

<epam>

Программирование на уровне типов на TypeScript:

выжимаем из компилятора все соки

Юрий Богомолов

ITsubbotnik





ЮРИЙ БОГОМОЛОВ

- Ведущий инженер-программист в EPAM Systems
- Архитектор фронтэнд-решений на проектах крупнейшего финансового брокера в России
- OSS-контрибьютор, ФП-евангелист

Связь

- TG: @ybogomolov
- Email: yuriy.bogomolov@gmail.com

ПРОГРАММИРОВАНИЕ НА УРОВНЕ ТИПОВ

Что такое программирование на уровне типов

- «Компилятор — что-то среднее между строгим родителем и лучшим другом»
- Бизнес-логику можно выражать не только в исполняющемся коде, но и в аннотациях типов
- Чем строже проверки — тем больше боли, но и больше гарантий корректности
- Отлично ложится на парадигму функционального программирования

Для чего это всё?

Для вас:

1. Полезное документирование кода
2. Ограничение использования кода некорректным способом

Для бизнеса:

1. Снижение количества ошибок и стоимости поддержки
2. Доказательство корректности написанных программ

Базовые понятия

Ключевые слова extends и infer:

```
type ExtractP<T> = T extends Promise<infer A> ? A : never;
```

```
type A = ExtractP<Promise<string>>; // => string
```

Очень приближенно можно считать, что:

- extends это ===;
- type A<T> это функция, принимающая на вход аргумент типа T и возвращающая тип;
- infer это способ «заглянуть внутрь» какого-либо обобщенного типа или структуры в целом.

Приёмы

Непрозрачные типы (opaque types) – способ отождествить элементарный тип с уникальным синонимом:

```
namespace Tag {  
    declare const OpaqueTagSymbol: unique symbol;  
    declare class OpaqueTag<S extends symbol> {  
        private [OpaqueTagSymbol]: S;  
    }  
    export type OpaqueType<T, S extends symbol> = T & OpaqueTag<S>;  
}  
  
type Opaque<T, S extends symbol> = Tag.OpaqueType<T, S>;
```

Приёмы: opaque type

```
declare const UUIDTag: unique symbol;
type UUID = Opaque<string, typeof UUIDTag>

const foo = (id: UUID): void => {
    console.log('Got id:', id);
};

foo('42'); // => Type '"42"' is not assignable to type 'UUID'.
foo('42' as UUID); // => Got id: 42
```

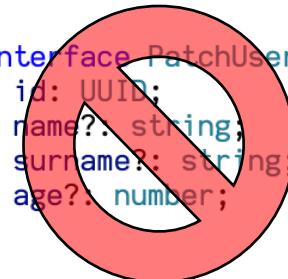
Приёмы

Способ сделать в типе хотя бы одно поле обязательным, а остальные — опциональными:

```
type AtLeastOne<T, Keys extends keyof T = keyof T> =  
Partial<T> & { [K in Keys]: Required<Pick<T, K>> }[Keys];
```

Приёмы: AtLeastOne

```
interface UserModel {  
    id: UUID;  
    name: string;  
    surname: string;  
    age: number;  
}
```



```
interface PatchUserModel {  
    id: UUID;  
    name?: string;  
    surname?: string;  
    age?: number;  
}
```

```
type UpdateUser = Pick<UserModel, 'id'> & AtLeastOne<Omit<UserModel, 'id'>>;  
  
const id = '53bb4453-8457-4098-a4e2-83aa122f7e60' as UUID;  
const updateName: UpdateUser = { id, name: 'John' }; // => Ok  
const updateSurname: UpdateUser = { id, surname: 'Doe' }; // => Ok  
const updateAge: UpdateUser = { id, age: 21 }; // => Ok  
const updateEmpty: UpdateUser = { id }; // => Property 'age' is missing
```

Приёмы

Условный оператор if-then-else, проверка на never и значение по умолчанию:

```
type If<T, U, True, False> = [T] extends [U] ? True : False;
```

```
type IfDef<T, True, False> = If<T, never, False, True>;
```

```
type OrElse<T, Fallback> = IfDef<T, T, Fallback>;
```

