Principles of Programming Languages
Assignment 4

Responsible TA: Amit Portnoy

Submission Date: 15/June/2022

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
└── part3
    └── src/*
```

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

Submit the solution to this part in `answers.pdf`. We can't stress this enough: the file *has to be a PDF file.*

1. Which of the following typing statement is true / false, explain why (5 pts).

   (a) $\{f : [T2 \to T3], g : [T1 \to T2], a : Number\} \vdash (f\ (g\ a)) : T3$

   (b) $\{x : T1, y : T2, f : [T2 \to T1]\} \vdash (f\ x) : T1$

   (c) $\{f : [T1 \to T2], x : T1\} \vdash ((lambda\ ()\ (f\ x))) : T2$

   (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 pts):

   (same level as in the PS)

   (a) `((lambda (f x1) (f 1 x1)) + #t)`

   (b) `((lambda (f1 x1) (f1 x1 1)) + *)`

Part 2: Async Fun with TypeScript

Complete the following functions in TypeScript in the file `part2/src/part2.ts`. You are requested to use generic types where appropriate and 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

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

(a) (8 pts)

Implement the following function that wraps a simple key-value table service:

```
function makeTableService<T>(sync: (table?: Table<T>) => Promise<Table<T>>): TableService<T> {
    // optional initialization code
    return {
        get(key: string): Promise<T> {
            return Promise.reject('not implemented')
        },
        set(key: string, val: T): Promise<void> {
            return Promise.reject('not implemented')
        },
        delete(key: string): Promise<void> {
            return Promise.reject('not implemented')
        }
    }
}
```

Where type `Table<T>` is an immutable object with `string` keys and generic `T` as values. The parameter `sync` represents an internal implementation layer (e.g., a file or database table) and it will be provided by our tests. When called without a parameter it returns the entire up-to-date internal table (it may fail asynchronously), when called with a table parameter it will first update the internal representation and return its new up-to-date state.

'get' should call `sync` and resolve with the value matching the 'key' on the most up-to-date table.

'set' and 'delete' should first call `sync` to get the most up-to-date table wait for it to resolve and then call sync with the changed table. The promise should resolve without any value (just call `resolve()`) if successful (after the second set/delete update sync is resolved).

If a value is missing during 'get' or 'delete', you are requested to reject the Promise with the following constant (it exists in the template. Do not create a new one):

```
export const MISSING_KEY = "___MISSING_KEY___"
```

Otherwise, if `sync` fails, you should reject with the same error.

(b) (3 pts)

Write a function, `getAll<T>`, that accepts a `TableService<T>` and a list of keys and returns a promise containing a list of the matching values for the store. If a key is missing the promise should reject with the `MISSING_KEY` constant.

Question 2.2

(12 pts)

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 Reference = {table: string, key: string}
```

Write an async function `constructObjectFromTables` that accepts an object that map strings to table services (as in the previous question) and a reference as above. The function should access the table from the reference's 'table' property and return the value matching the reference's 'key' property.

Before returning the resulting object, do the following recursively:

Go over the values of the object (you can use 'Object.entries' and 'Object.fromEntries') and, if a value contains the 'table' property (use " 'table' in obj "), assume this value is reference and search for the key in the appropriate table. Continue, this process until all references are replaced by their value.

For example, you may be given 'Book' object that refers to an 'Author':

```
type Author = {
    firstName: string,
    lastName: string,
}
type Book = {
    title: string,
    author: { table: 'authors', key: string },
}
```

Then given a 'book' reference, e.g., `{table: 'books', key: 'book_key_A'}`, you may fetch the following object from a 'books' table `{title: 'dune', author: {table: 'authors', key: 'author_key_A'}`, which should map to `{title: 'dune', author: {firstName: 'Frank', lastName: 'Herbert'}}` using an 'authors' table.

Each object may contain references (e.g., authors above may contain, Address, which refers to a third table), but there is no need to deep scan an object (e.g., if a book contains an array property, you don't need to scan the array for references). You can assume that there are no reference loops.

The final return async value should be a fully constructed object with all references replaced by a corresponding object.

If a key is missing the promise should reject with the `MISSING_KEY` constant. If a table is missing reject with the following constant:

```
export const MISSING_TABLE_SERVICE = "___MISSING_TABLE_SERVICE__"
```

Question 2.3

(10 pts)

Write the functions `lazyProduct` and `lazyZip` that take two parameter-less generator functions (you can find the signatures in the template).

