TYPESCRIPT
NA POWAŻNIE

Michał Miszczyszyn

TypeScript na poważnie

Michał Miszczyszyn

26.0.0-6-g4ac076e

Dla mojej żony, Martyny

Projekt okładki: Ewelina Sygut-Pawłowska
Redaktor merytoryczny: Michał Miszczyszyn
Redaktorka prowadząca: Martyna Wygonna-Miszczyszyn
Sugestie merytoryczne: Bartosz Cytrowski, Michał Michalczuk

TypeScript na poważnie
https://typescriptnapowaznie.pl

ISBN e-book EPUB: 978-83-957363-0-8
ISBN e-book MOBI: 978-83-957363-1-5
ISBN e-book PDF: 978-83-957363-2-2
ISBN druk: 978-83-957363-3-9

Wydanie I
Gdańsk 2020

Wydawnictwo Type of Web
email: hi@typeofweb.com
www: https://typeofweb.com

Spis treści

Errata

Dołożyłem wszelkich starań, aby w niniejszej książce znalazły się wyłącznie informacje rzetelne i sprawdzone. Przetestowałem każdy fragment kodu i każdy przykład. Pomimo tego, może się tak zdarzyć, że gdzieś wkradły się błędy.

Jeśli znajdziesz jakiekolwiek pomyłki, to proszę, koniecznie mi to zgłoś! Możesz to zrobić poprzez stronę typeofweb.com/errata. Pod tym adresem znajdziesz również wypisane poprawki, które do książki zostały wprowadzone już po premierze, aby łatwiej Ci było się z nimi zapoznać.

13
Typy warunkowe

Typy warunkowe (*conditional types*) to prawdopodobnie najtrudniejsza część TypeScripta. Jednocześnie, jak to zwykle bywa, jest to element najbardziej potężny i dający ogromne możliwości tworzenia rozbudowanych i zaawansowanych typów, dzięki którym Twoje aplikacje staną się jeszcze bardziej bezpieczne!

13.1 Co to są typy warunkowe?

Conditional types to możliwość wyrażania nieregularnych mapowań typów. Mówiąc prościej, pozwalają na zapisanie takiej transformacji, która wybiera typ w zależności od warunku. Myślę, że przeanalizowanie przykładu powinno rozjaśnić to, co właśnie napisałem. Typ warunkowy zawsze ma taką formę:

```
type R = T extends U ? X : Y;
```

Gdzie `T`, `U`, `X` i `Y` to typy. Notacja `... ? ... : ...` jest analogiczna do operatora trójargumentowego z JavaScriptu i robi dokładnie to, co Ci się wydaje. Przed znakiem zapytania podajemy warunek, w tym przypadku `T extends U`, a następnie wynik, jeśli test zostanie spełniony (`X`) oraz w przeciwnym wypadku (`Y`). Takie wyrażenie oznacza, że jeśli warunek jest spełniony, to otrzymujemy typ `X`, a jeśli nie, to `Y`.

13.2 Przykładowe użycie

Na razie nie było zbyt wielu konkretów, więc spójrzmy na prosty przykład:

```
type IsBoolean<T> = T extends boolean ? true : false;

type t01 = IsBoolean<number>; // false
type t02 = IsBoolean<string>; // false
type t03 = IsBoolean<true>; // true
```

Pamiętaj, że `t01`, `t02` i `t03` to typy, a nie wartości! Nasz *conditional type* `IsBoolean` przyjmuje parametr i sprawdza, czy jest on boolem – odpowiada za to fragment `extends boolean`. Wynikiem jest `true` lub `false` w zależności od tego, czy podany argument spełnia warunek. Co istotne, `true` i `false` tutaj to również typy (literały)! Udało nam się stworzyć wyrażenie wykonywane warunkowo, które zwraca jeden typ w zależności od drugiego.

13.3 Typy warunkowe na unii

Na razie może wydawać się to mało przydatne, ale zaraz się przekonasz o użyteczności takich konstrukcji. Rzućmy okiem na inne użycie:

```
type NonNullable<T> = T extends null | undefined
  ? never
  : T;

type t04 = NonNullable<number>; // number
type t05 = NonNullable<string | null>; // string
type t06 = NonNullable<null | undefined>; // never
```

Do warunkowego `NonNullable` podajemy parametr, a rezultatem jest ten sam typ, ale z usuniętymi `null` i `undefined`. Jeśli po ich wyeliminowaniu nic nie zostaje, to otrzymujemy `never`. Czy za-

czynasz dostrzegać, jakie to może być przydatne? Typy warunkowe pozwalają nam na tworzenie własnych, niejednokrotnie bardzo zaawansowanych mapowań opartych o warunki! Dokładne wyjaśnienie działania tego przykładu znajdziesz w sekcji 13.5.

