Advanced Programming
4th Exercise: Advanced Type Systems

Programming Group
guido.salvaneschi@unisg.ch

University of St.Gallen

1 Goals

In this exercise, you'll practice advanced type system features in TypeScript for type-level programming and enforcing program invariants. In particular, you'll learn how to implement a type-safe vector and a type-safe formatter.

2 Type Safety in TypeScript

TypeScript is a statically-typed superset of JavaScript that compiles to plain JavaScript. Static typing and type safety help catch errors early at compile time.

2.1 Advanced Types

TypeScript has a rich type system that includes Objects, Generics, Union Types, Intersection Types, Structural Types, Conditional Types, and Mapped Types. Open the file `01-advanced-types.ts` and complete the following tasks.

Task 1: Generics

Generics in TypeScript can be used with the following syntax:

```
type Vector<A> = A[];
```

Define the type `Pair`, which represents a pair of generic elements.

Task 2: Structural Types and KeyOf Operation

In TypeScript, you can specify the type of an object with structural types:

```
type Employee = {
    name: string;
    age: number;
    salary: number;
}
```

Define the type `Customer`, which model a customer with a name, an age, and a balance (i.e. a certain amount of money). Using `Customer`, define the type `CustomerBalance`, which represents the balance of a customer. Define the type `CustomerKey`, which represents the set of all fields of `Customer`.

Task 3: Union Types

Union types describe values that are compatible with **at least one** among multiple types:

```
type Boolean = true | false;
```

Define the type `EmployeeOrCustomerKey`, which contains any possible field that you can use to query an `Employee` or a `Customer`.

Task 4: Intersection Types

Intersection types describe values that are compatible with **all** among multiple types:

```
type Never = true & false;
```

Define the type `EmployeeAndCustomerKey`, which contains any possible field that you can use to query both an `Employee` and a `Customer`.

Task 5: Conditional Types

Conditional types select one of two possible types based on a subtype condition:

```
type SubType<A, B> = A extends B ? true : false;
```

Define the type function `HasKey<K, T>`, which is true if an object `T` has the key `K`, false otherwise.

```
type HasKey<K, T> = K extends keyof T ? true : false;
```

Task 6: Type Equality

Define the type function `TypeEqual<A,B>`, which is true if A and B are the same type, false otherwise.

```
type TypeEqual<A, B> =
    B extends A
        ? A extends B
            ? true
            : false
        : false;
```

Task 7: Type Guards

Define the type function `Not<A>`, which takes a boolean type `A` and returns its negation.

```
type Not<A extends boolean> = A extends true ? false : true;
```

Task 8: Mapped Types

With mapped types, we can create new object types based on existing ones. Here is the syntax:

```
type A = {
    [K in KS as B<K>]: C
};
```

So that `A` is the type of an object, whose fields are all `K` inside `KS`. All `K` are mapped to `B<K>`, and the type of each field of `A` has type `C`.
Use mapped types to create the type function `IntersectionNumber<A,B>`, which is an object whose fields are those both in `A` and `B`, but whose field types are all `number`.

```
type IntersectionNumber<A, B> = {
    [K in (keyof A & keyof B)]: number
}
```

Task 9: Mapped Types with Conditional Types

Use mapped types to create the type function `Intersection<A,B>`, which is an object (i) whose fields are those both in `A` and `B`, and (ii) the field type is the one in `A`, (iii) only if it is a subtype of the corresponding field in `B`.

```
type Intersection<A, B> = {
    [K in (keyof A & keyof B) as A[K] extends B[K] ? K : never]: A[K];
}
```

2.2 Type-Level Peano Numbers

Peano numbers are a simple way to represent natural numbers using only zero and the successor function. You can imagine a Peano number as a sequence of `1`, whose sum is the actual number it represents. Then, zero is the empty sequence, and the successor concatenates a `1` to the sequence. Open the file `02-peano-numbers.ts` and complete the following tasks.

Task 10: Peano Numbers

Define the types `Peano`, which represents a Peano number, `Zero`, which is the Peano number zero, and `Succ<N>`, which is the successor of the Peano number `N`.
You can apply the concatenation of two sequences in TypeScript with the following syntax:

```
type Concat<A extends any[], B extends any[]> = [...A, ...B];
```

```
type Peano = 1[];
type Zero = [];
type Succ<N extends Peano> = [1, ...N];  \\ or Concat<[1], N>
```

Task 11: Peano To Number

Sequences in TypeScript have a `length` field, which represents the number of elements in the sequence. Define the type function `ToNumber<N>`, which converts a Peano number `N` to its corresponding number.

```
type ToNumber<N extends Peano> = N['length'];
```

Task 12: Number To Peano

Define the type function `ToPeano<N>`, which converts a number `N` to its corresponding Peano number.

```
type ToPeano<I extends number, Result extends Peano = Zero> =
  ToNumber<Result> extends I
    ? Result
    : ToPeano<I, Succ<Result>>;
```

Task 13: Peano Addition

Define the type function `Add<X,Y>`, which computes the sum of two Peano numbers `X` and `Y`.

```
type Add<X extends Peano, Y extends Peano> = [...X, ...Y];
```

Task 14: Ordering of Peano Numbers

Let's look together at the type function `LessThan<X,Y>`, which is true if the Peano number `X` is less than the Peano number `Y`, false otherwise. Note that, in the definition, we use `infer` to extract a type variable from a type.

```
type LessThan<X extends Peano, Y extends Peano> =
    X extends Zero
        ? Y extends Zero
            ? false // 0 < 0
            : true  // 0 < Y
        : X extends Succ<infer PX extends Peano>
            ? Y extends Zero
                ? false // X < 0
                : Y extends Succ<infer PY extends Peano>
                    ? LessThan<PX, PY> // X-1 < Y-1
                    : never
            : never;
```

