Principles of Programming Languages 252
Home Assignment 3

Responsible Lecturer: Yuval Pinter
Responsible TA: Asif Eretz Kdosha

Submission Date: 08/06/2025

Preliminaries

Structure of a TypeScript Project

The template of every TypeScript assignment will contain two important files:

- `package.json` - lists the package 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 the modules listed in `package.json` 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 type-checks the code in this project. For all the assignments we will use 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 provided 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 global dependency for testing purposes (so no need to import it). 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 nice format.

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

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

describe("Assignment X", () => {
  it("sums two numbers", () => {
    expect(sum(1, 2)).toEqual(3);
  });
});
```

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

```
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
└── src
    ├── L5-typecheck.ts
    └── ... .ts
```

Make sure that when you extract the zip (using `unzip` on Linux), the result is flat, *i.e.*, not inside a folder (the file `part1.pdf` is in the root directory). 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*.

Question 1.1:

1. Find MGUs for the following pairs of type expressions, if such an MGU exists [3 points each]:
   (a) `[T1*[T1->T2]->N], [[T3->T4]*[T5->Number]->N]`
   (b) `[T1*[T1->T2]->N], [Number*[Symbol->T4]->N]`
   (c) `T1, T2`
   (d) `Boolean, Boolean`
2. Are the following typing statements true or false? Explain why (you might want to consult the guide published under "typing statements summary") [3 points each]:
   (a) `{f: [T2->T3], g: [T1->T2], a: Boolean} |- (f (g a)): T3`
   (b) `{f: [T2->T1], x: T1, y: T3} |- (f x): T1`

Question 1.2:

In the type inference implementation, we represent type variables (`TVar`) with a content field (which is a `Box` which contains a type expression value, or `#f` when empty). In this representation, we can have a `TVar` refer in its content to another `TVar` iteratively, leading to a chain of `TVars`. Write a **program** which, when we pass it to the type inference algorithm, creates a chain of length five, i.e., `Tvar1->Tvar2->Tvar3->Tvar4->Tvar5`. [7 points]

This is a theory question, submit as text in the pdf, not as a code file.

Part 2: Type Checker: Support `define` and `program` Expressions

Question 2.1: `define`

In L5, add support for `define` expressions in the type checker. For example, the following code should be typed `void`:

```
(define (myvar : number) 5)
```

The following code should raise a type error:

```
(define (myvar : number) (lambda (x y) (+ x y)))
```

Guidelines

Think what the typing rule for `define` expressions should be, and complete the function `typeofDefine` in the file `L5-typecheck.ts`. [10 points]

Question 2.2: `L5` Program

Add support for checking the type of an entire program, which is a sequence of expressions wrapped in a boundary:

```
(L5 <exp>+)
```

Guidelines

Remember that the type environment needs to be updated after each `define` statement in order to enable lookup of variables whose `VarDecl` in the `define` expression included a type. We had the same issue when we implemented the interpreter variable environment for `L1`, consult that implementation.

Complete the function `typeofDefine` in `L5-typecheck.ts`, and any other code you deem necessary. [25 points]

Part 3: Type Checker: Support the `Pair(T1,T2)` Compound Type

Add support for the `Pair` type in the type checker code. Notice that the type expression for pairs is denoted in the language as following:

```
(Pair T1 T2)
```

Follow the following steps: [20 points for 1–5, 20 points for 6]

1. Add `Pair` as part of the `TExp` type language (i.e., modify the file `TExp.ts`):
   (a) Modify the `TExp` type itself;
   (b) Modify the parser, specifically the function `parseCompoundTExp`.
2. Modify `unparseTExp` to support pairs;
3. Add primitives `cons`, `car`, `cdr` to the type checker's functions `typeofPrim` (in `L5-typecheck.ts`) and `isPrimitiveOp` (in `L5-ast.ts`);
4. Modify `checkNoOccurrence` to support pairs;
5. Add the `quote` special form as defined in L3, as well as the shorthand `'` equivalent;
6. Extend the type language implementation to support comparison of type expressions including pairs.

A working code example with pairs may look like the following (see more in the tests):

```
(lambda ([a : number] [b : number]): (Pair number number)   (cons a b))
```

Good Luck and Have Fun!