Homework 10 in COMS 342: Principles of Programming Languages
Spring 2018
Instructor: Dr. Hridesh Rajan

This homework focuses on types and type checking in programming language design.

This homework is due 04/22/2018 11:59 pm after which it will be considered late.
Homework submissions after 04/22/2018 11:59 pm will incur a 100% penalty.

Instructions:

1. Attempt All questions. For these questions, you will need to read textbook Chapter 10 that is available from Pages->Textbook.

2. Clone a fresh copy of the Typelang interpreter for all programming-related questions in this homework. Make sure that this clone is **up-to-date**.

3. Clone Typelang by using our instructions available at the Downloading and Installing the Interpreter Framework. However, instead of the Arithlang link in step 1 use the Typelang link that is https://github.com/hridesh/typelang.

4. For all questions in this homework, you are asked to write your answers either on paper and scan or using a word processing program, convert to a PDF file and upload your file as <lastname>hw10.pdf.

In this homework, you will learn about types and typechecking by writing well-typed and ill-typed programs and by trying out type checking rules for a number of expressions.

This homework needs an in-depth understanding of types and typechecking rules and hence we highly recommend reading the entire chapter 10 before attempting this homework.

Interpreter for Typelang is significantly different compared to previous interpreters, since support for checking types have been added. We are enumerating some major changes in following section but it is recommended that you review the code for Typelang for better understanding.

1. Env in Typelang is generic compared to previous interpreters.
2. Two new files Checker.java and Type.java have been added.
3. Type.java defines all the valid types of Typelang.
4. Checker.java defines type checking semantics of all expressions.
5. Typelang.g has changed to add type information in expressions. Please review the changes in file to understand the syntax.
6. Interpreter.java has been changed to add type checking phase before evaluation of Typelang programs.
7. Finally, Checker.java, if you run it directly, just checks the type of an expression.

(20 points)
Q1
The Typelang interpreter also provides a facility to just check the type of an expression, without evaluating it. This could be done by running the file “Checker.java” as the main program.

Check the type of the following programs. Each program is worth 2 points, and 0 points will be awarded for incorrect answers.

a. (lambda (x:num) (lambda (y:num) #t))

Write down the type and explain the type in a sentence.

b. (lambda (x:num) (lambda (y:num) (if x #t #f)))
Write down the type and explain the type in a sentence.

c. (lambda (x:num) (lambda (y:num) (if (> x y) #t y)))
Write down the type and explain the type in a sentence.

d. (lambda (x:num) (lambda (y:num) (if (> x y) x y)))
Write down the type and explain the type in a sentence.

e. (if (> 1 0) (lambda (x:num) (lambda (y:num) #t)) (lambda (x:num) (lambda (y:num) x)))
Write down the type and explain the type in a sentence.

(5 points)
f. Reflection: In the lecture, we talk about types as contracts between producer and consumers of values. In the case of an if expression, what are the producers and consumers?
Explain using the following form.
producer: write a sentence
consumer (s): write a sentence

(5 points)
g. Reflection: Based on the examples that you did above, how does the design of an if expression avoids surprising the consumers of values? Explain using 1-2 sentence.

(100 points)
Q2.

Write down type derivations for the following five expressions as shown in the lectures.
We do expect you to follow the type checking rules given in the textbook precisely.

(20 points)
a.

```
(let
        ((x : num 2))
        (let
                ((y : num 5))
                (+ x y)
        )
)
```

For this expression, since its long, consider using let0 to refer to the entire inner let expression,
i.e. (let ((y : num 5)) (+ x y)), until you need to expand it.

(20 points)
b.

```
(let
        ((x : num 2))
        (let
                ((z : (num -> num) (lambda (y : num) (+ x y))))
```

```
            (z 5)
        )
)
```

For this expression, since its long, consider using let0 to refer to the entire inner let expression, i.e. (let ((z : (num -> num) (lambda (y : num) (+ x y)))) (z 5)), until you need to expand it.

(20 points)
c.

```
(
        (lambda
            (x : num y : num)
            (+ x (+ y x))
        )
        1 2
)
```

(20 points)
d

```
(let
        ((f : (num -> (num -> num))
                    (lambda (x : num)
                                (lambda (y : num)
                                            (+ x y)
                                )
                    )
        ))
f)
```

For this expression, since its long, consider using lam0 to refer to the entire inner lambda expression, i.e. (lambda (x : num) (lambda (y : num) (+ x y))), until you need to expand it.

(20 points)
e

```
(let
        ((a : Ref num (ref : num 2)))
        (set! a (deref a))
)
```

(10 points)
Q3. The file Type.java contains implementation of all types. Which types are atomic types and which types are compound types? Write classnames as your answer.

Class Name	Atomic or Compound
..	..

Add as many rows as necessary in this table.

(100 points)
Q4. In formal discussions about types, we do not explicitly discuss the error value produced by type checking rules, but the implementation of type checking rules produces a type ErrorT that encapsulates a message to the programmer.

Write down all 20 remaining conditions under which rules in Checker.java produces a type ErrorT as follows. In the examples below, we are succinct, but do feel free to write slightly longer conditions. By slightly longer we mean 25 words or less.

The point allocation is as follows.
* 5 points are allocated for each case of ErrorT
  + 1 point for identifying the conditions correctly
  + 4 points for giving an appropriate (short) example
    (award 0/5 points for syntactically incorrect example)

Term	Condition, Example, and Type Error
Program	When the declared type of the define decl doesn't match with expression's type. Example: (define x:num #t) Type error: Expected num found bool in (define x #t)
..	..
..	..
ErrorExp	When an error expression is encountered. Example: a programmer cannot write this expression, its for internal purposes only. Type error: not applicable.