Programming Language Concepts

CSCI-344 **Programming 5**
Term 20161 **October 13, 2016**

Type Systems

Due: October 26, 2016

1 Introduction

In this programming assignment, you will extend the type checker for Typed Impcore (relatively easy) and build the type checker for Typed μScheme (relatively hard).

This programming assignment must be completed as a pair programming assignment; see [WK00] for useful guidelines on pair programming.

2 Description

Complete the following exercises. See Requirements and Submissions for important restrictions.

A. (15pts) Complete Exercise 2 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 449). The exercise asks you to complete the type checker for Typed Impcore by implementing the rules for array operations. You will modify the provided `timpcore.sml` interpreter.

   You need only complete the four cases of the `ty` function (within the `typeof` function) that currently raise `LeftAsExercise` exceptions (lines 1235–1238). You should not need to modify any other part of the interpreter (but you will need to read other parts of the interpreter, particularly the `datatype ty` definition (line 1002), which gives the SML representation of Typed Impcore types). A reasonable solution is 20 to 50 lines of SML, depending on the quality of the error messages (but high quality error messages are not required for this assignment).

   Tips, Advice, and Hints:

   - Although most of the existing cases of the `ty` function use `eqType` to compare the type of a subexpression with the required type of the subexpression, this approach will not work if the required type of the subexpression is an array type and you do not know the type of the elements of the array. Instead, you will need to use `case` expressions to match the type of a subexpression against `ARRAYTY t` in order both check that the type of the subexpression is an array type and to extract the type of the elements.

B. (75pts) Complete Exercises 22 and 23 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (pp. 455–458). The exercises asks you to implement renaming to avoid variable capture and to write the type checker for Typed μScheme. You will modify the provided `tuscheme.sml` interpreter.

   You need to complete the `renameForallAvoiding` function that currently raises a `LeftAsExercise` exception, many cases of the `ty` function (within the `typeof` function) that currently raise `LeftAsExercise` exceptions and many cases of the `elabdef` function that currently raise `LeftAsExercise` exceptions. You should not need to modify any other part of the interpreter (but you will need to read other parts of the interpreter, particularly the `datatype tyex` definition (lines 1022–1026), which gives the SML representation of Typed μScheme type-level expressions, and the `datatype exp`, `datatype value`, and `datatype def` definitions (lines 1168–1178, lines 1181–1187, and lines 1192–1195), which give the SML representation of Typed μScheme expressions, values, and definitions). A reasonable solution is about 120 lines of SML, depending on the quality of the error messages (but high quality error messages are not required for this assignment).

   Tips, Advice, and Hints:

   - Don't copy and paste the Typed Impcore `typeof` and `elabdef` functions into Typed μScheme; there are too many details to change to make it worthwhile. Use the structure as a guide, but start from scratch.
   - Compile early and compile often!

- Test early and test often! Be sure to test with both examples that should type check and examples that should *not* type check.
- Build the type checker one piece at a time:
  - First, write code to type check `LITERAL/NUM`, `LITERAL/BOOL`, and `LITERAL/SYM`.
  - Next, type check `VAL`. (Because `EXP` is just syntactic sugar for `VAL`, this will also type check `EXP`).
    - At this point, "programs" like `(val ans 1)` and `(val ans #t)` should type check.
  - Next, type check `IF`.
    - At this point, "programs" like `(val ans (if #t -1 1))` and `(val ans (if #f -1 1))` should type check.
  - Next, type check `VAR`.
    - At this point, "programs" like `(val x #t) (val y -1) (val z 1) (val ans (if x y z))` should type check.
  - Next, type check `APPLY`.
    - At this point, you should be able to test arithmetic and comparison primitives.
  - Next, type check `LETX/LET`. Be careful with the environments; be sure to type check sub-expressions with the right environment.
  - Next, type check `LAMDBA` (which is quite similar to `LETX/LET`). To create a function type, use the provided `funty` function.
  - Next, type check `APPLY`. Pattern match against the `FUNC` constructor of the `tyex` datatype to simultaneously determine if a type is a function type and to extract the argument types and the result type of the function type.
  - Next, type check `VALREC` and `DEFINE` (remembering that `DEFINE` is syntactic sugar for `VALREC`). Don't forget the necessary side condition that is described on page 282.
  - Next, type check `SET`, `WHILEX`, and `BEGIN`.
  - Next, type check `LITERAL/NIL` and `LITERAL/PAIR`. Remember that the empty list literal is polymorphic, but non-empty list literals are monomorphic.
  - Next, type check `LETX/LETSTAR` (by treating it as syntactic sugar for nested `LET`s).
  - Next, type check `TYLAMBDA`.
  - Finally, implement `renameForallAvoiding` and type check `TYAPPLY`.
- Reread Section 6.6 of *Programming Languages: Build, Prove, and Compare*, paying special attention to the useful functions and representations that are already implemented in the interpreter.
- Do not use SML's polymorphic equality (the `=` operator) to compare types. SML's polymorphic equality compares values for *structural equality*. Use the provided `eqType` function to compare types. More details about the `eqType` function may be found in Section 6.6.6 of *Programming Languages: Build, Prove, and Compare*.

