Principles of Programming Languages
Assignment 4

Responsible TA: Reem Al-Asam

Submission Date: 08/June/2023

Part 0: Preliminaries

Structure of a TypeScript Project

Every TypeScript assignment will come with two very important files:

- `package.json` - lists the dependencies of the project.
- `tsconfig.json` - specifies the TypeScript compiler options.

Before starting to work on your assignment, open a command prompt in your assignment folder and run `npm install` to install the dependencies.

What happens when you run `npm install` and the file `package.json` is present in the folder is the following:

1. `npm` will download all required modules and their dependencies from the internet into the folder `node_modules`.
2. A file `package-lock.json` is created which lists the exact version of all the packages that have been installed.

What `tsconfig.json` controls is the way the TypeScript compiler (`tsc`) analyzes and typechecks the code in this project. We will use for all the assignments the strongest form of type-checking, which is called the "strict" mode of the `tsc` compiler.

Do not delete or change these files (*e.g.*, install new packages or change compiler options), as we will run your code against our own copy of those files, exactly the way we provide them.

If you change these files, your code may run on your machine but not when we test it, which may lead to a situation where you believe your code is correct, but you would fail to pass compilation when we grade the assignment (which means a grade of zero).

Testing Your Code

Every TypeScript assignment will have Jest as a dependency for testing purposes. In order to run the tests, save your tests in the `test` directory in a file ending with `.test.ts` and run `npm test` from a command prompt. This will activate the execution of the tests you have specified in the test file and report the results of the tests in a very nice format.

An example test file `assignmentX.test.ts` might look like this:

```
import { sum } from "../src/assignmentX";

describe('test sum', () => {
    it('sums 1 and 2', () => {
        expect(sum(1,2)).toEqual(3);
    });
});
```

Every function you want to test must be `export`-ed, for example, in `assignmentX.ts`, so that it can be `import`-ed in the `.test.ts` file (and by our automatic test script when we grade the assignment).

```
export const sum = (a: number, b: number) => a + b;
```

You are given some tests in the `test` directory, just to make sure you are on the right track during the assignment. Make sure they all pass. Add more tests to cover the rest of your implementation.

What to Submit

You should submit a zip file called `<id1>_<id2>.zip` which has the following structure:

```
/
├── answers.pdf
├── part2
│   ├── src
│   │   └── part2.ts
│   └── test
│       └── test.test.ts
└── part3
    ├── src/*
    └── test
        ├── L5-ast.test.ts
        └── L5-typecheck.test.ts
```

You will create the file `answers.pdf` (any common text format is fine) and change the files `part2.ts`, and the files under `part3/src` in the places marked by the string `TODO`. Make sure not to include additional files and folders and specifically avoid `node_modules` which can take a lot of space.

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder. This structure is crucial for us to be able to import your code to our tests. Also, make sure the file is a `.zip` file – not a RAR or TAR or any other compression format.

Part 1: Theoretical Questions

(18 points)

Submit the solution to this part in `answers.pdf`. The file *MUST be a PDF file*.

1. Which of the following typing statement is true / false, explain why (6 points).
   (a) $\{f : [T1 \to T2], g : [T1 \to T2], a : T1\} \vdash (f\ (g\ a)) : T2$
   (b) $\{f : [T1 \times T2 \to T3]\} \vdash (lambda\ (x)\ (f\ x\ 100)) : [T2 \to T3]$
   (c) $\{f : [T1 \to T2]\} \vdash ((lambda\ (x)\ (f\ x))) : [T1 \to T2]$
   (d) $\{f : [T1 \times T2 \to T3], y : T2\} \vdash (lambda\ (x)\ (f\ x\ y)) : [T1 \to T3]$
2. Perform type inference manually on the following expressions, using the Type Equations method. List all the steps of the procedure (12 points):
   (a) `((lambda (f x1) (if x1 (f 1 x1) (f 3 x1))) + #t)`
   (b) `((lambda (f1 x1 y1) (f1 x1 y1)) * 1 3)`

Part 2: Async with TypeScript

(30 points)

Complete the following functions in TypeScript in the file `part2/src/part2.ts`. You are requested to provide types as precise as possible (avoid `any` unless otherwise noted).

Remember that it is crucial you do *not* remove the `export` keyword from the code in the given template.

Question 2.1

(6 points)

In this question you are requested to use promises and ***not** to use* the `async` and `await` keywords!

