Houzair Koussa

Software engineer @ Stratalis
Maintainer @ thephilosophicalcode.com

Union & Intersection Types In TypeScript

A Deep Dive

Prerequisites

1. Basic knowledge of TypeScript
2. Have used, or at least familiar with, the union and intersection types

Agenda

1. Elixir and set-theoretic types

Union and intersection types:

2. TypeScript vs. set-theoretic types
3. What they are - structural types
4. How they work - source code of TypeScript’s compiler

Elixir and set-theoretic types

Elixir

Functional programming language

Designed by José Valim

Released in 2012

Runs on top of Erlang's BEAM virtual machine

Currently - dynamic and strong type system

Shifting towards - *static* and strong *set-theoretic* type system

Elixir and set-theoretic types

Set-theoretic types - core ideas

1. A type is a set of values

    The type string is the set of all values that are string-like

2. Apply set-theoretic concepts and operations on types

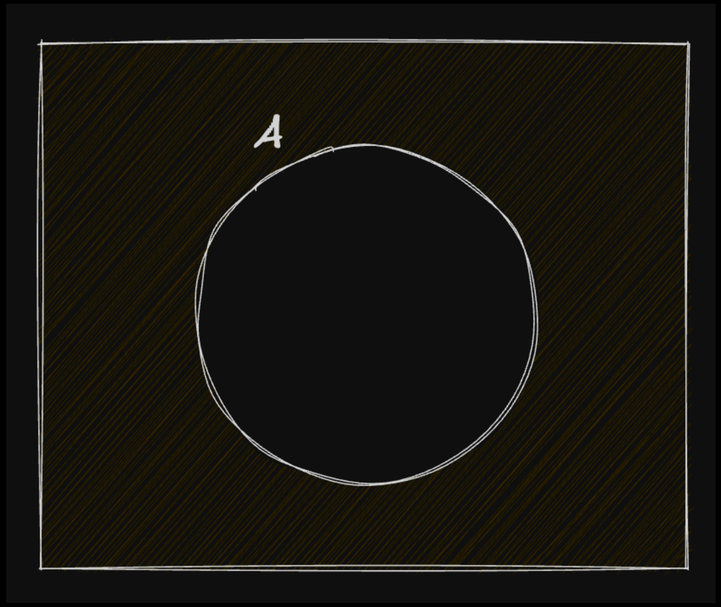
    The empty set is like the never type

    Type A = Type B iff A and B share exactly the same values

    For any type A and B - not(A), A ∪ B, A ∩ B …

Elixir and set-theoretic types

Set-theoretic types - not(A)

