Lekcja 37: Łączenie przeciążeń funkcji i typów warunkowych

W poprzedniej lekcji stwierdziliśmy, że typy warunkowe są w stanie opisać wszystko, co może zrobić przeciążenie funkcji, i są znacznie bardziej poprawne. Chociaż jest to technicznie prawdziwe, istnieją scenariusze, w których zdrowe połączenie przeciążeń funkcji i typów warunkowych zapewnia znacznie lepszą czytelność i wyraźniejsze wyniki. Jeden z takich scenariuszy dotyczy argumentów opcjonalnych. Funkcja fetchOrder jest synchroniczna. Jak wiemy, pobieranie czegoś z bazy danych lub zaplecza przez większość czasu odbywa się asynchronicznie. Zrefaktoryzujmy fetchOrder, aby umożliwić asynchroniczne pobieranie danych. Funkcja powinna łączyć dwa różne wzorce asynchroniczne:

1. Jeśli przekażemy jeden argument (number, Customer lub Product), otrzymamy w zamian obietnicę z odpowiednim wynikiem (Order lub Order[]).

2. Jako drugi argument możemy przekazać callback. To wywołanie zwrotne pobiera wynik (Order lub Order []) jako parametr; funkcja fetchOrder zwraca void.

To klasyczny wzorzec, który możemy zobaczyć w wielu bibliotekach Node.js. Albo przekażemy wywołanie zwrotne, albo zwrócimy obietnicę. Ciekawą częścią tego przykładu jest to, że drugi argument jest całkowicie opcjonalny. Oznacza to, że kształt funkcji może być bardzo różny. Spójrzmy na każdą głowicę funkcji oddzielnie.

```
// A callback helper type

type Callback<Res> = (result: Res) => void

// Version 1. Similar to the version from

// the previous lesson, but wrapped in a promise

function fetchOrder<Par extends FetchParams>(

inp: Par

): Promise<FetchReturn<Par>>

// Version 2. We pass a callback function that

// gets the result, and return void.

function fetchOrder<Par extends FetchParams>(

inp: Par, fun: Callback<FetchReturn<Par>>

): void
```

Przy tak różnym kształcie funkcji nie wystarczy wykonać typ warunkowy dla prostej sumy.

Typy krotki dla nagłówków funkcji

Możliwym rozwiązaniem byłoby wykonanie typu warunkowego dla sumy całego zestawu głowic funkcyjnych. W JavaScript mamy możliwość skondensowania wszystkich argumentów funkcji w krotkę z pozostałymi parametrami.

```
function doSomething(...rest) {

return rest[0] + rest[1]

}
```

// Returns “JavaScript”

doSomething(‘Java’, ‘Script’)

Ten parametr reszta można wpisać jako krotkę. Wpiszmy argumenty wersji wywołania zwrotnego jako krotka:

function fetchOrder<Par extends FetchParams>(

...args: [Par, Callback<FetchReturn<Par>>]

): void

Wpiszmy też argumenty wersji obietnicy jako krotkę.

function fetchOrder<Par extends FetchParams>(

...args: [Par]

): Promise<FetchReturn<Par>

Podsumowujemy całą listę argumentów każdej nagłówka funkcji na oddzielne typy krotek. Oznacza to, że możemy utworzyć typ warunkowy, który wybiera właściwy typ wyjścia.

// A small helper type to make it easier to

// read

type FetchCb<T extends FetchParams> =

Callback<FetchReturn<T>>

type AsyncResult<

FHead, Par extends FetchParams

> = FHead extends [Par, FetchCb<Par>>] ? void :

FHead extends [Par] ? Promise<FetchReturn<T>> :

never;

Typ warunkowy ma następującą postać:

1. Jeśli nagłówek funkcji FHead jest podtypem krotki FetchParams i FetchCb, zwróć void.

2. W przeciwnym razie, jeśli nagłówek funkcji jest podtypem krotki FetchParams, zwróć obietnicę.

3. W przeciwnym razie nigdy nie zwraca

Możemy użyć tego nowo utworzonego typu warunkowego i powiązać go z naszą funkcją.

function fetchOrder<

Par extends FetchParam,

FHead

>(...args: FHead) : AsyncResult<FHead, Par>

I to prawie załatwia sprawę. Ale ma też wysoką cenę:

1. Czytelność. Typy warunkowe są już trudne do odczytania. W tym przypadku mamy dwa zagnieżdżone typy warunkowe: stary FetchReturn, który niezawodnie zwraca odpowiedni typ zwrotu; oraz nowy AsyncResult, który informuje nas, czy otrzymamy unieważnienie lub obietnicę z powrotem.

2. Poprawność. Gdzieś po drodze możemy utracić informacje o powiązaniach dla naszych parametrów typu ogólnego. Oznacza to, że nie otrzymujemy rzeczywistego typu zwracanego, ale sumę wszystkich możliwych typów zwracanych. Upewnienie się, że niczego nie stracimy, wymaga od nas wielu wiązań

parametry, w ten sposób zapełniając nasze generyczne sygnatury i ogólne ograniczenia. W takich przypadkach lepszym pomysłem może być nadal poleganie na przeciążeniach funkcji.

Przeciążenia funkcji są w porządku

Przewiń do naszego początkowego opisu funkcji. Opisaliśmy dwie możliwe wersje i ich wyniki. To dokładnie odzwierciedla przeciążenia funkcji, które wykonalibyśmy bez typów warunkowych. Zobaczmy więc, jak możemy zaimplementować cały zestaw możliwych funkcji:

```
// Wersja 1

function fetchOrder <Par rozszerza FetchParams> (

inp: par

): Obietnica <FetchReturn <Par>>

// Wersja 2

function fetchOrder <Par rozszerza FetchParams> (

inp: Par, fun: Callback <FetchReturn <Par>>

): nieważne

// Implementacja!

function fetchOrder <Par rozszerza FetchParams> (

inp: Par, fun ?: Callback <FetchReturn <Par>>

): Promise <FetchReturn <Par>> | void {

// Pobierz wynik

const res =

pobierz (`/ backend? inp = $ {JSON.stringify (inp)}`)

. then (res => res.json ())

// Jeśli jest oddzwonienie, zadzwoń

if (fun) {

res.then (wynik => {

zabawa (wynik)

})

} else {
```

// W przeciwnym razie zwróć obietnicę wyniku

powrót res

}

}

Jeśli przyjrzymy się uważnie, zobaczymy, że nie zostawiamy całkowicie typów warunkowych. Sposób, w jaki traktujemy typ FetchReturn, jest nadal typem warunkowym, opartym na typie unii FetchParams. Różnorodność wejść i wyjść została ładnie skondensowana w jeden typ. Jednak złożoność różnych głowic funkcyjnych była lepiej dostosowana do przeciążeń funkcji. Zachowanie wejścia i wyjścia jest jasne i łatwe do zrozumienia, a kształt funkcji jest na tyle inny, że kwalifikuje się do jawnego zdefiniowania. Ogólna zasada dotycząca twoich funkcji:

1. Jeśli argumenty wejściowe opierają się na typach unii i musisz wybrać odpowiedni typ zwracanej wartości, najlepszym rozwiązaniem jest typ warunkowy.

2. Jeśli kształt funkcji jest inny (np. Argumenty opcjonalne), a związek między argumentami wejściowymi a typami wyjściowymi jest łatwy do zrozumienia, przeciążenie funkcji załatwi sprawę.