways to make them work!

Recursive loops unlock so many cool algorithms that we didn't cover yet. In the next chapter, we will use them to parse string literals and build **type-safe** Domain Specific Languages! 🤗

Challenges 🎖️

As always, it's time to put what we've seen into practice! These challenges are harder than those of previous chapters, so don't feel bad if you don't manage to solve all of them 🙂

Let's go! 👇

Challenge Solution Format ✨ Reset

```ts
/**
 * Implement an `AssignAll` generic that takes a tuple
 * containing object types and merges all of them.
 *
 * Bonus: make it *Tail-Recursive*.
 */
namespace assign {

  declare function assign<
    // Infer `Objects` as a tuple of objects:
    Objects extends [{}, ...{}[]]
  >(...objects: Objects): AssignAll<Objects>;

  type AssignAll<Tuple> = TODO

  // Two objects
  const res1 = assign({ name: "Michel", age: 82 }, { childrenCount: 3 });
  type expected1 = { name: string; age: number, childrenCount: number };
  type test1 = Expect<Equal<typeof res1, expected1>>;
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Type the `all` function to take a list of promises and
 * to turn them into a single promise containing a list of values.
 */
namespace promiseAll {

  declare function all<
    // Infer `Promises` as a tuple of promises:
    Promises extends [Promise<any>, ...Promise<any>[]]
  >(promises: Promises): Promise<UnwrapAll<Promises>>;

  type UnwrapAll<Promises> = TODO

  // Two promises
  const res1 = all([Promise.resolve(20), Promise.resolve("Hello" as const)]);
  type expected1 = Promise<[number, "Hello"]>;
  type test1 = Expect<Equal<typeof res1, expected1>>;

  // Three promises
  const res2 = all([Promise.resolve(true), Promise.resolve("!"), Promise.resolve({})])
```

Challenge Solution Format ✨ Reset

```ts
/**
 * Type the `filterTable` function to take a Table,
 * a list of column names, and to return a table that
 * only contains columns with these names.
 */
namespace filterTable {

  type Column = { name: string, values: unknown[] }

  declare function filterTable<
```

```
  declare function filterTable<
    // Infer `T` as a tuple containing columns:
    T extends [Column, ...Column[]],
    // Infer `N` as a union of string literal type:
    N extends string
  >(table: T, columnNames: N[]): FilterTable<T, N>;

  type FilterTable<Table, NameUnion> = TODO

  declare const userTable: [
    { name: 'firstName', values: string[] },
```

Challenge | Solution — Format ✨ | Reset

```
/**
 * Type lodash's `zip` function.
 * `zip` takes several arrays containing different types of
 * values, and turn them into a single array containing
 * tuples of values for each index.
 *
 * For example, `zip([1, 2], [true, false], ['a', 'b'])`
 * returns `[[1, true, 'a'], [2, false, 'b']]`.
 */
namespace zip {
  declare function zip(...arrays: TODO): TODO;

  const res1 = zip([1, 2], [true, false]);
  // => [[1, true], [2, false]]
  type test1 = Expect<Equal<typeof res1, [number, boolean][]>>;

  const res2 = zip([1, 2], [true, false], ['a', 'b']);
  // => [[1, true, 'a'], [2, false, 'b']]
  type test2 = Expect<Equal<typeof res2, [number, boolean, string][]>>;
```

Bonus challenges

Here are some additional type-only exercises to keep flexing your type-level muscles 💪

Challenge | Solution — Format ✨ | Reset

```
/**
 * Make a `Filter` generic that takes a tuple,
 * an arbitrary `Cond` type, filters out any element
 * that isn't assignable to `Cond`.
 */
namespace filter {
  type Filter<Tuple, Cond> = TODO

  type res1 = Filter<[1, 2, "oops", 3, "hello"], number>;
  type test1 = Expect<Equal<res1, [1, 2, 3]>>;

  type res2 = Filter<["a", 1, "b", true, "c"], string>;
  type test2 = Expect<Equal<res2, ["a", "b", "c"]>>;

  type res3 = Filter<["hello", null, 42, {}, [], undefined], {}>;
  type test3 = Expect<Equal<res3, ["hello", 42, {}, []]>>;
  //                                ^      ^
  // Note: strings and numbers are assignable to the `{}` object type.
}
```

Challenge | Solution — Format ✨ | Reset

```
/**
 * Write a `WithIndex` type level function
 * that takes a tuple, and maps it to a tuple
 * of [value, index] pairs.
 *
 * Hint: you will need to use T["length"]
 * to generate indices
 */
namespace withIndex {
  type WithIndex<
    Tuple extends any[],
    Output extends any[] = []
  > = TODO

  type res1 = WithIndex<['a']>;
  type test1 = Expect<Equal<res1, [['a', 0]]>>;

  type res2 = WithIndex<['a', 'b']>;
  type test2 = Expect<Equal<res2, [['a', 0], ['b', 1]]>>;
```

Challenge | Solution

Format ✨ | Reset

```ts
/**
 * Write a `Take` type level function
 * that takes a tuple, a `N` number and
 * returns the first `N` elements of this
 * tuple.
 *
 * Hint: you will need to use T["length"]
 * to read the length of a tuple `T`.
 */
namespace take {
  type Take<
    Tuple extends any[],
    N,
    Output extends any[] = []
  > = TODO

  type res1 = Take<[1, 2, 3], 2>;
  type test1 = Expect<Equal<res1, [1, 2]>>;

  type res2 = Take<[1, 2, 3], 1>;
```

Footnotes

1. Even though passing functions to functions isn't natively supported at the type level, there is a smart trick to emulate this feature. I'm not covering it in this chapter because it's complicated and hacky, but if you are curious, you should check HotScript out. It's an open-source library of **type-level Higher Order Functions** that I created. ↩

2. This is a current limitation of TypeScript < v4.9 that is being discussed in this GitHub issue and will hopefully be fixed in an upcoming version. ↩

⟸ Previous
5. Code branching with Conditional Types

Next ⟹
7. Template Literal Types