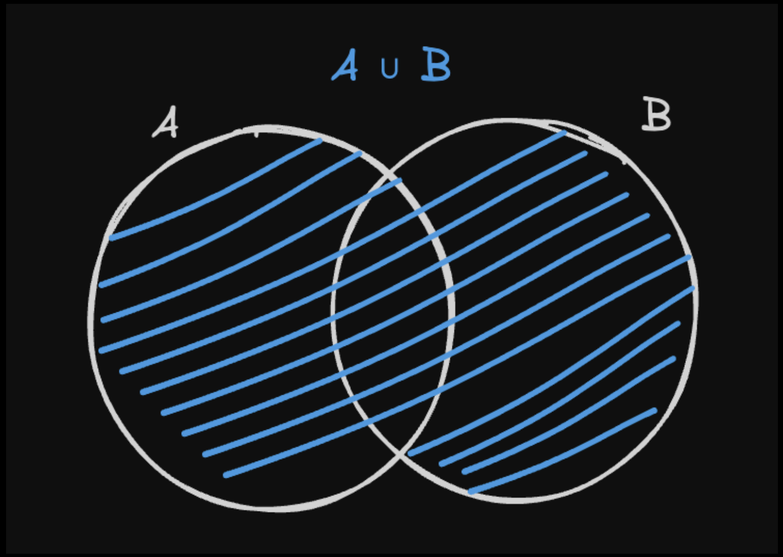


not(A) is the set of all values x such that $x \notin A$

So, not(string) is the set of all values that are not string-like

Elixir and set-theoretic types

Set-theoretic types - A ∪ B

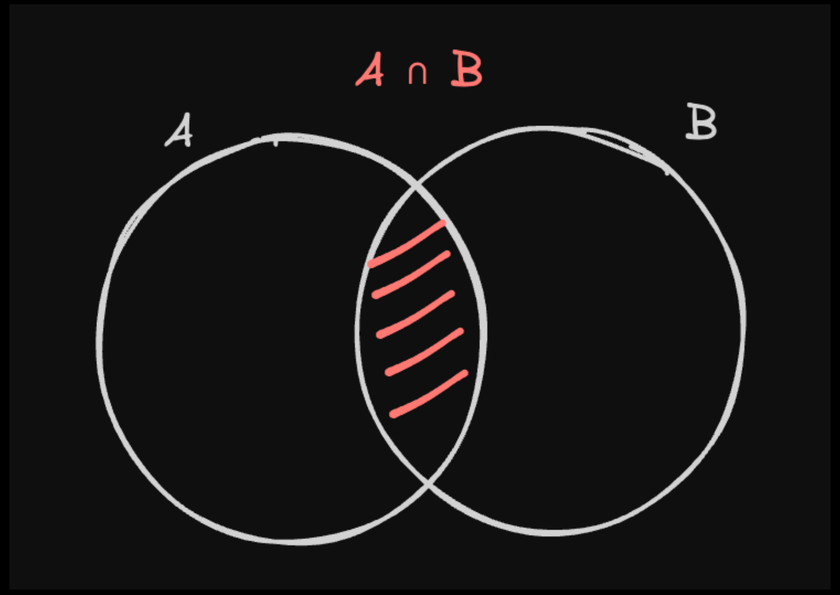


A ∪ B is the set of all values x such that x ∈ A **or** x ∈ B

So, ‘string ∪ boolean’ is the set of true, false and all strings

Elixir and set-theoretic types

Set-theoretic types - A ∩ B



A ∩ B is the set of values x such that x ∈ A **and** x ∈ B

So, ‘Nullable ∩ null’ is the set containing null

Elixir and set-theoretic types

Use case - function overloading

```elixir
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

negate(x) such that:

1. If x is an integer, then negate(x) is -x

    negate(418) = -418

2. If x is a boolean, then negate(x) is not x

    negate(true) = false

Elixir and set-theoretic types

Use case - function overloading

Intuitive but over-generating type:

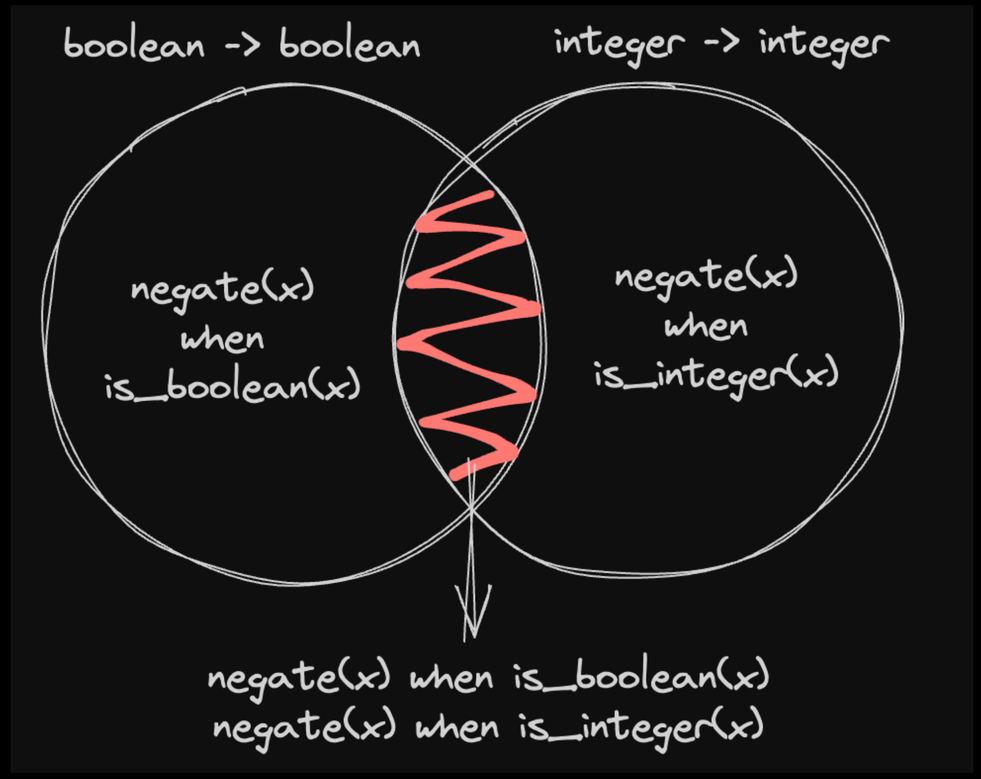
negate: (x: boolean | integer) => boolean | integer

Type of x	Expected return type	Actual return type	Over-generated return type
boolean	boolean	boolean | integer	integer
integer	integer	boolean | integer	boolean

Elixir and set-theoretic types

Use case - function overloading

((x: integer) => integer) ∩ ((x: boolean) => boolean)



Union and intersection

Introspection

Set-theoretic types are quite intuitive.

But, how do they compare to TypeScript’s union and intersection types?

TypeScript vs. set-theoretic types

Refresher - union (A | B)

Core idea - A | B is a type that is like A **or** like B **or** both (if possible)

Note - A | B is the union-intersection of A and B