Приёмы: If, IfDef,OrElse

```
type Ex1 = IfDef<string & never, 't', 'f'>; // => 'f'  
type Ex2 = IfDef<string | never, 't', 'f'>; // => 't'  
type Ex3 = OrElse<Intersect<{ foo: string }, { bar: number }>, string>;  
// => string  
type Ex4 = OrElse<Intersect<{ foo: string }, { foo: number }>, string>;  
// => { foo: string }
```

Приёмы

Assertion на уровне типов:

```
const assertType = <T>(  
    expect: [T] extends [never] ? never : T  
): T => expect;
```

Приёмы: assertType

```
it('If<T, Eq, True, False>', () => {
  type Assertion1 = If<string, boolean, never, true>;
  expect(assertType<Assertion1>(true)).toBeTruthy();

  type Assertion2 = If<string, string, true, never>;
  expect(assertType<Assertion2>(true)).toBeTruthy();
});

it('IfDef<T, True, False>', () => {
  type Assertion1 = IfDef<string, true, never>;
  expect(assertType<Assertion1>(true)).toBeTruthy();

  type Assertion2 = IfDef<string | never, true, never>;
  expect(assertType<Assertion2>(true)).toBeTruthy();

  type Assertion3 = IfDef<string & never, never, true>;
  expect(assertType<Assertion3>(true)).toBeTruthy();
});
```

Приёмы

Пересечение двух типов:

```
type Intersect<A extends {}, B extends {}> =  
  Pick<A, Exclude<keyof A, Exclude<keyof A, keyof B>>>  
    extends { [x: string]: never } ?  
  never :  
  Pick<A, Exclude<keyof A, Exclude<keyof A, keyof B>>>;
```

Приёмы: Intersect

Выведение типа never в случае, если аргумент функции не является нужным типом:

```
const neverIntersect = <
  A extends {},
  B extends {},
  NeverIntersect extends IfDef<Intersect<A, B>, never, {}> =
    IfDef<Intersect<A, B>, never, {}>
>(
  a: A & NeverIntersect,
  b: B & NeverIntersect,
): A & B & NeverIntersect => ({ ...a, ...b });
```

Приёмы: Intersect

Выведение типа never в случае, если аргумент функции не является нужным типом:

```
interface A { foo: string; }
interface B { bar: number; }

const a1: A = { foo: 'foo' };
const b1: B = { bar: 42 };

const c1 = neverIntersect<A, B>(a1, b1); // => A & B
const c2 = neverIntersect<A, A>(a1, a1);
// => Argument of type 'A' is not assignable
//      to parameter of type 'never'
```

Пример: модульная система для фронта

```
const register = <
  OwnStore extends {},
  ReqStore extends {},
  Routes extends Route = never,
  Permission extends string = never,
  Extra = {},
  ComponentTypes extends {} = never,
  // Typelevel laws:
  StoresNeverIntersect = IfDef<Intersect<OwnStore, ReqStore>, never, {}>
>(component: Module<OwnStore, ReqStore, Routes, Permission, Extra, ComponentTypes>) =>
  <S extends {}, R extends Route, P extends string, E extends {}, C extends {}>(
    registry: StoresNeverIntersect & Registry<S, R, P, E, C>
  ): Either<Error[], Registry<S, R, P, E, C>> =>
  registry.modules[component.name] != null ?
    left([new Error(`Module ${component.name} already registered`)])
  : right({
    modules: { ...registry.modules, [component.name]: component }
  } as Registry<S, R, P, E, C>);
```

Типы высших порядков (Higher-Kinded Types)

Проблема: в TypeScript нельзя выразить что-то подобное:

```
interface Functor<F> {
    // Ошибка TS2315: Type 'F' is not generic.
    map: <A, B>(f: (a: A) => B) => (fa: F<A>) => F<B>;
}
```

