Inferencia en TypeScript

Typescrip
t

Refinamiento

```ts
type Unit = 'cm' | 'px' | '%';
const units: Unit[] = ['cm', 'px', '%'];

// Check if a value is a valid unit
function parseUnit(value: string): Unit | null {
    for (const unit of units) {
        if (value.endsWith(unit)) {
            return unit;
        }
    }
    return null;
}
```

Typescript

Refinamiento

```
function parseWidth(width: number | string | null | undefined): Width | null {
    if (width == null) {
        return null;
    }
    // width ya no puede ser null ni undefined
                width: number | string | null | undefined
                (parameter) width: string | number
    if (typeof width === 'number') {
        return {
            unit: 'px',
            value: width
        };
    }

    const unit = parseUnit(width);
    if (unit) {
        return {
            unit,
            value: parseFloat(width)
        };
    }

    return null;
}
```

Typescript

Refinamiento

```
function parseWidth(width: number | string | null | undefined): Width | null {
    if (width == null) {
        return null;
    }
    // width ya no puede ser null ni undefined

    if (typeof width === 'number') {
        return {
            unit: 'px',
            value: width
        };
    }
    // width ya solo puede ser string

                                (parameter) width: string
    const unit = parseUnit(width);
    if (unit) {
        return {
            unit,
            value: parseFloat(width)
        };
    }

    return null;
}
```

Typescript

Refinamiento

```
function parseWidth(width: number | string | null | undefined): Width | null {
    if (width == null) {
        return null;
    }
    // width ya no puede ser null ni undefined

    if (typeof width === 'number') {
        return {
            unit: 'px',
            value: width
        };
    }
    // width ya solo puede ser string

    const unit = parseUnit(width);
    if (unit) {
        retu (property) unit: Unit
            unit,
            value: parseFloat(width)
        };
    }

    return null;
}
```

Typescript

Refinamiento

```ts
type UserTextEvevt = {value: string};
type UserMouseEvent = {value: [number, number]};

type UserEvent = UserTextEvevt | UserMouseEvent;

function handleUserEvent(event: UserEvent): void {
    if(typeof event.value === 'string') {
        // handle text event
        console.log('Text:', event.value.toLocaleUpperCase());
    }
    else {
        // handle mouse event
        console.log('Mouse Psoition:', `[${event.value[0]}, ${event.value[1]}]`);}
}
```

Typescript

Refinamiento

```ts
type UserTextEvevt = {
    type: 'TextEvent';
    text: string;
    target: HTMLInputElement;
};
type UserMouseEvent = {
    type: 'MouseEvent';
    position: [number, number];
    target: HTMLElement;
};

type UserEvent = UserTextEvevt | UserMouseEvent;
```

```ts
function handleUserEvent(event: UserEvent): void {
    if (event.type === 'TextEvent') {
        // handle text event
        // TS puede asegurar el tipo de target gracias a la TAG 'type' con valor 'TextEvent'
        console.log('Text:', event.text.toLocaleUpperCase(), event.target);
    } else {
        // handle mouse event
        console.log(
            'Mouse Psoition:',
            `[${event.position[0]}, ${event.position[1]}]`,
            event.target
        );
    }
}
```

Typescript

Refinamiento

```
type Day = Weekday | 'Sabado' | 'Domingo';
type Day = Weekday | "Sabado" | "Domingo"
Function lacks ending return statement and return type does not include 'undefined'. ts(2366)
View Problem  No quick fixes available

type Weekday = 'Lunes' | 'Martes' | 'Mie
type Day = Weekday | 'Sabado' | 'Domingo

export function getNextDay(w: Weekday): Day {
    switch (w) {
        case 'Lunes':
            return 'Martes';
        case 'Martes':
            return 'Miercoles';
        case 'Miercoles':
            return 'Jueves';
        case 'Jueves':
            return 'Viernes';
        // case 'Viernes':
        //     return 'Sabado';
    }
}
```

Typescript

Refinamiento de Arreglos. Uplas

```ts
const upla = [1, true]; //  (number | boolean)[]

function tuple<T extends unknown[]>(...args: T) {
    return args;
}

const result = tuple( args: 1,  args: true); // [number, boolean]

function makeCoords(x: number, y: number) {
    return [x, y];
}

function makeUplaCoords(x: number, y: number): [number, number] {
    return [x, y];
}

function makeUplaCoordsWithTuple(x: number, y: number) {
    return tuple(x, y);
}

function makeUplaCoords2(x: number, y: number) {
    return [x, y] as const;
}
```

```ts
const c1 = makeCoords( x: 1,  y: 2); // number[]
const c2 = makeUplaCoords( x: 1,  y: 2); // [number, number]
const c3 = makeUplaCoords2( x: 1,  y: 2); // readonly [number, number]
const c4 = makeUplaCoordsWithTuple( x: 1,  y: 2); // [number, number]
```