These functions then return a new generator function which lazily applies product or zip (respectively) during the iteration. That is, you are requested not to convert the original generator to an array at any point.

The product operation of two lists is the Cartesian product, where the list are iterated from left to right. That is, given `[1, 2, 3]` and `['a', 'b']` their product is
`[[1, 'a'], [1, 'b'], [2, 'a'], [2, 'b'], [3, 'a'], [3, 'b']]`.

The zip operation iterates of the two lists and return a new list in the same length that "pairs up" elements from both lists (you can assume both input generators will return the same number of elements). That is, given `[1, 2, 3]` and `['a', 'b', 'c']` their zip is
`[[1, 'a'], [2, 'b'], [3, 'c']]`.

Question 2.4

(14 pts)

In this question you can choose whether to use async functions, promises, or generators as needed.

Implement the following function that wraps a *reactive* key-value table service:

```
export async function makeReactiveTableService<T>(sync: (table?: Table<T>) => Promise<Table<T>>, optimis
    // optional initialization code
    let _table: Table<T> = await sync()
    const handleMutation = async (newTable: Table<T>) => {
        // TODO implement!
    }
    return {
        get(key: string): T {
            if (key in _table) {
                return _table[key]
            } else {
                throw MISSING_KEY
            }
        },
        set(key: string, val: T): Promise<void> {
            return handleMutation(/* TODO */)
        },
        delete(key: string): Promise<void> {
            return handleMutation(/* TODO */)
        },
        subscribe(observer: (table: Table<T>) => void): void {
            // TODO
        }
    }
}
```

The new reactive service is similar to that in Question 2.1 with the following differences:

1. The 'make' function itself is now async. It retrieves the initial table when called.
2. No need to sync on get or before mutations.
3. The new `subscribe` function can be called with an observer function which should be called every time the underlying table is changed. It should receive the updated table as a parameter.
4. There is a new 'optimistic' flag: If 'true' then the observer should be called immediately when a change is requested. The update should later be reverted if the mutation failed by calling the observer with the previous table value. Otherwise, if the flag false then the observer should only be called after a mutation promise was fulfilled.

Part 3: Type Checking System

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 towards the following goals, which you will complete:

1. Complete the type checking code in L5 to support the missing AST expression types quoted literal expressions and `set!`.
2. Extend the L5 language and type checking code to support a new `record` construct for user-defined types.

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`.

(3.1) Support Type Checking with Quoted Literal Expressions and Set!

(6 pts)

Complement the code in `src/L51-typecheck.ts` so that the type checker system can deal with the L5 AST nodes of type `literal` and `set!`.

In `answers.pdf`, write the typing rule for `LitExp` and `SetExp` expressions:

```
Typing rule set!:
  ...
```

You must implement the functions `typeofLit`, and `typeofSet` in file `src/L51-typechecker.ts`.

(3.2) Extend L5 with Record Types

(30 pts) (6 pts per subquestion)

The main programming pattern that we have adopted when writing interpreters is that of disjoint union types and functions that enumerate all cases of the disjoint union to perform structural induction (an operation we called "folding the algebraic data type" in class).

In this section, we introduce a programming construct to encourage the use of this pattern called `record`. Records allow the programmer to define new compound types with the `define-type` special form, and to traverse record values with the `type-case` special form. The following example demonstrates how these constructs are used:

```
> (define-type Shape
    (circle (radius : number))
    (rectangle (width : number)
               (height : number)))

> (define (area : (Shape -> number))
    (lambda ((s : Shape)) : number
      (type-case Shape s
        (circle (r) (* (* r r) 3.14))
        (rectangle (w h) (* w h)))))

> (area (make-circle 1))
3.14
```

```
> (area (make-rectangle 2 3))
6

> (define (r1 : Shape) (make-rectangle 2 3))

> (circle? r1)
#f

> (rectangle? r1)
#t
```