Типы высших порядков (Higher-Kinded Types)

При помощи идеи из статьи

[Lightweight higher-kind polymorphism](#)

возможно выражать типы высших порядков, ставя в соответствие типу $A < B >$ тип $\text{Kind}'A', B$.

```
const URI = 'Result';
type URI = typeof URI;

declare module 'fp-ts/lib/HKT' {
    interface URItoKind2<E, A> {
        Result: Result<E, A>;
    }
}

// Result container:
type Failure<E> = { tag: 'Failure', error: E };
type Success<A> = { tag: 'Success'; value: A };
type Result<E, A> = Failure<E> | Success<A>;
```

Типы высших порядков (Higher-Kinded Types)

Теперь стало возможным выражать экземпляры тайп-классов вроде Functor, Applicative или Monad для нашего контейнера!

Такой подход активно используется в библиотеке fp-ts и ее экосистеме.

```
// Functor
export const map = <E, A, B>(
  fa: Result<E, A>,
  f: (a: A) => B,
): Result<E, B> => {
  switch (fa.tag) {
    case 'Failure':
      return fa;
    case 'Success':
      return success(f(fa.value));
  }
};

export const result: Functor2<URI> = {
  URI,
  map,
};
```

ВЫЖИМАЕМ ИЗ КОМПИЛЯТОРА ВСЕ СОКИ

Булев тип и проверка условий

Используя литералы, мы можем выразить булев тип, который станет краеугольным камнем в операциях с типами.

Синтаксис доступа к полю типа через индексатор станет способом делать проверки условий.

```
type False = 'f';
type True = 't';
type Bool = False | True;
```

```
type Foo = {
    t: number;
    f: boolean;
}[True]; // => number
```

Булева логика

Мы можем определить ряд вспомогательных функций для операций над булевыми значениями:

```
type If<Cond extends Bool, Then, Else> = {  
    f: Else;  
    t: Then;  
}[Cond];  
  
type Not<Cond extends Bool> = If<Cond, False, True>;  
type And<Cond1 extends Bool, Cond2 extends Bool> = If<Cond1, Cond2, False>;  
type Or<Cond1 extends Bool, Cond2 extends Bool> = If<Cond1, True, Cond2>;  
type BoolEq<Cond1 extends Bool, Cond2 extends Bool> = If<Cond1, Cond2, Not<Cond2>>;
```

Что такое число?

Аксиомы Пеано*:

1. 0 является натуральным числом.
2. Число, следующее за натуральным, тоже является натуральным.
3. 0 не следует ни за каким натуральным числом.

*: в оригинале первым натуральным числом была единица, а не ноль

```
type Zero = { isZero: True };
type Nat = Zero | { isZero: False, prev: Nat };

type Succ<N extends Nat> = { isZero: False, prev: N };
type Prev<N extends Succ<Nat>> = N['prev'];
type IsZero<N extends Nat> = N['isZero'];

type _0 = Zero;
type _1 = Succ<_0>;
type _2 = Succ<_1>;
type _3 = Succ<_2>;
```

Базовая арифметика

Сложение $A + B$:

Если B — ноль, то результат равен A .

Иначе — следующему числу за результатом сложения A и $B-1$.

Умножение $A \times B$:

Если B — ноль, то результат равен нулю.

Иначе — сложению A и результата умножения $A \times (B-1)$.

```
type Add<A extends Nat, B extends Nat> = {  
    f: B extends Succ<Nat> ? Succ<Add<A, Prev<B>>> : never;  
    t: A;  
}[IsZero<B>];
```

```
type Mul<A extends Nat, B extends Nat> = {  
    f: B extends Succ<Nat> ?  
        Mul<A, Prev<B>> extends infer R ?  
            Add<A, R extends Nat ? R : never> : never :  
            never;  
    t: Zero;  
}[IsZero<B>];
```