Typescript

Comprobaciones de tipo. Type guards

```ts
function isString(value: unknown): value is string {
    return typeof value === 'string';
}

function parseInput(value: string | number) {
    if (isString(value)) {
        console.log('String', value.toUpperCase());
    } else {
        console.log('Number', value);
    }
}

parseInput('Hola');
parseInput(10);
```

```ts
type Coord3D = [number, number, number];
type Coord2D = [number, number];
function isCoord3D(value: Coord3D | Coord2D): value is Coord3D {
    return value.length === 3;
}

function printCoord(coord: Coord3D | Coord2D) {
    if (isCoord3D(coord)) {
        console.log(`3D [${coord[0]}, ${coord[1]}, ${coord[2]}]`);
    } else {
        console.log(`2D [${coord[0]}, ${coord[1]}]`);
    }
}

printCoord([1, 2, 3]);
printCoord([1, 2]);
```

Typescript

Aserciones

```ts
function formatInput(value: string) {
    return value.toUpperCase();
}
function getUserInput(): string | number {
    throw new Error('Not implemented');
}
const input = getUserInput(); // string | number
// Forzamos a TS a creer que input es de tipo string
const textFormated = formatInput(input as string);
```

```ts
const user = {id: 1
const customType = user as string;
```

Conversion of type '{ id: number; name: string; }' to type 'string' may be a mistake because neither type sufficiently overlaps with the other.

View Problem Quick Fix... (⌘.)

```ts
const user = {id: 1, name: 'Juan'};
// const customType = user as string;
const customType = user as unknown as string;
```

Typescript

Aserciones

```ts
////////////////////////////////////////////////////////////////////////////
// Aserciones no nulas
////////////////////////////////////////////////////////////////////////////
type Dialog = { id?: string };
function closeDialog(dialog: Dialog) {
    if (!dialog.id) {
        return;
    }
    setTimeout(() => {
        // Podemos asumir que Dialog tiene id, (TS solo sabe que un Dialog puede o no tener id)
        // Asumimos que como Dialog tiene id existe un elemento en el DOM con ese id. (TS solo sabe que queryElementById podria devolver null)
        removeFromDOM(dialog, document.getElementById(dialog.id!)!);
    });
}
function removeFromDOM(dialog: Dialog, element: Element) {
    // parentNode puede ser null, pero como es llamada desde closeDialog
    // asumimos ques el dialogo tiene id y por tanto existe y tiene un padre
    element.parentNode!.removeChild(element);
    delete dialog.id;
}

closeDialog({ id: 'dialog-1' });
```

Typescript

Extracción de tipos

```ts
type APIResponse = {
    user: {
        email: string;
        name: string;
        followers: {
            totalCount: number;
            list: [{ email: string; name: string }];
        };
        posts: {
            totalCount: number;
            list: [
                {
                    id: number;
                    title: string;
                    body: string;
                    comments: {
                        totalCount: number;
                        list: [{ email: string; name: string; body: string }];
                    };
                }
            ];
        };
    };
};

type User = APIResponse['user'];
type Post = User['posts']['list'][0];
type Review = Post['comments']['list'][0];
```

```ts
export function renderUser(user: User) {
    return `
        <div>
            <h2>${user.name}</h2>
            <p>${user.email}</p>
            <p>${user.followers.totalCount} followers</p>
            <p>${user.posts.totalCount} posts</p>
        </div>
    `;
}

export function renderPost(post: Post) {
    return `
        <div>
            <h3>${post.title}</h3>
            <p>${post.body}</p>
            <p>${post.comments.totalCount} comments</p>
        </div>
    `;
}

export function renderReview(review: Review) {
    return `
        <div>
            <h4>${review.name}</h4>
            <p>${review.email}</p>
            <p>${review.body}</p>
        </div>
    `;
}
```

Typescript

Refinamiento de tipos con keyof

```ts
type ResponseKeys = keyof APIResponse; // 'user'
type UserKeys = keyof User; // 'email' | 'name' | 'followers' | 'posts'
type PostKeys = keyof Post; // 'id' | 'title' | 'body' | 'comments'
type ReviewKeys = keyof Review; // 'email' | 'name' | 'body'
```

```ts
export function getSection(section: UserKeys, response: APIResponse) {
    return response['user'][section];
} // string | follwers | posts

export function getSectionRefined<K extends UserKeys>(
    section: K,
    response: APIResponse
): User[K] {
    return response['user'][section];
}
```

```ts
const post = getSection('posts', response); // string | follwers | posts
const postRefined = getSectionRefined('posts', response); // posts
postRefined.totalCount;
```