This program is equivalent to the following typical TypeScript pattern:

```typescript
type Shape = Circle | Rectangle;
interface Circle {
    tag: "Circle";
    radius: number;
}
interface Rectangle {
    tag: "Rectangle";
    width: number;
    height: number;
}

const isShape = (x: any): x is Shape => isCircle(x) || isRectangle(x);
const isCircle = (x: any): x is Circle => x.tag === 'Circle';
const isRectangle = (x: any): x is Rectangle => x.tag === 'Rectangle';

const makeCircle = (radius: number): Circle => ({tag: "Circle", radius});
const makeRectangle = (width: number, height: number): Rectangle =>
  ({tag: "Rectangle", width, height});

const area = (s: Shape): number =>
    isCircle(s) ? 3.14 * s.radius * s.radius :
    isRectangle(s) ? s.width * s.height :
    s;

console.log(area(makeRectangle(2,3))); // 6
console.log(area(makeCircle(1)));      // 3.14
```

Note the different naming conventions induced by the Scheme-like definitions:

- Type predicates: isCircle, isRectangle, isShape vs. circle?, rectangle?, shape?
- Value constructors: makeCircle, makeRectangle vs. make-circle, make-rectangle

Our task is to extend L5 to support records, including type checking with record values.

You are provided with a parser for L51 which defines new concrete and abstract syntax for `define-type` and `type-case`. The definitions are included in two files:

1. L5-ast.ts
2. TExp.ts

Pay attention to the definition of the new types for `UserDefinedNameTExp`, `Field`, `Record` and `UserDefinedTExp`.

To support the definition of the type predicates, the new type `any` is also defined (`AnyTExp`).

The evaluation of the `define-type` expression defines a record constructor for each record covered by the user defined type, which takes one parameter for each of the fields in the record definition. For example:

```
> (define-type Shape
    (circle (radius : number))
    (rectangle (width : number)
               (height : number)))
```

This defines two constructors: `make-circle` and `make-rectangle`, and three type predicates: `circle?`, `rectangle?`, `Shape?`. It defines three types: `Shape`, `circle` and `rectangle`.

The type of these automatically defined functions is:

```
make-circle: (number -> circle)
circle?: (any -> boolean)
make-rectangle: (number * number -> rectangle)
rectangle?: (any -> boolean)
Shape?: (any -> boolean)
```

Your mission is to extend the type checking system to support user defined `record` and user defined types.

To this end, you must define typing rules that concern user-defined and record types, `any` and the `type-case` expression.

(3.2.1) Type Compatibility

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: Change this definition to account for user defined types and any.
// Purpose: Check that type expressions are equivalent
// as part of a fully-annotated type check process of exp.
// Return an error if the types are not compatible - true otherwise.
// Exp is only passed for documentation purposes.
const checkEqualType = (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 the type `any` and disjoint union types such as the `Shape` example, 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, the function `area` in the example above expects a parameter of type `Shape` but it can be invoked with a parameter of type `rectangle` or `circle`.

Update the function `checkEqualType(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`, user-defined types, records or names of user defined types.

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

Because the relations of sub-types relies on user-defined types in the whole program, the signature of

`checkEqualType` must be changed to add a Program parameter as in
`checkEqualType(te1, te2, exp, program)`.

You are provided with eight new procedures in `src\L5-typecheck.ts` to help you implement this task (they all appear at the beginning of the file with a comment L51):

- `getTypeDefinitions`
- `getDefinitions`
- `getRecords`
- `getItemByName`
- `getUserDefinedTypeByName`
- `getRecordByName`
- `getRecordParents`
- `getTypeByName`

You will also note that a new parameter `Program p` is used to pass the required information about user defined types from the `typeOfExp` main procedure to the sub-procedures `typeofXXX` which are invoked from there until `checkEqualType` is invoked. (In other words, type checking is now an operation that must be performed globally over a complete program including all type definitions and global variable definitions).

In this extended type system, names of user-defined types (implemented as `UserDefinedTypeNameTExp`) refer to user-defined types or to records. A partial is defined over type expressions - so that for example, if `(define-type UD (R1 ...) (R2 ...))` is defined, then `R1 < UD` and `R2 < UD`. Similarly, all type expressions are more specific than the `any` universal type.

You must implement the two functions `isSubType` and `checkEqualType` (which invokes `isSubType`) according to the rules of this type system. Refer to the test cases for `checkEqualType` to verify your definition (you may need to define more test cases to cover all the code of the functions).