Базовая арифметика

```
type _5 = Add<_2, _3>;
// => { isZero: "f"; prev: Succ<Succ<Succ<Succ<Zero>>> }
```

```
type _6 = Mul<_2, _3>;
// => { isZero: "f"; prev: Succ<Succ<Succ<Succ<Succ<Zero>>>> }
```

```
type _9 = Mul<_3, _3>;
// { isZero: "f"; prev: Succ<Succ<Succ<Succ<Succ<Succ<Succ<Zero>>>>>> }
```

```
type _7 = Add<_1, Mul<_2, _3>>;
// => { isZero: "f"; prev: Succ<Succ<Succ<Succ<Succ<Succ<Zero>>>>> }
```

Базовая арифметика

Операция \leq — если первый operand равен нулю, а второй нет, то первый operand меньше второго; в противном случае вычитаем по единице из обоих и рекурсивно продолжаем:

```
type Lteq<M extends Nat, N extends Nat> = {
    f: If<
        IsZero<N>,
        False,
        Lteq<M extends Succ<Nat> ? Prev<M> : never, N extends Succ<Nat> ? Prev<N> : never>
    >;
    t: IsZero<N>;
}[IsZero<M>];
```

Базовая арифметика

Два числа равны, если первое меньше или равно второму, а второе меньше или равно первому:

```
type NatEq<M extends Nat, N extends Nat> =  
    Lteq<M, N> extends infer A ?  
    Lteq<N, M> extends infer B ?  
    And<A extends Bool ? A : never, B extends Bool ? B : never> : never : never;
```

Числа Фибоначчи

Знакомая со школы формула:

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$$

```
type Fib<N extends Nat> = {
    f: If<
        NatEq<-1, N>,
        -1,
        Add<
            Fib<N extends Succ<Nat> ? Prev<N> : never>,
            Fib<N extends Succ<Succ<Nat>> ? Prev<Prev<N>> : never>
        >
    >;
    t: Zero;
}[IsZero<N>];
```

Числа Фибоначчи

Ряд Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21...

```
declare const fib7: Fib<_7>;
// => Succ<Succ<Succ<Succ<Succ<Succ<Succ<Succ<Succ<Succ<Zero>>>>>>>>
// => 13
```

Факториал

Формула:

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

```
type Fact<N extends Nat> = {
    f: Mul<N, Fact<N extends Succ<Nat> ? Prev<N> : never>>;
    t: _1;
}[Or<IsZero<N>, NatEq<_1, N>>];
```

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

```
declare let fact4: Fact<_4>; // 24 times Succ<...Succ<Zero>>...
```

Односвязные списки

Та же идея, что и с числами Пеано:
список либо пустой,
либо состоит из головы и хвоста.

Список можно составить из головы и хвоста.

Для непустого списка можно определить операции взятия головы и хвоста.

```
type HNil = { isNil: True; };
type HCons = { isNil: False; head: any; tail: HList; };
type HList = HNil | HCons;

type Cons<Head, Tail extends HList> =
  { isNil: False, head: Head, tail: Tail };
type Head<Xs extends HCons> = Xs['head'];
type Tail<Xs extends HCons> = Xs['tail'];
type IsNil<Xs extends HList> = Xs['isNil'];
```

Обращение списка

Чтобы развернуть список, мы берем его голову и добавляем к обращенному хвосту.

```
type Reverse<Xs extends HList> = Rev<Xs, HNil>;
type Rev<Xs extends HList, T extends HList> = {
    f: Xs extends HCons ? Rev<Tail<Xs>, Cons<Head<Xs>, T> : HNil;
    t: T;
}[IsNil<Xs>];

type ABC1 = Cons<'a', Cons<'b', Cons<'c', Cons<1, HNil>>>>;
type _1CBA = Reverse<ABC1>;
// => HCons<1, HCons<'c', HCons<'b', HCons<'a', HNil>>>>
```

Длина списка

Если список пуст, его длина равна нулю.