13.4 Zagnieżdżanie

Co ciekawe, *conditional types* możemy zagnieżdżać. Stwórzmy teraz generyk, który zwraca typ zawierający nazwę podanego parametru:

```
type TypeName<T> =
  T extends string ? "string" :
  T extends number ? "number" :
  T extends boolean ? "boolean" :
  T extends undefined ? "undefined" :
  T extends Function ? "function" :
  T extends Array<any> ? "array" :
  T extends null ? "null" :
  T extends symbol ? "symbol" :
  "object";
```

Może się to wydawać nieco długie i skomplikowane, a na pewno żmudne, ale efekt końcowy jest zadowalający:

```
type t07 = TypeName<string>; // 'string'
type t08 = TypeName<number>; // 'number'
type t09 = TypeName<boolean>; // 'boolean'
type t10 = TypeName<undefined>; // 'undefined'
type t11 = TypeName<function>; // 'function'
type t12 = TypeName<array>; // 'array'
type t13 = TypeName<null>; // 'null'
type t14 = TypeName<symbol>; // 'symbol'
type t15 = TypeName<object>; // 'object'
```

13.5 Warunkowe typy dystrybutywne

Distributive conditional types to cecha typów warunkowych, która sprawia, że ich użycie na unii działa tak, jakbyśmy użyli warunku na każdym z komponentów wchodzących w jej skład osobno, a następnie wyniki połączyli znowu unią. Brzmi skomplikowanie? Ależ skąd! Oba zapisy poniżej oznaczają dokładnie to samo:

```
type t16 = NonNullable<string | null | undefined>;
// string

type t17 =
  | NonNullable<string>
  | NonNullable<null>
  | NonNullable<undefined>;
// string
```

Pierwsze użycie `NonNullable` jest tak naprawdę interpretowane, jak to drugie. Stąd nazwa: komponenty z unii podanej jako parametr są rozdystrybuowane pomiędzy wiele użyć typu warunkowego i każdy z nich sprawdzany jest osobno. Ten podział jest szczególnie wyraźnie widoczny, gdy rezultat jest generykiem:

```
type Ref<T> = { current: T };
type RefVal<T> = T extends number
  ? Ref<T>
  : T extends string
  ? Ref<T>
  : never;

type t18 = RefVal<string>; // Ref<string>;
type t19 = RefVal<string | number>;
// Ref<string> | Ref<number>;
```

Zwróć uwagę, że do `t19` przypisana jest alternatywa dwóch typów `Ref<string>` i `Ref<number>`, a nie `Ref<string | number>`! TypeScript dokonał podziału unii. Można też zaobserwować podob-

ne zachowanie w `TypeName`, a rezultat jest dokładnie taki, jak moglibyśmy tego oczekiwać:

```
type t20 = TypeName<string | number | number[]>;
// "string" | "number" | "array"
```

Zauważ, że `TypeName` zadziałało pomimo tego, że nigdzie nie definiowaliśmy, w jaki sposób ma obsłużyć przypadek, gdy parametrem jest unia. Dzięki dystrybucji komponentów TypeScript rozbił nasz skomplikowany typ na mniejsze składowe, co bardzo ułatwiło nam zadanie.

Głównym zastosowaniem tej cechy typów warunkowych jest filtrowanie unii, a więc tworzenie nowej, z której usunięto część typów. Aby weliminować jakiś komponent z unii należy w warunku zwrócić `never` tak, jak to zrobiliśmy w `NonNullable`. Unia dowolnego typu i `never` daje tylko ten typ[1].

```
type StringsOnly<T> = T extends string ? T : never;
type Result = StringsOnly<"abc" | 123 | "ghi">;
//  "abc" | never | "ghi"
//  "abc" | "ghi"
```

