Lekcja 43: Promisfy

Gdy przyzwyczaisz się do pisania obietnic, powrót do starego sposobu wywołania zwrotnego w programowaniu asynchronicznym staje się coraz trudniejszy. Jeśli jesteś podobny do mnie, pierwszą rzeczą, którą musisz zrobić, jest zawrzeć wszystkie funkcje typu callback w obietnicę i przejść dalej. Dla mnie stało się to wzorcem, którego używam tak często, że napisałem własną funkcję promisify: funkcję, która pobiera każdą inną funkcję opartą na wywołaniu zwrotnym i tworzy z niej funkcję w stylu obietnicy. Oto, czego oczekujemy od zachowania:

```
// e.g. a function that loads a file into a string

declare function loadFile(

fileName: string,

cb: (result: string) => void

);

// We want to promisify this function

const loadFilePromise = promisify(loadFile)

// The promised function takes all arguments

// except the last one and returns a promise

// with the result

loadFilePromise('./chapter7.md')

// which is a string according to loadFile

.then(result => result.toUpperCase())
```

Myśląc o typach, zawsze zaczynamy od zadeklarowania funkcji i upewnienia się, że wiemy, czego się spodziewać jako dane wejściowe i wyjściowe. Dane wejściowe to funkcja, którą można obiecać, co oznacza, że na końcu funkcji ma wywołanie zwrotne. Dane wyjściowe to obiecana funkcja, co oznacza funkcję, która przyjmuje wiele argumentów i zwraca obietnicę. Funkcja z wywołaniem zwrotnym jako wejściem i obiecaną funkcją jako wyjściem. Istnieją nasze typy danych wejściowych i wyjściowych:

```
declare function promisify<

Fun extends FunctionWithCallback

>(fun: Fun): PromisifiedFunction<Fun>;
```

Teraz zamodelujmy nasze nowo utworzone typy. FuntionWithCallback jest dziwne, ponieważ musi pracować dla potencjalnie nieskończonej listy argumentów, zanim dotrzemy do ostatniego, wywołania zwrotnego. Jednym ze sposobów na osiągnięcie tego byłaby lista przeciążeń funkcji, w której upewniamy się, że ostatni argument jest wywołaniem zwrotnym. Ale ta metoda musi się gdzieś skończyć. Poniższy przykład zatrzymuje się przy trzech przeciążeniach:

```
type FunctionWithCallback =

(

(
```

```
arg1: any,
cb: (result: any) => any
) => any
) |
(
(
arg1: any,
arg2: any, cb: (result: any) => any
) : any
) |
(
(
arg1: any,
arg2: any,
arg3: any,
cb: (result: any) => any
) : any
)
```

Dużo bardziej elastycznym rozwiązaniem jest zmienna krotka. Typ krotki w TypeScript to tablica z następującymi funkcjami.

1. Długość tablicy jest zdefiniowana.

2. Typ każdego elementu jest znany (i nie musi być taki sam).

Na przykład jest to typ krotki:

```
type PersonProps = [string, number]

const [name, age]: PersonProps = ['Stefan', 37]
```

Argumenty funkcji można również opisać jako typy krotek. Na przykład:

```
declare function hello(name: string, msg: string):
void;
```

jest takie samo jak:

```
declare function hello(...args: [string, string]):
void;
```

Typ krotki wariadycznej to typ krotki, który ma te same właściwości - zdefiniowaną długość i typ każdego elementu jest znany - ale gdzie dokładny kształt nie został jeszcze zdefiniowany. Jest to doskonały przypadek użycia naszej funkcji w stylu wywołania zwrotnego. Ostatni argument musi być funkcją zwrotną; wszystko wcześniej nie zostało jeszcze zdefiniowane. Więc FunctionWithCallback może wyglądać jak to:

type FunctionWithCallback<T extends any[] = any[]> =

(...t: [...T, (...args: any) => any]) => any;

Najpierw definiujemy ogólnyj. Potrzebujemy typów ogólnych do definiowania wariadycznych typów krotek, ponieważ później będziemy musieli zdefiniować kształt. Ten parametr typu ogólnego rozszerza tablicę any [] w celu przechwytywania wszystkich krotek. Domyślnie używamy również any [], aby ułatwić korzystanie z niego. Następnie pojawia się definicja funkcji.

Najpierw definiujemy rodzaj. Potrzebujemy typów ogólnych do definiowania wariadycznych typów krotek, ponieważ później będziemy musieli zdefiniować kształt. Ten parametr typu ogólnego rozszerza tablicę any [] w celu przechwytywania wszystkich krotek. Domyślnie używamy również dowolnego [], aby ułatwić korzystanie z niego. Następnie pojawia się definicja funkcji.