Если список содержит голову и хвост, то его длина равна $1 + (\text{длина хвоста})$.

```
type Length<Xs extends HList> = {
    f: Xs extends HCons ? Succ<Length<Tail<Xs>>> : Zero;
    t: Zero;
}[IsNil<Xs>];

type LengthOfABC1 = Length<ABC1>; // => 4
```

МАТЕМАТИЧЕСКАЯ ЛОГИКА И ТЕОРИЯ ТИПОВ

Соответствие Карри-Ховарда

Конструкции интуиционистской логики	Конструкции типизированного λ-исчисления	Типы TypeScript
Истинное высказывание	Тип T	null и undefined
Ложное высказывание	Тип \perp	never
Дизъюнкция $A \vee B$	Тип-сумма	$A \mid B$
Конъюнкция $A \wedge B$	Тип-произведение	$A \& B$
Импликация $A \rightarrow B$	Тип функции	$(a: A) \Rightarrow B$
Квантор всеобщности \forall	Тип зависимого произведения Π	⚠
Квантор существования \exists	Тип зависимой суммы Σ	⚠
Modus ponens (правило вывода)	Применение функции	$f(a)$
Введение импликации	Абстрагирование по переменной	$(a: A) \Rightarrow \dots$
Доказательство высказывания	Построение терма для типа	<code>const a: A = ...</code>

Соответствие Карри-Ховарда

// Гипотетический силлогизм:

// $((p \rightarrow q) \wedge (q \rightarrow r)) \vdash p \rightarrow r$

```
type HSProof = <P, Q, R>(pq: (p: P) => Q, qr: (q: Q) => R) => (p: P) => R;  
const proofHS: HSProof = (pq, qr) => (p) => qr(pq(p));  
const proofHS: HSProof = (pq, qr) => compose(qr, pq);
```

// Применяем modus ponens:

```
const proofHS1 = proofHS<number[], number, string>(  
  (list) => list.length,  
  (len) => len.toString(),  
)([1, 2, 3]); // => "3"
```

Соответствие Карри-Ховарда

```
// Конструктивная дилемма
// ((p → q) ∧ (r → s) ∧ (p ∨ r)) ⊢ q ∨ s
```

```
type CDPProof<P, Q, R, S> = (
    pq: (p: P) => Q,
    rs: (r: R) => S,
    p_OR_r: P | R,
) => Q | S;
```

```
const proofCD = <P, Q, R, S>(
    isP: (p: P | R) => p is P
): CDPProof<P, Q, R, S> =>
    (pq, rs, p_OR_r) => isP(p_OR_r) ? pq(p_OR_r) : rs(p_OR_r);
```

Соответствие Карри-Ховарда

```
// Конструктивная дилемма: пример
const proof1 = proofCD<number, string, boolean, number>(
  (p): p is number => typeof p === 'number',
)((p) => p.toString(), (r) => r ? 1 : 0, 10); // => "10"

const proof2 = proofCD<number, string, boolean, number>(
  (p): p is number => typeof p === 'number',
)((p) => p.toString(), (r) => r ? 1 : 0, false); // => 0
```

Что дальше?

- Зависимые типы (dependent types):
 - Idris, Agda, Coq, DJS 😊
- Линейные и/или аффинные типы (linear types, affine types):
 - Rust, F*, Idris, Haskell (до какой-то степени)
- Множественные типы (quantitative types):
 - Idris 2 (WIP), Pikelet

КАК ЖИТЬ ДАЛЬШЕ?

Что вы можете сделать уже сейчас?

ЗАСТАВЬТЕ КОМПИЛЯТОР РАБОТАТЬ НА ВАС

1. Включите максимально строгий режим компилятора:

`strict: true,`

`strictNullChecks: true,`

`noImplicitAny: true,`

`strictFunctionTypes: true,`

`strictPropertyInitialization: true`

2. Начните использовать **Type-Driven Development** для новых задач

3. Заставляйте некорректные бизнес-кейсы выводиться в параметры типа **never**

4. Используйте алгебраические типы данных и системы эффектов

[github://YBogomolov/talk-typelevel-ts](https://github.com/YBogomolov/talk-typelevel-ts)

Q&A