13.6 Przykład użycia

Do czego to wszystko może się nam przydać? Wyobraźmy sobie sytuację, że mamy typ jakiegoś modelu w API, który zawiera zarówno pola, jak i metody. Potrzebujemy taki obiekt zserializować jako JSON i wysłać do użytkownika, a wtedy nie będzie w nim funkcji i pozostaną wyłącznie dane. W związku z tym, chcielibyśmy stworzyć taki typ, w którym będą wszystkie pola naszego modelu z pominięciem metod. Czy jest to możliwe? Zacznijmy od zdefiniowania modelu z dwoma własnościami i jedną funkcją:

[1]Można powiedzieć, że `never` to element neutralny dla unii.

```
type Model = {
  name: string;
  age: number;

  save(): Promise<void>;
};
```

Mogłaby to też być klasa, jak np. przy wykorzystaniu biblioteki Sequelize albo TypeORM. Teraz, chcielibyśmy otrzymać taki sam typ, ale bez `save`. W tym celu musimy wykonać dwa kroki: Pobrać nazwy tych pól, które nas interesują, a następnie stworzyć z nich obiekt. Użyjemy do tego mapowania typów, które dokładnie omawiałem w rozdziale 12:

```
type FieldsNames<T extends object> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];

type OnlyFields<T extends object> = {
  [K in FieldsNames<T>]: T[K];
};

type ModelFields = OnlyFields<Model>;
// { name: string; age: number; }
```

Dzieje się tu bardzo dużo, więc przejdźmy przez oba typy krok po kroku. Zacznijmy do końca:

```
type ModelFields = OnlyFields<Model>;
```

Tę linijkę możemy zastąpić bezpośrednio rozwinięciem `OnlyFields`, aby było nam łatwiej zrozumieć, co robimy:

```
type ModelFields = {
  [K in FieldsNames<Model>]: Model[K];
};
```

Jest to mapowanie typu, które oznacza mniej więcej tyle, że dla każdego pola `K` w typie `FieldsNames<Model>`, tworzymy własność

o nazwie `K` z typem `Model[K]`. Najważniejsze więc, aby zrozumieć, co dzieje się w `FieldsNames<Model>`. Zapiszmy w tym celu pomocniczy typ `A`:

```
type A = {
  [K in keyof Model]: Model[K] extends Function
    ? never
    : K;
}[keyof Model];
```

Jest to dokładnie to samo, co w `FieldsNames`, tylko zamiast `T` podstawiłem nasz `Model`. Idźmy dalej, możemy rozwinąć zapis `keyof Model` do `"name" | "age" | "save"`:

```
type A = {
  [K in | "name" | "age" | "save"]:
    Model[K] extends Function
      ? never
      : K;
}[keyof Model];
```

Następnym krokiem byłoby rozpisanie tych trzech pól osobno, bez sygnatury indeksu. `K` zamieniam na kolejne nazwy:

```
type A = {
  name: Model["name"] extends Function
    ? never
    : "name";
  age: Model["age"] extends Function
    ? never
    : "age";
  doSth: Model["save"] extends Function
    ? never
    : "save";
}[keyof Model];
```

Teraz możemy odczytać typy pól kryjących się pod `Model['name']`, `Model['age']` oraz `Model['save']` i ręcznie wstawić je w odpowiednie miejsca:

```
type A = {
  name: string extends Function
    ? never
    : "name";
  age: number extends Function
    ? never
    : "age";
  save: (() => Promise<void>) extends Function
    ? never
    : "save";
}[keyof Model];
```

Pozostaje nam już tylko odpowiedź na pytanie, czy te typy `extends Function`, czyli czy są one funkcjami? Jeśli tak, to zamiast typu wstawiamy to, co jest po znaku zapytania (czyli `never`), a przeciwnym wypadku to, co po dwukropku:

```
type A = {
  name: "name";
  age: "age";
  save: never;
}[keyof Model];
```

Kolejnym krokiem tutaj jest odczytanie wartości z pól tak powstałego obiektu. Służy temu omawiania w poprzednim rozdziale składnia `obj[keyof obj]`. W rezultacie:

```
type A = "name" | "age" | never;
// Czyli to samo, co 'name' | 'age'
```

Wróćmy do typu `ModelFields` i podstawmy znaleziony przez nas element tej układanki:

```
type ModelFields = {
  [K in "name" | "age"]: Model[K];
};
```

Teraz już z górki. Wiemy, że ten zapis oznacza tak naprawdę stworzenie dwóch pól w obiekcie:

```
type ModelFields = {
  name: Model["name"];
  age: Model["age"];
};
```

Ostatni krok to proste podstawienie:

```
type ModelFields = {
  name: string;
  age: number;
};
```

Wow, to było coś, prawda? Krok po kroku odtworzyliśmy skomplikowaną pracę, którą normalnie wykonuje za nas kompilator TypeScripta. Przejrzyj to dokładnie i powoli, aby lepiej zrozumieć działanie typów warunkowych.