Implement an asynchronous function called `delayedSum` that takes two numbers as parameters and returns their sum after a specified delay. The function should introduce an artificial delay using `setTimeout`. The `delay` parameter is passed in milliseconds (same as for `setTimeout`).

```
const delayedSum = (a: number, b: number, delay: number): ... =>
   ...
```

Write a test function which verifies that the delay before the returned sum is at least the delay provided as argument. Use `Date.now()` to remember when a call is made and when the promise is resolved. You can subscribe two date values and obtain a duration in milliseconds.

```
const testDelayedSum = () => {
   ...
}
```

Question 2.2

(12 points)

In this question you are requested *to use* the `async` keyword! `Promise` is OK **only** if used as a type or as a static object (e.g., `Promise.all`)

(do not use `new Promise(...)`, `Promise.prototype.then`, `Promise.prototype.catch`, etc.,).

Given the following value type:

```
export type Post = {
    userId: number;
    id: number;
    title: string;
    body: string;
}
```

Write an asynchronous function called `fetchData` that takes a URL as a parameter and uses `fetch` to make an HTTP request to an API endpoint. The function should return a promise that resolves with the response data. Handle any errors that may occur during the request.

The `fetch` Node.js API is an asynchronous primitive which has the following (simplified) signature:

```
function fetch(input: URL): Promise<Response>;

interface Body {
    ...
```

```
    json(): Promise<any>;
}

interface Response extends Body {
    ...
    readonly ok: boolean;
    readonly status: number;
    readonly statusText: string;
    readonly url: string;
}
```

In your code, it is acceptable to cast the value returned by fetch to the type expected from the API without testing at runtime that the returned JSON satisfies the `Post` type requirements.

```
// When invoking fetchData(postsUrl) you obtain an Array Post[]
const postsUrl = 'https://jsonplaceholder.typicode.com/posts';

// When invoking fetchData(postsUrl + postId) you obtain a single Post
// for example: https://jsonplaceholder.typicode.com/posts/1
const postUrl = 'https://jsonplaceholder.typicode.com/posts/';

// When invoking fetchData(invalidUrl) you obtain an error
const invalidUrl = 'https://jsonplaceholder.typicode.com/invalid';
```

Write a test function `testFetchData` that tests both successful calls with array and single post results and an error call to `fetchData`.

Question 2.3

(12 points)

Implement an async function called `fetchMultipleUrls` that takes an array of URLs as a parameter and fetches the data from each URL using `fetch`. The function should return a promise that resolves with an array of the fetched data in the same order as the input URLs.

1. Make sure all the fetch requests are sent at once without waiting in sequence one after the other.
2. Wait for all the requests to complete before resolving the result.
3. If any of the requests fails, the whole call should fail.

Use the `Promise.all` function as part of your implementation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Write a test function `testFetchMultipleUrls` that retrieves 20 different posts from this API endpoint, where the last element is the ID of the requested post:

```
const apiUrl1 = 'https://jsonplaceholder.typicode.com/posts/1';
```

Part 3: Type Checking System

(52 points)

In this part, we will work on the Type Checking system studied in class - on the type checker version https://github.com/bguppl/interpreters/blob/master/src/L5/L5-typecheck.ts. The code attached to this assignment under `part3` contains an updated version of the type system with additions toward the following goals, which you will complete:

1. Complete the type inference code in L5 to support the missing AST expression types program and define.
2. Extend the L5 language and type checking code to support a new compound type `union`.

All modifications with respect to the base system discussed in class are marked with comments of the form `// L51`. You will complete the places marked by the string `TODO L51`.

Question 3.1 Support Type Checking with Program and Define

(12 points)

Complement the code in `src/L5-typecheck.ts` so that the type checker system can deal with the L5 AST nodes of type `program` and `define`.

In `answers.pdf`, write the typing rule for `define`.

```
Typing rule define:
...
```

You must implement the functions `typeOfProgram`, and `typeOfDefine` in file `src/L5-typecheck.ts`.

Question 3.2 Extend L5 to Support Union Types

(40 points)

In the language $L5$ presented in class, the only composite type that we considered was that of procedures. For example, the type of:

```
(lambda (x) (* x x))
```

is `(number -> number)` which is a composite type.

In this question, we will introduce another composite type to describe the union of two other types - similar to the way we can define unions in TypeScript.

To support type unions, we will have to change the following parts of the type checker code:

1. Extend the syntax of the type language TExp to support union.
2. Adjust the definition of `checkEqualType` to consider the case where we compare types that can be unions. We renamed `checkEqualType` to the more appropriate name `checkCompatibleType` in the file in the template.
3. Adjust the typing rules of IfExp expression types to take advantage of union types.
4. Consider the case of how Procedure Types compare when their arguments are of type union in `checkCompatibleType`.