```
type A = { foo: string; }

type B = { bar: string; }

type U =  A | B;

const likeA: U = { foo: "foo" }
const likeB: U = { bar: "bar" }
const likeAB: U = { foo: "foo", bar: "bar" }
```

TypeScript vs. set-theoretic types

Refresher - intersection (A & B)

Core idea - A & B is a type that is both like A **and** B

```ts
type A = { foo: string; }

type B = { bar: string; }

type I =  A & B;

const likeAB: I = { foo: "foo", bar: "bar" }

const likeAOnly: I = { foo: "foo" }

const likeBOnly: I = { bar: "bar" }
```

TypeScript vs. set-theoretic types

Heads up

The upcoming snippets will feel familiar and not weird at all.

Despite this, let’s remain skeptical

TypeScript vs. set-theoretic types

```ts
type A = { foo: string; }

type B = { bar: string; }

type U =  A | B;

const o: U = {foo: "foo", bar: "bar"}

const o_: A = {foo: "foo", bar: "bar"}

const o__: B = {foo: "foo", bar: "bar"}
```

`{foo: “foo”, bar: “bar”}` is of type A | B but not of type A nor B

TypeScript vs. set-theoretic types

TypeScript's union is not set-theoretic

Recall - A ∪ B is the set of all values x such that x ∈ A **or** x ∈ B

So, if something is neither in A nor in B, then that thing cannot be in A ∪ B

Yet, `{foo: "foo", bar: "bar"}` is in "A ∪ B" despite being neither in A nor B

TypeScript vs. set-theoretic types

```ts
type A = { foo: string; }

type B = { bar: string; }

type I =  A & B;

const o: I = {foo: "foo", bar: "bar"}

const o_: A = {foo: "foo", bar: "bar"}

const o__: B = {foo: "foo", bar: "bar"}
```

`{foo: “foo”, bar: “bar”}` is of type A & B but not of type A nor B

TypeScript vs. set-theoretic types

TypeScript's intersection is not set-theoretic

Recall - A ∩ B is the set of values x such that x ∈ A **and** x ∈ B

So, if something is neither in A nor in B, then that thing cannot be in A ∩ B

Yet, `{foo: "foo", bar: "bar"}` is in "A ∩ B" despite being neither in A nor B

Union and intersection

what they are / how they work

We know about set-theoretic types

and that TypeScript’s union and intersection types are not set-theoretic

Naturally, we wanna understand:

1. What they are - structural types
2. How they work - source code of TypeScript’s compiler

Structural types

TypeScript has a structural type system

Core idea - a type is a declaration about the properties of a particular structure

Intuitively, a type is like a description of the fields and methods of an object

For example, the type ‘string’ describes objects/structures that have fields and methods usually associated with strings - like ‘length’ or ‘split()’

Structural types

Two types are equal iff they share the same structure

Crudely, they have the same methods and fields

In other words, they extend each other!

```
type IsEqual<T, U, Y=true, N=false> =
  (<G>() => G extends T ? 1 : 2) extends
  (<G>() => G extends U ? 1 : 2) ? Y : N;

isEqual<any, any> // OK
isEqual<string, number> // NOT OK
```

Structural types

What about { foo: “foo”, bar: “bar” } ?

type A = {foo: string;} describes an object that only contains the field “foo”

“foo” describes an object that only contains string methods and fields

Ditto for type B = {bar: string;}

{ foo: “foo”, bar: “bar” } is neither a structure only containing the field “foo” nor the field “bar”

So, it is neither of type A nor B

Structural types

What about { foo: “foo”, bar: “bar” } and union?

Recall - A | B is a type that is like A **or** like B **or** both (if possible)

So, A | B describes a structure that contains

1. Only the fields and methods of A or
2. Only the fields and methods of B or
3. Only the fields and methods of A and B (if possible)

In other words, A | B cares about partial structural likeliness

{ foo: “foo”, bar: “bar” } partially (and fully) contains the fields of A and B

Structural types

What about { foo: “foo”, bar: “bar” } and intersection?

Recall - A & B is a type that is both like A **and** B

So, A & B describes a structure that only contains the fields and methods of A and B

In other words, A & B cares about full structural likeliness

{ foo: “foo”, bar: “bar” } fully contains the fields of A and B

Union and intersection

What they are - in summary

Union and intersection are structural declarations

Union cares about partial structural likeliness

Intersection cares about full structural likeliness

Unlike set-theoretic union and intersection, an object may not be of type A nor B but of type ‘A | B’ or ‘A & B’

Union and intersection

How they work

We wanna witness the workings of:

1. Union and intersection are structural declarations

2. Union cares about partial structural likeliness

3. Intersection cares about full structural likeliness

Source code of TypeScript's compiler

Overview

Open source at https://github.com/microsoft/TypeScript

TypeScript's compiler is written in TypeScript

Focus:

1. src/compiler/types.ts
2. src/compiler/checker.ts

Source code of TypeScript's compiler

Union and intersection are structural declarations

1. Interface Type - a type is just an object/structure!

