Principles of Programming Languages 212
Assignment 4

Responsible TA: Amit Portnoy

Submission Date: 6/June/2021

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 Mocha and Chai as dependencies 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 { expect } from "chai";
import { sum } from "../src/assignmentX";

describe("Assignment X", () => {
  it("sums two numbers", () => {
    expect(sum(1, 2)).to.equal(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 basic tests in the `test` directory, just to make sure you are on the right track during the assignment.

What to Submit

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

```
/
├── part1.pdf
├── part2
│   └── src
│       └── part2.ts
└── part3
    └── src/*
```

You will create the file `part1.pdf` and change the files `part2.ts`, and the files under `part3/src` in the places marked by the string `TODO`. Make sure to to include additional 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 as `part1.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) $\{f : [T1 \rightarrow T2], g : [T1 \rightarrow T2], a : T1\} \vdash (f\ (g\ a)) : T2$
   (b) $\{x : T1, y : T2, f : [T2 \rightarrow T1]\} \vdash (f\ y) : T1$
   (c) $\{f : [T1 \rightarrow T2]\} \vdash (lambda\ (x)\ (f\ x)) : [T1 \rightarrow T2]$
   (d) $\{f : [T1 * T2 \rightarrow T3]\} \vdash (lambda\ (x)\ (f\ x\ 100)) : [T1 \rightarrow T3]$
2. Perform type inference manually on the following expressions, using the Type Equations method. List all the steps of the procedure (12 pts):
   (a) `((lambda (x1) (+ x1 1)) 4)`
   (b) `((lambda (f1 x1) (f1 x1 1)) 4 +)`

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)

Write a function `makePromisedStore<K, V>` that returns an object that simulates a remote key-value store like a database or a web-service.

Since we are not really using a remote store, the actual data is going to be stored using a simple JavaScript Map. The object returned from `makePromisedStore` has the following type that delegates to the Map object:

```
type PromisedStore<K, V> = {
    get(key: K): Promise<V>,
    set(key: K, value: V): Promise<void>,
    delete(key: K): Promise<void>
}
```

For a successful 'get' the value matching the 'key' should be resolved. For successful 'set' and 'delete' the promise should resolve without any value (just call `resolve()`).

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___"
```

(b) (3 pts)

Write a function, `getAll<K, V>`, that accepts a `promisedStore<K, V>` 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

In this question you are requested *to use* the `async` keyword! (`Promise` is OK **only** if used as a type)

(a) (10 pts)

Write a function `asyncMemo` that takes another function that has a single parameter and returns a new function that caches any application of the original function using the `PromisedStore` from the previous question. Any subsequent application with the same parameter will then use the cached result.

That is, the **keys** stored in the `PromisedStore` are the parameter values that the original function was called with and the **values** in the `PromisedStore` are those values returned when running the original function with the parameter in the key.

Wrappers like `asyncMemo` can be used, for example, when the target function has some high cost (e.g., it solves a travelling sales men problem) *and* we want to maintain a persistent 'memo' in the (asynchronously accessed) local network, which will enable has to re-use that memo in different executions.

(b) (2 pts)

The type of `asyncMemo` is

```
type AsyncMemoType = <T, R>(f: (param: T) => R) => (param: T) => Promise<R>
```

Explain why the wrapped function returns `Promise<R>`.

Question 2.3

(10 pts)

Write the functions `lazyFilter` and `lazyMap` that take a parameter-less generator function and a filter or map callbacks, respectively (you can find the signatures in the template).

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

Question 2.4

(15 pts)

In this question you can choose whether to use async functions or promises.

Write an async function called `asyncWaterfallWithRetry`. The function expects an array of async functions as a parameter and runs the functions in sequence, where each function is called with the resolved return value of its predecessor. That is the first, function takes no parameters and any subsequent function has a single parameter that match the resolved return type of its predecessor. To make the typing easier (although not ideal) you can always except at least one input function and you can declare that input functions accept and return type `any` (except for the first function, which doesn't have a parameter).

If one of the functions in the list fails (i.e., rejects), then we want to wait 2 seconds using setTimeout and then try again (tip: create a new `Promise`, and send its `resolve` as a callback to the `setTimeout`) . We only retry two times and then re-throw the error if it still happens (that is, we will attempt at most three times to invoke each function). If a function fails three times, `asyncWaterfallWithRetry` will fail.

Part 3: Type Inference System

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

1. Complete the type inference code in L5 to support the missing AST expression types `program, define,` quoted literal expressions and `set!`.
2. Extend the L5 language and type inference code to support the `class` construct as introduced in Assignment 2.

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`.

(3.1) Support Type Inference with Program, Define, Quoted Literal Expressions and Set!

(12 pts)

Complement the code in `src/part3/src/L51-typeinference.ts` so that the type inference system can deal with the L5 AST nodes of type `define`, `program`, `literal` and `set!`.

In `part1.pdf`, write the typing rule for `define` and `set!` expressions:

```
Typing rule define:
  For every: type environment _Tenv,
             variable _x1
             expressions _e1 and
             type expressions _S1, _U1:

    If   _Tenv ... |- ...
    Then _Tenv |- (define _x1 _e1) : ...
```

Make sure that this typing rule allows type inference of recursive functions.

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

You must implement the functions `typeofDefine`, `typeofLit`, `typeofProgram` and `typeofSet` in file `src/part3/src/L51-typeinference.ts`.

(3.2) Define the Type of Primitives cons, car, cdr

(3 pts)

Extend the procedure `typeofPrim` in `src/part3/src/L51-typecheck.ts` so that the type inference system supports the primitives `cons`, `car` and `cdr`. Use the type expression for `PairTExp` that has been added in `src/part3/src/TExp51.ts`.

(3.3) Extend L5 with Class Types

(20 pts)

We introduced in Assignment 2 the `class` construct, with the following syntax:

```
<cexp> ::= ...
```

```
  | ( class : <TypeName>
      ( <varDecl>+ )
      ( <binding>+ ) ) /
    ClassExp(typeName:String,
             fields:VarDecl[],
             methods:Binding[]))
```

We introduce this construct in L5, and support type annotations in classes as in this example. To make type inference more consistent and support methods with parameters, we change slightly the syntax of class methods from what was done in Assignment 2: given a class value `c-value` (the return value of the class constructor), and a symbol corresponding to the name of a method `method`, the application expression `(c-value method)` always returns a closure. (In Assignment 2, this application was returning the application of the closure.)

```
(define pair
  (class : Tpair
    ((a : number)
     (b : number))
    ((first (lambda () : number a))
     (second (lambda () : number b))
     (scale (lambda (k) : pair (pair (* k a) (* k b))))
     (sum (lambda () : number (+ a b))))))

(define (p34 : Tpair) (pair 3 4))
(define f
  (lambda ((x : Tpair))
    (* ((x 'first)) ((x 'second)))))

(p34 'first)      ; --> #<procedure>
((p34 'first))    ; --> 3
((p34 'sum))      ; --> 7
((p34 'scale) 2)  ; --> #pair<6,8>
(f p34)           ; --> 12
```

The evaluation of the `class` expression returns a class constructor, which takes one parameter for each of the fields in the class definition. In the example above, the `pair` class constructor has type:

```
[number * number -> Tpair]
```

The evaluation of a class constructor returns a class value which has the type defined by the class. In the example above, the `pair` value constructor returns a value of type `Tpair`.

The definition of a `class` defines a new user defined compound type, which we call a `ClassTExp` and which is associated to a `TVar`. In our example we call it `Tpair`, which can be used in any place a Type Expression (`TExpr`) is expected (for example, it is used in the declaration of the `p34` variable).

Your mission is to extend the type inference system to support user defined `class` types. You will need to do the following:

1. Extend the parser (in file `src/part3/src/L51-ast.ts`) to support class constructs. This will be similar to what you saw in Assignment 2 but with the extension of type annotations where appropriate. You must implement the function `parseGoodClassExp`.
2. Add to the parser (in file `src/part3/src/L51-ast.ts`) a function

   ```
   const parsedToClassExps: (p: Parsed) => ClassExp[];
   ```

to collect the list of typed names in an AST, so that the type inference system can use the corresponding type definitions. This function must be used in `makeTEnvFromClasses` in file `src/part3/src/L51-typeinference.ts`. This function scans the AST, collects all user defined types, and inserts them in the `TEnv` that will be used initially when analyzing the AST to infer its type.

3. Read the code and learn how the type expression parser (in file `src/part3/src/TExp51.ts`) has been extended so that class types can be recognized as type expressions, and unparsed. (You do not have to write code about this, the implementation is provided.)

4. Extend the type inference system to support class expressions in each of their possible positions:

   (a) As operator in an application (e.g., `(p34 'sum)`). Applying a class value to anything which is not one of the defined method names is a type error. For example, `(p34 'x)` is a type error. This behavior is implemented in function `checkEqualType` in `L51-typeinference.ts`.

   (b) As parameter of an application (e.g., `(f p34)`).

   (c) As class value constructor (e.g., `(pair 1 2)`)

   (d) As class definition (e.g., `(define pair (class ...))`). The definitions of the methods in the class definition must be type checked, while taking into account the type of the fields of the class.

To this end, you must define typing rules that concern class types. In these typing rules, use the following notation:

```
// We extend the 2 rules seen in class for application
// with a 3rd rule for the case of class-values as operator:
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

// New case:
// An expression (class_value 'method) returns a closure
// whose type is defined in the class's type
Class-value Application and (n = 1):
    If   _Tenv |- _class_value : ClassTExp[{_mi, _Ui} i = 1..k],
         _Tenv |- _e1 = _mi
    Then _Tenv |- (_class_value _e1) : _Ui

// A class expression returns a class-value constructor
Typing rule ClassExp:
For every: type environment _Tenv,
           variables _x1, ..., _xn, n >= 0
           symbols _m1, ..., _mk, k >= 0
           procedure expressions _p1, ..., _pk and
           symbol _ct and
           type expressions _S1, ...,_Sn, _U1, ...,_Uk :

    If   _Tenv o {_x1:_S1, ..., _xn:_Sn } |- _pi:_Ui for all i = 1..k ,
    Then _Tenv |- (class : _ct ( _x1 ... _xn ) ((_m1 _p1) ... (_mk _pk))) :
                  [_S1 * ... * _Sn -> _ct]
                 _ct : ClassTExp(_ct, (_m1 : _p1) ... (_mk : _pk))
```

You must implement the function `typeofClass` in file `src/part3/src/L51-typeinference.ts` to implement the typing rule.

Good Luck and Have Fun!