Pay attention that union types can be recursively embedded as in the following example:

```
(lambda ((x : (union number (union boolean string)))) : (union string number)
```

```
  (if (number? x)
      (* x x)
      "Not a number"))
```

(3.2.1) Extend TExp with Union Types

(12 points)

In `src/TExp.ts` the concrete and abstract syntax of TExp type expressions is defined as in $L5$:

```
;; Type language
;; <texp>          ::= <atomic-te> | <compound-te> | <tvar>
;; <atomic-te>     ::= <num-te> | <bool-te> | <void-te>
;; <num-te>        ::= number   // num-te()
;; <bool-te>       ::= boolean  // bool-te()
;; <str-te>        ::= string   // str-te()
;; <void-te>       ::= void     // void-te()
;; <compound-te>   ::= <proc-te> | <tuple-te> | <union-te> TODO L51
;; <non-tuple-te> ::= <atomic-te> | <proc-te> | <tvar>
;; <proc-te>       ::= [ <tuple-te> -> <non-tuple-te> ] // proc-te(param-tes: list(te), return-te: te)
;; <tuple-te>      ::= <non-empty-tuple-te> | <empty-te>
;; <non-empty-tuple-te> ::= ( <non-tuple-te> *)* <non-tuple-te> // tuple-te(tes: list(te))
;; <empty-te>      ::= Empty
;; TODO L51
;; <union-te>      ::= (union <texp> <texp>) // union-te(components: list(te))
;; <tvar>          ::= a symbol starting with T // tvar(id: Symbol, contents; Box(string|boolean))

;; Examples of type expressions
;; number
;; boolean
;; void
;; [number -> boolean]
;; [number * number -> boolean]
;; [number -> [number -> boolean]]
;; [Empty -> number]
;; [Empty -> void]

;; TODO L51
;; Support the following type expressions:
;; [union number boolean]
;; [union [union number boolean] string]
;; [Empty -> [union boolean number]]
;; [union [T1 -> T1] [Empty -> T1]]
```

You must extend the implementation to support the concrete and abstract syntax of the TExp type language with unions as indicated in the three `TODO L51` places.

Note that the abstract syntax of nested union types is a flat `UnionTExp` data structure. This takes into account that union is a **commutative** and **associative** operator in set theory, and accordingly:

```
Concrete syntax:
    (union boolean number)
    (union number boolean)
Abstract syntax:
```

```
    {tag: UnionTExp, components: [{tag: BoolTExp}, {tag: NumTExp}]}
```

Both concrete types are represented by the same abstract syntax where the components are sorted in alphabetical order of the `unparseTExp` values.

Similarly:

```
Concrete syntax:
    (union string (union boolean number))
    (union number (union string boolean))
Abstract syntax:
    {tag: UnionTExp,
     components: [{tag: BoolTExp}, {tag: NumTExp}, {tag: StringTExp}]}
```

You must update:

1. The definition of `TExp` (with the introduction of `UnionTExp`)
2. Update the parser to support union (function `parseCompoundTExp` must be changed).
3. Update the unparser to support union (function `up` inside `unparseTExp` must be updated). When unparsing union types that have more than two components, write them in alphabetical order in the following parenthesizes structure: `(union a (union b c))` and not `((union a b) c)`.

(3.2.2) Type Compatibility: Define checkCompatibleType

(16 points)

The type checker in L5 relies on **type invariance**: it determines that an expression of type `T1` passed where a type `T2` is expected is acceptable exactly when `T1 = T2`. This is implemented in file `L5-typecheck.ts` in the function `checkEqualType`:

```
// TODO L51
// Purpose: Check that type expressions are compatible
// as part of a fully-annotated type check process of exp.
// Return an error if te1 is not compatible with te2 - true otherwise.
// Exp is only passed for documentation purposes.
export const checkCompatibleType = (te1: TExp, te2: TExp, exp: Exp): Result<true> =>
  equals(te1, te2) ? makeOk(true) :
  bind(unparseTExp(te1), (te1: string) =>
    bind(unparseTExp(te2), (te2: string) =>
        bind(unparse(exp), (exp: string) =>
            makeFailure<true>(`Incompatible types: ${te1} and ${te2} in ${exp}`))));
```

With the introduction of **union types**, this definition of invariance is not appropriate anymore. Instead, we must account for the fact that a type can be a sub-type of another type, in which case a value of the sub-type can be accepted where the super-type is expected. For example:

```
(define f (lambda ((x (union number boolean))) : (union number string)
  (if (number? x)
      (* x x)
      "Not a number")))

(f 1)      ;; This is type safe because number is compatible with (union number boolean)
(f (f 1)) ;; This is not type safe because there is no inclusion between
          ;; (union number boolean) and (union number string)
```

Update the function `checkCompatibleType(te1, te2, exp)` so that it verifies that `te1` is acceptable where

`te2` is expected (that is, `te1` is a sub-type of `te2` or equal to `te2`). Pay attention to the fact that `te1` or `te2` can be any TExp, including union types.