Typescript

Refinamiento con Record<key, value>

```ts
type YearRecord = Record<Moths, number>;
```

```
Property 'December' is missing in type '{ January: number; February: number; March: number; April: number; May: number; June: number; July: number; August: number; September: number; October: number; November: number; }' but required in type 'YearRecord'. ts(2741)

'yearRecord' is assigned a value but never used. eslint(@typescript-eslint/no-unused-vars)

const yearRecord: YearRecord

View Problem  Quick Fix... (⌘.)
```

```ts
const yearRecord: YearRecord = {
    January: 1,
    February: 2,
    March: 3,
    April: 4,
    May: 5,
    June: 6,
    July: 7,
    August: 8,
    September: 9,
    October: 10,
    November: 11,
    // December: 12
};
```

Typescript

Implementación de Record<key, value> - Mapped Types

```ts
type MonthInfo = { name: MothNames; days: number; position: number };

type YearInfo = { [key in MothNames]: MonthInfo };

const yearInfo: YearInfo = {
    January: { name: 'January', days: 31, position: 1 },
    February: { name: 'February', days: 28, position: 2 },
    March: { name: 'March', days: 31, position: 3 },
    April: { name: 'April', days: 30, position: 4 },
    May: { name: 'May', days: 31, position: 5 },
    June: { name: 'June', days: 30, position: 6 },
    July: { name: 'July', days: 31, position: 7 },
    August: { name: 'August', days: 31, position: 8 },
    September: { name: 'September', days: 30, position: 9 },
    October: { name: 'October', days: 31, position: 10 },
    November: { name: 'November', days: 30, position: 11 },
    December: { name: 'December', days: 31, position: 12 }
};
```

Typescript

Potencia de los Mapped types

```ts
// Potencia de Key in en ejemplos
type Account = {
    id: number;
    isEmployee: boolean;
    notes: string[];
};

// Hacer todos los campos de Account opcionales
type OptionalAccount = { [P in keyof Account]?: Account[P] };
// Hacer todos los campos de Account nullable
type NullableAccount = { [P in keyof Account]: Account[P] | null };
// Hacer todos los campos de Account solo lectura
type ReadonlyAccount = { readonly [P in keyof Account]: Account[P] };
// Hacer todos los campos de Account solo lectura y opcionales
type ReadonlyOptionalAccount = {
    readonly [P in keyof Account]?: Account[P];
};
// Hacer todos los campos de Account modificables
type MutableAccount = { -readonly [P in keyof Account]: Account[P] };
// Hacer todos los campos de Account requeridos
type RequiredAccount = { [P in keyof Account]-?: Account[P] };
```

Typescript

Built-in types

```ts
// Ejemplo de Typos predefinidos en TypeScript
type PartialAccount = Partial<Account>; // Todos los campos de Account opcionales

type RequiredAccount = Required<Account>; // Todos los campos de Account requeridos

type NotNullableAccount = NonNullable<Account>; // Todos los campos de Account no nullable

type ReadonlyAccount = Readonly<Account>; // Todos los campos de Account solo lectura

type SubAccount = Pick<Account, 'id' | 'isEmployee'>; // Solo id y isEmployee
```

Typescript

Tipos condicionales

```ts
type isString<T> = T extends string ? true : false;
type A = isString<string>; // true
type B = isString<number>; // false

const text = 'Hola';
const num = 1;
type C = isString<typeof text>; // true
type D = isString<typeof num>; // false
```

```ts
type APIError = { code: number; message: string };
type Result = { data: unknown };

type GetAPIType<T> = T extends APIError ? APIError : Result;
type T1 = GetAPIType<APIError>; // APIError
type T2 = GetAPIType<Result>; // Result
type T3 = GetAPIType<string>; // Result!!!!!!!!
```

Typescript

Tipos condicionales

```ts
// Si elevamos una variable de tipo T a un array de tipo []T, entonces el tipo de retorno es un array de T.
type ToArray<T> = T[];
type Ar1 = ToArray<string>; // string[]
type Ar2 = ToArray<number>; // number[]
// Si lo hacemos con un tipo T que sea una union de tipos, entonces el tipo de retorno es un array de dicha union.
// No hay propiedad distributiva en los alias
type Ar3 = ToArray<string | number>; // (string | number)[]

// Si aplicamos un tipo condicional le estamos indicando a TS que distribuya T sobre el tipo array
type ToArray2<T> = T extends unknown ? T[] : T[]; // ⇒ T extends unknown ? T[] : never;
type Ar4 = ToArray2<string>; // string[]
type Ar5 = ToArray2<string | number>; // number[] | string[]
```