Ten sam efekt można uzyskać korzystając z typów biblioteki standardowej TypeScripta. W tym przypadku `Pick`:

```
type ModelFields = Pick<Model, FieldsNames<Model>>;
```

Więcej o typach wbudowanych w TS porozmawiamy jeszcze w rozdziale 21.5.

13.7 Opóźnione warunki

Czasem w miejscu użycia typu warunkowego jest zbyt mało informacji, aby kompilator był w stanie odpowiedzieć, czy warunek jest spełniony, czy nie. Rezultatem jest wtedy typ warunkowy zamiast

zwykłego. Taką sytuację nazywamy *deferred conditional type*, czyli opóźnionym typem warunkowym. Zakładając, że mamy funkcję generyczną:

```
declare function getEntityID<T>(
  x: T,
): T extends Entity ? string : number;
```

Poniższe użycie sprawi, że otrzymamy stałą mającą typ warunkowy:

```
function getEntityData<U>(x: U) {
  const id = getEntityID(x);
  // U extends Entity ? string : number
}
```

Typ `id` nie może być w tym miejscu jednoznacznie ustalony, gdyż zależy on od `x`, a więc od parametru generycznego `U` i dlatego kompilator jeszcze nie wie, czy będzie to `string`, czy `number`. Czy to oznacza, że nasz `result` ma nieznany typ? Ależ skąd! Jest nim `U extends User ? string : number`. Tylko co z nim możemy zrobić?

13.8 Kompatybilność typów warunkowych

Okazuje się, że typy warunkowe są kompatybilne z unią ich możliwych rezultatów. Jeśli mamy typ `T extends U ? A : B`, to jest on zgodny z `A | B`. Ma to sens, prawda? W końcu taki warunek może zwrócić co najwyżej `A` lub `B`! Wracając do kodu z poprzedniej sekcji:

```
function getEntityData<U>(x: U) {
  const id = getEntityID(x);
  // U extends Entity ? string : number

  const foo: string | number = id; // OK!
}
```

Przypisanie `id` do `foo: string | number` jest bezpieczne i prawidłowe.

13.9 infer

Bardzo długo społeczność TS prosiła o dodanie możliwości pobierania typu zwracanego przez funkcję tylko na podstawie jej definicji. Pojawiało się wiele propozycji realizacji tego trudnego zadania, aż w końcu twórcy TypeScripta zdecydowali się na stworzenie mechanizmu bardziej ogólnego. Dzięki słowu kluczowemu `infer` możemy w pewnym sensie ręcznie sterować inferencją typów i powiedzieć TypeScriptowi coś w stylu „podaj mi typ tego, co jest w tym miejscu, czymkolwiek to jest". To niezwykle potężne narzędzie:

```
type ReturnType<T> = T extends (
  ...args: unknown[]
) => infer R
  ? R
  : never;

type t21 = ReturnType<typeof document.createElement>;
// | HTMLAnchorElement
// | HTMLMenuElement
// | HTMLInputElement
```

Powyżej zdefiniowany `ReturnType<T>` mówi: „Jeśli T jest funkcją, to niech TypeScript przypisze do `R` jej typ zwracany (`infer R`), a całe wyrażenie niech zwraca ten typ. W przeciwnym wypadku niech zwróci `never` ".

Taki ogólny mechanizm pozwala nam inferować w zasadzie dowolne parametry w typach generycznych:

```
type PromiseValue<T> = T extends Promise<infer R>
  ? R
  : never;
```

```
const promise = Promise.resolve(12);
type t22 = PromiseValue<typeof promise>;
// number
```

W tym przypadku stworzyliśmy generyk `PromiseValue`, który jako parametr przyjmuje typ obietnicy i zwraca jej zawartość. Jeśli przekazalibyśmy mu coś, co nie jest Promisem, otrzymalibyśmy `never`.

Niestety, jak już wspominałem w sekcji 12.12, aktualnie nie jest możliwe odczytanie typu z zagnieżdżonych Promisów w ten sposób, ani, w ogólności, rekurencyjne używanie generyków bez specjalnych sztuczek.

