Type Argument Propagation

This lesson talks about an interesting language feature related to generic functions introduced in TypeScript 3.4.

WE'LL COVER THE FOLLOWING

- Overview
- Pointfree style
- Propagated generic type arguments
- Why does this matter?

Overview

Let's have a look at the following example. Imagine that you're fetching a collection of objects from some backend service and you need to map this collection to an array of identifiers.

```
interface Person {
  id: string;
  name: string;
  birthYear: number;
}

function getIds(persons: Person[]) {
    return persons.map(person => person.id);
}
```

Next, you decide to generalize the `getIds` function so that it works on any collection of objects that have the `id` property.

```
function getIds<T extends Record<'id', string>>(elements: T[]) {
  return elements.map(el => el.id);
}
```

Fair enough. However, the code for this simple function is quite verbose. Can

we make it more concise?

Pointfree style

We can do this by taking advantage of a functional programming technique called **pointfree style**. Ramda is a nice library that will let us compose this function from other functions: `map` and `prop`.

```
import * as R from 'ramda';

const getIds = R.map(R.prop('id'));
```

`map` is *partially applied* with a mapper function, `prop`, which extracts the `id` property from any object. The result of `getIds` is a function that accepts a collection of objects.

Sadly, TypeScript (pre 3.4) has bad news for you. The type of `getIds` is inferred to `(list: {}) => {}` which is not exactly what you'd expect.

You can explicitly type `map` but it makes the expression really verbose:

```
const getIds = R.map<Record<'id', string>, string>(R.prop('id'));
```

This is where *propagated generic type arguments* come in. In TypeScript 3.4 the type of `getIds` will correctly infer to `<T>(list: readonly Record<"id", T>[]) => T[]`. Success!

Propagated generic type arguments

Now that we know what *propagated generic type arguments* are about, let's decipher the name.

`R.map(R.prop('id'))` is an example of a situation when we pass a generic function as an argument to another generic function.

Before version 3.4 of TypeScript, the type of parameters of the inner function type was not *propagated* to the result type of the call.

Why does this matter?

Even if you're not particularly excited about pointfree style programming,

keep in mind that some popular libraries rely on function composition and

partial application and will benefit from this change.

For example, in RxJS it is possible to compose new operators from existing ones using the `pipe` **function** (as opposed to the `pipe` method). TypeScript 3.4 will certainly improve typing in such scenarios.

Other examples include Redux (`compose` for middleware) and Reselect.