(3.2.2) Typing Rules

Define the typing rule for `define-type` and `type-case` using the same notation as in typing rules we saw in class, for example:

```
Typing rule Application:
For every: type environment _Tenv,
           expressions _f, _e1, ..., _en, n >= 0 , and
           type expressions _S1, ..., _Sn, _S, and
           expression _class_value and
           symbols _m1, ..., _mk, k >= 0 and
           type expressions _U1, ..., _Uk

Procedure with parameters (n > 0):
    If   _Tenv |- _f : [_S1 * ... * _Sn -> _S],
         _Tenv |- _e1 : _S1, ..., _Tenv |- _en : _Sn
    Then _Tenv |- (_f _e1 ... _en) : _S

Parameter-less Procedure (n = 0):
    If   _Tenv |- _f : [Empty -> _S]
    Then _Tenv |- (_f) : _S
```

```
// Your turn:
Typing rule Define-type:
...

Typing rule Type-case:
...
```

Write the answer in the file answers.pdf.

Pay attention that in a manner similar to the typing rule for `IfExp`, we expect the typing rule for `type-case` to return compatible types across all the branches of the `type-case`.

You have to implement the functions `typeofIf` and `typeofTypeCase` according to the updated typing rules taking into account sub-typing relations among the alternative branches of the expressions.

Your implementation must use the new function `checkCoverType` which is provided (and the functions it invokes). Consult the test cases in `test\L5-typecheck.test.ts` to understand the behavior of these functions.

(3.2.3) Initial TEnv

When type checking a program, we may encounter the functions implicitly defined when `define-type` is invoked (value constructors and type predicates). We must be able to retrieve their type. To this end, we will initialize a type environment where the type of these implicit procedures is defined.

Implement the function `initTEnv` which takes a Program AST `p` as input and returns an initial type environment in which the types of all the user defined value constructors and type predicates are registered.

You will use `getTypeDefinitions` and `getRecords` in this definition.

In addition, use `getDefinitions` to extend the initTEnv function to also include the type definition of all globally defined variables. The reason this is needed is that the typing rule for define-expressions shown in the code does not support mutually recursive functions like the `odd/even` examples we saw in class. (You should confirm this by writing an appropriate test case.) Adding global variables into `initTEnv` will enable you to analyze mutually recursive functions safely. Add a test case to verify this behavior.

Consult the test cases for `initTEnv` in `test\L5-typecheck.test.ts`.

(3.2.4) Semantic Checks on User Defined Types and type-case

As part of the type checking `type-case` expressions, we must verify that semantic aspects of the expression are validated:

- When a `type-case` is used on user-defined type UD, we must have exactly one clause for each of the constituent sub-types of the UD type. For example, if `(define-type UD (R1 ...) (R2 ...))` is defined, then we expect in `(type-case x UD ...)` to have one clause for `R1` and one for `R2` (in any order). We do not want any other record id to occur in this `type-case`.
- In each of the case clauses, the number of variable declarations that are added must correspond to the number of fields defined in the corresponding Record type. For example, if `(define-type UD (R1 (f1 : number)) (R2))` is defined, then we expect in `(type-case x UD (R1 (x) ...) (R2 () ...))` - where `(x)` has one variable declaration because `R1` has one field, and similarly for `R2`.

Implement this test in the function `checkTypeCase`.

We also must verify that all user-defined types are globally consistent. There are two aspects to this test:

- If a record named `R1` is defined below `UD1` and also below `UD2` - then it must have the same fields definition.
- Recursive type definitions are possible. For example:

```
(L51
    (define-type Env
        (Empty-Env)
        (Extended-Env (var : string) (val : number) (tail : Env)))
    (define (env-length : (Env -> number))
        (lambda ((e : Env)) : number
            (type-case Env e
                (Empty-Env () 0)
                (Extended-Env (var val tail) (+ 1 (env-length tail))))))
    (define (e1 : Env) (make-Extended-Env "a" 1 (make-Empty-Env)))
    (env-length e1)
)
```

  Such a recursive definition is valid only if there exists a “base case” for the recursion - such as the `Empty-Env` record in this example.

Implement these checks in the function `checkUserDefinedTypes`.

Which functions should invoke each of these two functions? Add their invocations in the right places in the code.

Add test-cases to cover these definitions.

(3.2.5) Type Checking

You finally must implement the functions `typeofTypeCase` and `typeofDefineType` in file `src/L51-typecheck.ts` to implement the typing rules you defined above.

Implement also the function `typeofDefineType` (it is very simple - what should `define-type` return?).

Consult the test cases for the type checker with user defined types and `any` in `test\L5-typecheck.test.ts` and make sure they all pass.

Good Luck and Have Fun!