Task 15: Fibonacci Peano Numbers

Define the type function `Fibonacci<N extends Peano>`, which computes the N-th Fibonacci Peano number. Remember that the Fibonacci sequence is defined as:

- `F(0) = 0`
- `F(1) = 1`
- `F(n+2) = F(n+1) + F(n)` for `n >= 0`

```
type Fibonacci<N extends Peano> =
    N extends Zero
        ? Zero // F(0) = 0
        : N extends One
            ? One // F(1) = 1
            : N extends Succ<Succ<infer P extends Peano>>
                ? Add<Fibonacci<Succ<P>>, Fibonacci<P>> // F(N+2) = F(N+1) + F(N)
                : never;
```

2.3 Type-Safe Vectors

Vectors (or arrays) are a common data structure in programming. However, they can lead to runtime errors if accessed out of bounds. In this section, we will implement a type-safe vector in TypeScript that prevents out-of-bounds access at compile time. Open the file `03-safe-vectors.ts` and complete the following tasks.

Task 16: Unsafe Vectors

To define a safe vector, we will keep track of its length at the type level, using Peano numbers. We start by defining the type lambda:

```
type UnsafeVector<T, N extends Peano> = {
    readonly content: T[];
};
```

Note however that this definition is unsafe, because the resulting type does not depend on `N`, and therefore we lose information about the length of the vector. To make it clearer, try to think about the actual type you get when calling `UnsafeVector<number, Zero>` and `UnsafeVector<number, One>`.

Task 17: Phantom Types

The type of the vector should depend on `N`, but this information will only be used at compile time, and not at runtime. Ideally, we would like to have a type that depends on `N`, but it is inaccessible at runtime (or even erased). This is usually called a **Phantom Type**.
In TypeScript, all of these requirements can be satisfied using the type:

```
type Phantom<X> = (_: never) => X;
```

Note how this type depends on `X`, but it is impossible to construct an argument to retrieve `X` at runtime (unless you give up static typing altogether by casting to `never`).

Task 18: Safe Vectors

We can now define the correct type for safe vectors:

```
type SafeVector<T, N extends Peano> = {
    readonly content: T[];
    readonly size: Phantom<N>;
};
```

Now, we define the methods to manipulate safe vectors, which have to correctly track the length of the vector:

1. `empty`: creates a vector of length zero.
2. `push`: creates a vector of length `N+1` by adding an element to a vector of length `N`.
3. `pop`: creates a vector of length `N-1` by removing the last element from a vector of length `N`. Notably, `N` must not be zero.
4. `get`: retrieves the element at index `I` from a vector of length `N`. Notably, `I` must be less than `N`.

Task 19: Safe Vector as Class

Look at `SafeVectorAlternative` as a cleaner implementation of safe vectors. Note how we can avoid phantom types by leveraging visibility modifiers.

2.4 Type-Safe Formatter (Optional)

In C-like languages it is common to find a function `format` or `sprintf` that takes some string describing a string and values to replace in specific parts of the string. For example, in Rust,

```
format!("Hello, %s! It's %d degrees today!", "HSG", 19)
```

produces the string `"Hello HSG! It's 19 degrees today!"`.

But what happens if we had swapped the arguments? Or if we give too many? Or too few? In C that could turn sour really quickly. If you're lucky you read garbage on the screen or quit with a segmentation fault. In the worse case, you open a backdoor to be exploited.

With TypeScript, we can write a function that examines the format string so that we're guaranteed, at compile time, that the arguments are as expected. In this section, we will implement this function. Open the file `04-safe-formatter.ts` and complete the following tasks.

Task 20: Unsafe Format

The strategy is the following: we will define a function `format` that takes the parts of the formatter, then it returns a function expecting the values. For example, we will write the example above as follows:

```
format("Hello ", "%s", "! It's ", "%d", " degrees today!")("HSG", 19)
```

We will use a crucial operator: the spread operator `...`. It allows us to treat the (variadic) arguments of a function as an array, and it allows us to pattern match on an array to select its head(s) and tail. Start by writing the type of the as-of-yet-unsafe type function `UnsafeFormat`.

```
type UnsafeFormat = (...format: string[]) => (...args: string[]) => string;
```

Task 21: Examine Format

We need a type-level function, `ExamineFormat<F>` that takes an array of strings, for example,

```
["hello", "%s", "%d", "=>", "%s"]
```

and produces a list of expected argument types for formatting, in this case:

```
[string, number, string]
```

Implement the type function `ExamineFormat<F>`, which constructs a lists of argument types by looking at the elements of the format strings one by one.

```
type ExamineFormat<F> =
    F extends [ "%d", ...infer Tail ] ? [ number, ...ExamineFormat<Tail> ] :
    F extends [ "%s", ...infer Tail ] ? [ string, ...ExamineFormat<Tail> ] :
    F extends [ string, ...infer Tail ] ? ExamineFormat<Tail> :
    F extends [] ? [] :
    never;
```

Task 22: Safe Format

With `ExamineFormat` defined, modify the type of `UnsafeFormat` you presented earlier to define the safe type function `Format`.

```
type Format = <F extends string[]> (...format: F) => (...args: ExamineFormat<F>) =>
    string;
```

Task 23: Safe Format Implementation

Implement the `format` function with the type you proposed. Demonstrate through some examples that it's type-safe. What are examples of a format and arguments which are rejected by the type system?

```
const format: Format =
    <F extends string[]> (...format: F) =>
    (...args: ExamineFormat<F>) =>
    {
        let i = 0;
        let result: string = "";
        for (let j=0; j<format.length; j++) {
            let c = format[j] == "%s" || format[j] == "%d" ? args[i++] : format[j];
            result = result.concat(c);
        }
        return result;
    }
```