C. (bonus 5pts) Complete Exercise 32 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 461). The exercise asks you to demonstrate that the provided `appearsUnprotectedIn` function is not conservative enough. Write your program in a file named `problemC.scm`.

2.1 Interpreter Source Code

Source code for the interpreters is available on the CS Department file system at:

`/usr/local/pub/mtf/plc/src/bare/prog05-code`

and packaged as an archive at:

`/usr/local/pub/mtf/plc/src/bare/prog05-code.tar`

Note that this source code contains just the SML code from the textbook, with simple comments identifying page numbers. There is a `Makefile` for building the interpreters and running tests.

Copy the interpreter source code to a local directory and make modifications to your local copy; for example, executing

```
$ tar xvf /usr/local/pub/mtf/plc/src/bare/prog05-code.tar
```

will copy the interpreter source code to a new local directory named `prog05-code`.

2.2 Tests

The `prog05-code/tests` directory contains a number of tests for the Typed Impcore and Typed μScheme type checkers, as well as scripts for running the tests.

For example, executing

```
$ make timpcore-tests.out
```

from your `prog05-code` directory will build your Typed Impcore interpreter and run the Typed Impcore tests; the test results will be echoed to the terminal and also saved in the `timpcore-tests.out` file.

For each `test.imp` file, if the test has no type errors, then there is a `test.soln.tychk` file containing the output of the reference interpreter (the name and/or value defined by each declaration and the type of the defined name and/or value; see *Programming Languages: Build, Prove, and Compare* (p. 400)). If the test has type errors, then there is a `test.soln.tyerr` file containing the output of the reference interpreter, concluding with the type error message reported by the reference solution type checker.

Similarly, executing

```
$ make tuscheme-tests.out
```

from your `prog05-code` directory will build your Typed μScheme interpreter and run the Typed μScheme tests; the test results will be echoed to the terminal and also saved in the `tuscheme-tests.out` file.

For each `test.scm` file, if the test has no type errors, then there is a `test.soln.tychk` file containing the output of the reference interpreter (the name and/or value defined by each declaration and the type of the defined name and/or value; see *Programming Languages: Build, Prove, and Compare* (p. 426)). If the test has type errors, then there is a `test.soln.tyerr` file containing the output of the reference interpreter, concluding with the type error message reported by the reference solution type checker.

2.3 Reference Interpreters

Reference Typed Impcore and Typed μScheme interpreters are available on the CS Department Linux systems (e.g., `glados.cs.rit.edu` and `queeg.cs.rit.edu` and ICLs 1 and 2) at:

`/usr/local/pub/mtf/plc/bin/timpcore`

and

`/usr/local/pub/mtf/plc/bin/tuscheme`

Use the reference interpreters to develop tests and to check your interpreters.

3 Requirements and Submission

Always use pattern matching to inspect and deconstruct values. The use of `null`, `hd`, `tl`, `#1`, `#2`, etc., will result in zero credit for the assignment.

Your modified interpreters must be a valid Standard ML program. In particular, they must compile with Moscow ML or MLton without any error messages. If your submission produces error messages (e.g., syntax errors or type errors), then your submission will not be tested and will result in zero credit for the assignment.

Write a `README.txt` file. Your `README.txt` file should be formatted as follows:

```
Names:
Time spent on assignment:
Additional Collaborators:
```

Submit `README.txt`, `timpcore.sml`, `tuscheme.sml`, and (optionally) `problemC.scm` to the `Programming 05` Dropbox on MyCourses by the due date. Only one submission is required per group.

References

[WK00] Laurie A. Williams and Robert R. Kessler. All I Really Need to Know About Pair Programming I Learned in Kindergarten. *Communications of the ACM*, 43(5):108–114, May 2000.

Document History

October 13, 2016
Original version