```
export interface Type {
    flags: TypeFlags;
    id: TypeId;
    checker: TypeChecker;
    ...
}
```

Source code of TypeScript's compiler

Union and intersection are structural declarations

2. UnionOrIntersectionType - an object that extends the Type object. It contains the types being unionised or intersected.

```
export interface UnionOrIntersectionType extends Type {
    types: Type[]; // Constituent types
    ...
}
```

3. Union - an object that extends the UnionOrIntersectionType object

4. Intersection - an object that extends the UnionOrIntersectionType object

Source code of TypeScript's compiler

Union cares about partial structural likeliness

We wanna type check an object (source) against some union type (target)

So, we'll check if *the source object is partially like the target object*

1. The target is optimally created as a union type object containing its constituent types and upserted to a "global" map of union type objects:

   getUnionType

   getUnionTypeWorker

   addTypesToUnion

   addTypeToUnion

Source code of TypeScript's compiler

Union cares about partial structural likeliness

2. We check if *at least one* of the target's constituents object type is related to our source object:

checkTypeRelatedTo

isRelatedTo

unionOrIntersectionRelatedTo

typeRelatedToSomeType

Source code of TypeScript's compiler

Intersection cares about full structural likeliness

We wanna type check an object (source) against some intersection type (target)

So, we'll check if *the source object is fully like the target object*

1. The target is optimally created as an intersection type object containing its constituent types and upserted to a "global" map of intersection type objects:

   getIntersectionType

   createIntersectionType

   addTypesToIntersection

   addTypeToIntersection

Source code of TypeScript's compiler

Intersection cares about full structural likeliness

2. We check if *all* of the target's constituents object type are related to our source object:

checkTypeRelatedTo

isRelatedTo

unionOrIntersectionRelatedTo

typeRelatedToEachType

Union and intersection

How they work - in summary

Union and intersection are structural declarations - they are objects that extend the base Type object

Union cares about partial structural likeliness - check if a source object is related to *at least one* of a union's constituents

Intersection cares about full structural likeliness - check if a source object is related to *all* of an intersection's constituents

Thank you!

Slides @ github.com/houzyk/talks

Notes

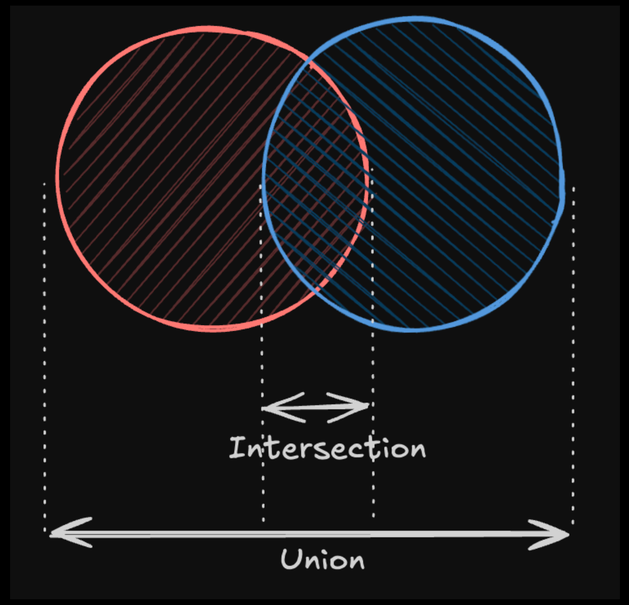
1. Set-theoretic union vs. overloaded negate
2. Union type - a note from the docs
3. Recursive types
4. Structural union and intersection as set-theoretic (conjecture)
5. TS/JS - Compile time vs. run time

Notes

Set-theoretic union vs. overloaded negate

((x: integer) => integer) ∪ ((x: boolean) => boolean) over-generates:

1. Any function that meets any one side of the union criterion satisfies the type
2. negate() meets exactly both sides; not just one



Notes

Union type - a note from the docs

“It might be confusing that a *union* of types appears to have the *intersection* of those types’ properties […]

the name *union* comes from type theory. […]

Notice that given two sets with corresponding facts about each set, only the *intersection* of those facts applies to the *union* […]

If we had a room of tall people wearing hats, and another room of Spanish speakers wearing hats, after combining those rooms, the only thing we know about *every* person is that they must be wearing a hat.”

Notes

Recursive types