13.9.1 Wiele inferencji

Słowo kluczowe `infer` może się w jednym wyrażeniu pojawić wiele razy, na różnych pozycjach. Co ciekawe, może dotyczyć tego samego typu, albo wielu różnych. Zacznijmy od tego drugiego przypadku. Weźmy typ podobny do tego, który znany jest każdej osobie piszącej JavaScript na frontendzie z użyciem React:

```
type Component<Props, State> = {
  props: Props;
  state: State;
};
```

Możemy teraz bez problemu stworzyć typ warunkowy, który wydobędzie `Props` i `State`:

```
type GetStateAndProps<C> = C extends Component<
  infer Props,
  infer State
>
  ? [Props, State]
  : never;
```

```
const c = {
  props: 123,
  state: "aa",
};
type t23 = GetStateAndProps<typeof c2>;
// [number, string]
```

Tutaj mieliśmy do czynienia z inferencją `props` i `state`, a wynikiem jest tupla ich typów. Operatora `infer` można używać również w więcej niż dwóch miejscach, aby wyciągać typy z naprawdę złożonych generyków. Taki kod rzadko pisze się samemu, ale używają go różne biblioteki, aby zapewnić nam większą wygodę i bezpieczeństwo!

13.9.2 Inferencja tego samego typu

Możliwe jest również nakazanie TypeScriptowi wnioskowania tego samego typu w różnych miejscach. Rezultatem będzie unia lub część wspólna, w zależności od pozycji tych typów. Aby ustalić, z którym z tych dwóch przypadków mamy do czynienia, przyda nam się wiedza na temat kowariancji i kontrawariancji z rozdziału 10, gdyż okazuje się, że to właśnie od tego zależy, jaki typ zostanie wywnioskowany! Przywołując fragment kodu napisany wcześniej w rozdziale 10:

```
type Covariant<T> = () => T;
type Contravariant<T> = (x: T) => void;
```

Możemy spróbować inferować typy różnej wariancji. Posłużą nam do tego obiekty z metodami `createUser` i `createModerator` oraz `saveUser` i `saveModerator`:

```
type GetCovariantType<T> = T extends {
  createUser: Covariant<infer R>;
  createModerator: Covariant<infer R>;
}
  ? R
  : never;
```

```
type GetContravariantType<T> = T extends {
  saveUser: Contravariant<infer R>;
  saveModerator: Contravariant<infer R>;
}
  ? R
  : never;
```

Dla przypomnienia, producenci (funkcje tworzące coś) są kowariantni, a konsumenci (funkcje, które coś przyjmują) kontrawariantni:

```
const repository1 = {
  createUser: (): User => ({
    name: "Michał",
    age: 21,
  }),
  createModerator: (): Moderator => ({
    name: "Kasia",
    age: 19,
    channels: [],
  }),
};
type t24 = GetCovariantType<typeof repository1>;
// { name: string; age: number; }
// czyli User | Moderator

const repository2 = {
  saveUser: (x: User) => {},
  saveModerator: (x: Moderator) => {},
};
type t25 = GetContravariantType<typeof repository2>;
// User & Moderator
```

W rezultacie do `t24` przypisane zostanie `User | Moderator`, natomiast do do `t25` – `User & Moderator`. Dzieje się tak właśnie dlatego, że w pierwszym przypadku inferowany typ znalazł się na po-

zycji kowariantnej, a w drugim na kontrawariantnej. Znów, takiego kodu nie pisze się codziennie, ale niemal codziennie się go używa za pośrednictwem różnych bibliotek. Dlatego warto znać ogólny mechanizm i zasady działania operatora `infer` w różnych kontekstach.

13.10 Podsumowanie

Conditional Types dają ogromne możliwości tworzenia bardzo rozbudowanych i skomplikowanych konstrukcji opartych o zaawansowane typy. Są one niezwykle przydatne i w rozdziale 17 pokażę Ci bardzo konkretny i z życia wzięty sposób na dedukowanie typu zmiennej na podstawie przypisanego do niej walidatora. Połączymy świat typów w czasie kompilacji ze sprawdzaniem wartości w trakcie działania aplikacji i uda nam się to bez duplikowania kodu, a to częsta bolączka osób zaczynających pracę z TS! Posłużą nam do tego właśnie typy warunkowe.