Make sure to enumerate all cases where a type expression can be considered a sub-type of another and implement and verify all appropriate test cases in `test\L5-typecheck.test.ts`).

In this assignment, when comparing a Type Variable T_1 with any other type, T_1 is only compatible with itself - all other comparisons involving a TVar and another type will fail.

We will address the case of comparing `ProcTExp te1` and `ProcTExp te2`, in (3.2.4) to complete this function. At this stage, you can skip this case in your first implementation, and then finish this case later.

Write tests to cover type checking of union types for primitives, applications of procedures that receive union typed parameters and return union types, and let expressions.

(3.2.3) Change the IfExp Typing Rule

(6 points)

In the definition of the $L5$ type system, we introduced the following restrictions:

1. The test part must be of type boolean
2. The `then` and `else` parts must be of the same type.

This was implemented in this typing procedure in the type checker:

```
// Purpose: compute the type of an if-exp
// Typing rule:
//   if type<test>(tenv) = boolean
//      type<then>(tenv) = t1
//      type<else>(tenv) = t1
// then type<(if test then else)>(tenv) = t1
export const typeofIf = (ifExp: IfExp, tenv: TEnv): Result<TExp> => {
    const testTE = typeofExp(ifExp.test, tenv);
    const thenTE = typeofExp(ifExp.then, tenv);
    const altTE = typeofExp(ifExp.alt, tenv);
    const constraint1 = bind(testTE, testTE => checkEqualType(testTE, makeBoolTExp(), ifExp));
    const constraint2 = bind(thenTE, (thenTE: TExp) =>
                            bind(altTE, (altTE: TExp) =>
                                checkEqualType(thenTE, altTE, ifExp)));
    return bind(constraint1, (_c1: true) =>
                bind(constraint2, (_c2: true) =>
                    thenTE));
};
```

With the support of union types, we can now support heterogeneous `IfExp` expressions as in: `(if #t 1 "ok")` which has type `(union number string)`.

Update `typeOfIf` according to this new typing rule. Define the new exact typing rule in the comments of the procedure.

To this end, you must implement the following function: `makeUnion(te1: TExp, te2: TExp): UnionTExp` which computes the UnionTExp which represents the union of the two types `te1` and `te2`.

(3.2.4) Compare ProcTExp Types with Unions

(6 points)

We learned in class (see https://bguppl.github.io/interpreters/class_material/1.3TypescriptTypeChecking.html#comparing-return-types) that procedure types are **contravariant** on parameter types and **covariant** on return type.

Let us consider the following example to understand this notion:

```
;; [N | B -> S]
(define (f : ((union number boolean) -> string))
  (lambda ((x : (union number boolean))) : string
    "F OK"))

;; [N -> S]
(define (g : (number -> string))
  (lambda ((x : number))) : string
    "G OK"))

;; [[N -> S] -> B]
(define (h : ((number -> string) -> boolean))
  (lambda ((t : (number -> string))) : boolean
    (t 1)))

;; [[N |B -> S] -> B]
(define (i : (((union number boolean) -> string) -> boolean))
  (lambda ((t : ((union number boolean) -> string))) : boolean
    (t #t)))

;; The following calls are both type safe:
(h g) ;; g has exactly the type expected by h
(h f) ;; f has a type more specific than g
      ;; in the body of h, we call (f 1) which is ok.

;; The following call is type safe:
(i f) ;; f has exactly the type expected by i
;; But the following call is NOT type safe:
(i g) ;; g has a type more general than f - it cannot replace f
      ;; because in the body of i we call (g #t)
      ;; which is not ok for g: N -> S.
```

More generally, when we compare two `ProcTExp` types te_1 and te_2 in `checkCompatibleType(te1, te2)`, we will conclude that $te_1 \subseteq te_2$ iff:

1. $te_1 =$ `ProcTExp(paramTEs: (p11...p1n1), returnTE: r1)`
2. $te_2 =$ `ProcTExp(paramTEs: (p21...p2n2), returnTE: r2)`
3. $n_1 = n_2$
4. $r_1 \subseteq r_2$
5. $\forall i \in [1 \ldots n_1], p_{2,i} \subseteq p_{1,i}$ (Note the inversion!)

Update the procedure `checkCompatibleType` accordingly and prepare tests to cover all cases of positive and

negative compatibility checks.

Good Luck and Have Fun!