Typescript

Built-in types

```ts
type AType = 'a' | 'b' | 'c';
type BType = 'a' | 'b';

// Excluye de A los tipos que no esten en B
type ExcludeAFromB = Exclude<AType, BType>; // 'c'

// Usa los tipos que estan en A y en B
type ExtractAFromB = Extract<AType, BType>; // 'a' | 'b'

// Elimina los tipos null y undefined de un tipo
type OnlyString = NonNullable<string | null | undefined>; // string
```

Typescript

Built-in types con condicionales

```ts
type User = { id: number; name: string; age: number };
type FN = () => User;

// Recupera el tipo devuelto por una funcion
type ReturnFN = ReturnType<FN>; // User

// Omitir un tipo de un objeto
type UserDTO = Omit<User, 'id'>; // { name: string; age: number }

// Tipo en el que resuelve una promesa
type ResolvedType = Awaited<Promise<string>>; // string
```

```ts
async function resolveAPI() {
    return Promise.resolve({id: 1, name: 'test'});
}
type APIElement = Awaited<ReturnType<typeof resolveAPI>>; // {id: number, name: string}
```

Typescript

Tipos de Literal de plantilla

```ts
// Cuando usamos una union en una interpolación, el tipo resultante es la union de todos los posibles tipos
// que pueden formarse a partir de los miembros de la union.
type EmailLocaleId = 'email 1' | 'email 2';
type SMSLocaleId = 'sms 1' | 'sms 2';
type LocaleId = `Locale ${EmailLocaleId | SMSLocaleId}`; // "Locale email 1" | "Locale email 2" | "Locale sms 1" | "Locale sms 2"
```

```ts
// En el caso de combinaciones se realizan todas las combinaciones posibles
type Language = 'english' | 'spanish' | 'french';
type Country = 'usa' | 'mexico' | 'canada';
// "english usa" | "english mexico" | "english canada" | "spanish usa" | "spanish mexico" | "spanish canada" | "french usa" | "french mexico" | "french canada"
type Speak = `${Language} ${Country}`;
```

```ts
type Measurement = 'temperature' | 'humidity' | 'pressure';
type Measurements = Record<`${Measurement}s`, number[]>;
const measurements: Measurements = {
    temperatures: [],
    humiditys: [],
    pressures: []
};
```

```ts
function addMeasurement(measurement: Measurement, value: number) {
    measurements[`${measurement}s`].push(value);
}
const newMeassurements = addMeasurement('temperature', 10);
```

Typescript

Tipos nominales

```ts
export type CompanyID = string & { readonly brand: unique symbol };
export type OrderID = string & { readonly brand: unique symbol };
export type UserID = string & { readonly brand: unique symbol };
export type ID = CompanyID | OrderID | UserID;

function generateCompanyID(): CompanyID {
    return v4() as CompanyID;
}

function generateOrderID(): OrderID {
    return v4() as OrderID;
}

function generateUserID(): UserID {
    return v4() as UserID;
}
```

```ts
const companyID = generateCompanyID();
const orderID = generateOrderID();
const userID = generateUserID();

function printOrderID(id: OrderID): void {
    console.log(id);
}

printOrderID(orderID);
printOrderID(companyID);
printOrderID(userID);
```

Typescript

Prototype

```
import './array-zip';


console.log([4, 5, 6].zip([1, 2, 3]));
```

```
interface Array<T> {
    zip<U>(other: Array<U>): Array<[T, U]>;
}

function zip<T, U>(this: Array<T>, other: Array<U>): Array<[T, U]> {
    const result: Array<[T, U]> = [];
    for (let i = 0; i < this.length; i++) {
        result.push([this[i], other[i]]);
    }
    return result;
}

Array.prototype.zip = zip;
```

Typescript

Companion Pattern

```ts
type Unit = 'EUR' | 'USD' | 'GBP' | 'JPY';
type Currency = {
    unit: Unit;
    value: number;
};

type CurrencyCompanion = {
    DEFAULT: Unit;
    from(value: number, unit: Unit): Currency;
};
const Currency: CurrencyCompanion = {
    DEFAULT: 'EUR',
    from(value: number, unit: Unit = Currency.DEFAULT): Currency {
        return {
            unit,
            value
        };
    }
};

const currency = Currency.from(10, 'USD');
const currency2 = Currency.from(10, 'EUR');
const currency3 = Currency.from(10, 'GBP');
```