```
type List<R> = R & { next: List<R> | null }

const l: List<{foo: string}> = {
    foo: "1",
    next: {
        foo: "2",
        next: {
            foo: "3",
            next: {
                foo: "4",
                next: null
            }
        }
    }
}
```

Notes

Structural union and intersection as set-theoretic (conjecture)

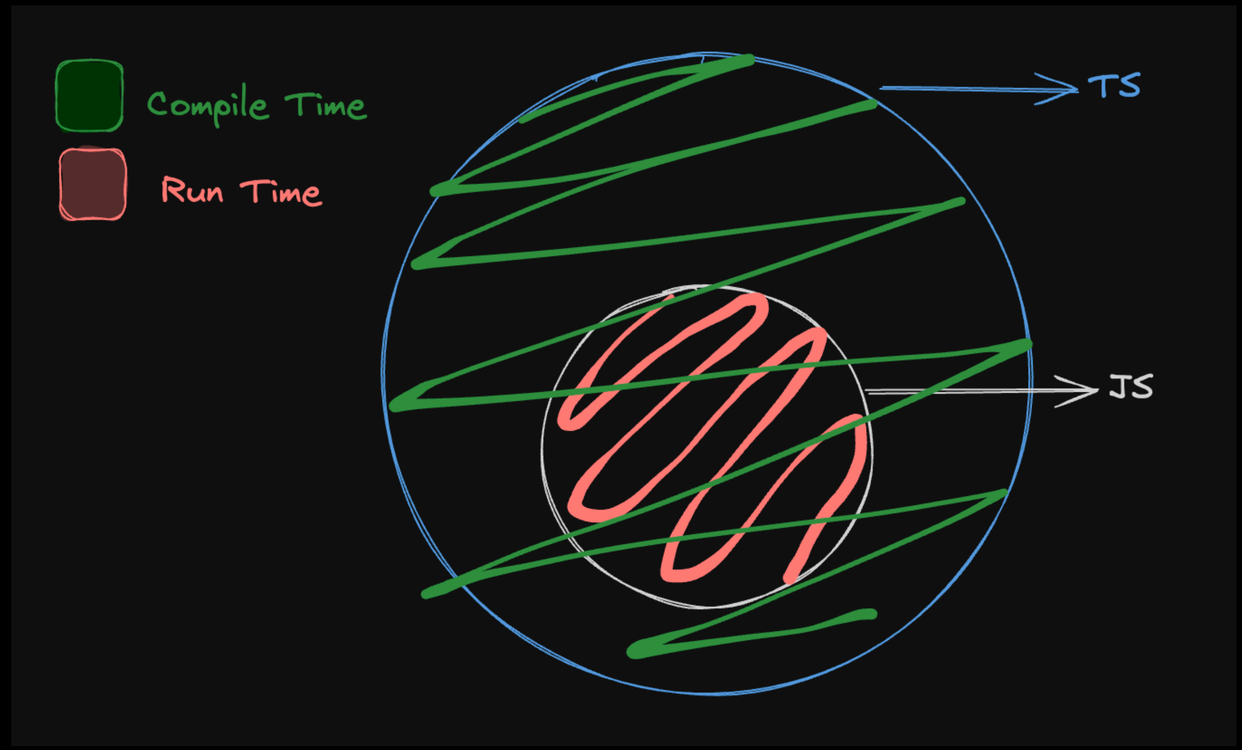
Object - a set of ordered pairs (k,v) where k is unique

A | B - set of all objects x such that for all keys k of x and value v of k, (k,v) is an element of A or B

A & B - set of all objects x such that for all key-value pairs (kA,vA) of A there is a key-value (kX,vX) of x such that (kA,vA) = (kx,vX) and for all key-value pairs (kB, vB) of B, there is a key-value (kX,vX) of x such that (kB,vB) = (kx,vX)

Notes

TS/JS - Compile time vs. run time



Typically, TS-only code is executed at compile time but JS code like runtime type checks (“typeof”) is also executed at compile time