Lista argumentów jest typu [... T, (... args: any) => any]). Najpierw zmienna część, którą definiujemy poprzez użycie, a następnie funkcja wieloznaczna. Gwarantuje to, że przekazujemy tylko funkcje, które mają wywołanie zwrotne jako ostatni argument. Zauważ, że wyraźnie używamy tutaj. Jest to jeden z rzadkich przypadków użycia, w których każdy ma dużo sensu. Nie przejmujemy się jeszcze tym, co przekazujemy jako funkcję, o ile kształt jest nienaruszony. Nie chcemy też, by przeszkadzały nam przekazywanie argumentów. Jest to funkcja pomocnicza i każda będzie dobrze. Następnie zajmijmy się zwracanym typem, obiecaną funkcją. Obiecana funkcja jest typem warunkowym, który sprawdza kształt, który zdefiniowaliśmy w FunctionWithCallback. To tutaj odbywa się faktyczne sprawdzanie typów i gdzie wiążemy typy z rodzajami.

type PromisifiedFunction<T> =

(...args: InferArguments<T>)

=> Promise<InferResults<T>>

PromisifiedFunction <T> pobiera wszystkie argumenty oryginalnej funkcji w stylu wywołania zwrotnego minus wywołanie zwrotne. Otrzymujemy tę listę argumentów za pośrednictwem InferArguments <T>.

type InferArguments<T> =

T extends (

... t: [...infer R, (...args: any) => any]

) => any ? R : never

Warunkowe testy sprawdzają tę samą deklarację typu, którą zdefiniowaliśmy w FunctionWithCallback, ale zamiast przepuszczać argumenty przed wywołaniem zwrotnym, wnioskujemy z całej listy argumentów i zwracamy je. PromisifiedFunction <T> zwraca obietnicę z wynikami zdefiniowanymi w funkcji wywołania zwrotnego. Dlatego musimy wywnioskować wyniki z oryginalnej funkcji w ten sam sposób:

type InferResults<T> =

```
T extends (

...t: [...infer T, (res: infer R) => any]

) => any ? R : never
```

Typ funkcji, który sprawdzamy w naszym warunku, ponownie ma bardzo podobny kształt do FunctionWithCallback. Tym razem jednak chcemy poznać listę argumentów wywołania zwrotnego i wywnioskować z nich. To już działa jak urok. loadFile pobiera poprawne typy, a także funkcje z innymi typami, a inne listy argumentów robią to samo.

```
declare function addAsync(

x: number, y: number,

cb: (result: number) => void

);

const addProm = promisify(addAsync);

// x is number!

addProm(1, 2).then(x => x)
```

Typy są gotowe! Możemy sprawdzić, jak ta funkcja będzie działać po zakończeniu. I to wszystko, co musimy zrobić po stronie TypeScript. Kilka wierszy kodu i znamy dane wejściowe i wyjściowe naszej funkcji. Ponieważ jest to funkcja użyteczna, najprawdopodobniej będziemy używać promisify więcej niż raz. Dobrze spędzony czas na wpisywaniu. Implementacja wymaga również kilku wierszy kodu:

```
function promisify<

Fun extends FunctionWithCallback

>(f: Fun): PromisifiedFunction<Fun> {

return function(...args: InferArguments<Fun>) {

return new Promise((resolve) => {

function callback(result: InferResults<Fun>) {

resolve(result)

}

args.push(callback);

f.call(null, ...args)

})

}

}
```

Tutaj możemy zobaczyć pewne zachowanie odzwierciedlone w JavaScript, które widzieliśmy już w TypeScript. W nowo utworzonej funkcji używamy parametrów reszty - typu InferArguments. Zwraca obietnicę z wynikiem typu InferResults. Aby to uzyskać, musimy utworzyć wywołanie zwrotne,

pozwolić mu rozwiązać obietnicę z wynikami i dodać ją z powrotem do list argumentów. Teraz możemy wywołać oryginalną funkcję z pełnym zestawem argumentów. Jest to operacja odwrotna do tego, co zrobiliśmy z naszymi typami, gdzie zawsze staraliśmy się wyeliminować funkcję wywołania zwrotnego. Tutaj musimy dodać go ponownie, aby był kompletny. Typy są złożone, ale tak rozsądne, że możemy zaimplementować całą funkcję promisify bez rzutowania typów. To dobry znak!