Programming Language Concepts

CSCI-344 **Programming 5**
Term 20235

Type Systems

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 problems. See Requirements and Submissions for important restrictions.

A. (20pts) Complete Exercise 18 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 394). 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 1591–1594). 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 (lines 1321–1325), which gives the SML representation of Typed Impcore types). A reasonable solution is 30 to 70 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.
   - Review the solution to the problem of adding lists to Typed Impcore from Recitation 06.

B. (145pts) Complete Exercises 19 and 28 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (pp. 395–396 and pp. 397–398). The exercises asks you to write the type checker for Typed μScheme and to implement renaming of type variables to avoid capture. You will modify the provided `tuscheme.sml` interpreter.

   You need to complete the `renameForallAvoiding` function that currently raises a `LeftAsExercise` exception, all cases of the `ty` function (within the `typeof` function) that currently raise `LeftAsExercise` exceptions and all cases of the `typdef` 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 1342–1347), which gives the SML representation of Typed μScheme type-level expressions, and the `datatype exp`, `datatype value`, and `datatype def` definitions (lines 1518–1530, lines 1535–1542, and lines 1547–1551), which give the SML representation of Typed μScheme expressions, values, and definitions). A reasonable solution is 150 to 200 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/BOOLV`, and `LITERAL/SYM`.
    - At this point, "programs" like `1` and `#t` still will not type check, because `VAL` and `EXP` definitions do not yet type check.
  - 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 `IFX`.
    - 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 `SET`, `BEGIN`, and `WHILE`.
  - Next, type check `LETX/LET`. Be careful with the environments; be sure to type check sub-expressions with the right environment.
  - Next, type check `LETX/LETSTAR` (by treating it as syntactic sugar for nested `LET`s).
  - Next, type check `LAMDBA` (which is quite similar to `LETX/LET`). To create a function type, use the `FUNTY` constructor of the `tyex` datatype.
  - Next, type check `APPLY`. Pattern match against the `FUNTY` 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`).
  - Next, type check `LETRECX`.
  - 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 `TYLAMBDA`. Don't forget to check the restriction that a `type-lambda` may not abstract over a type variable that's free in the type environment.
  - Finally, implement `renameForallAvoiding` and type check `TYAPPLY`.

    The specification for `renameForallAvoiding` is

    > `renameForallAvoiding`$([\alpha_1, \ldots, \alpha_n], \tau, C)$ must return a type $\forall\beta_1, \ldots, \beta_n.\ \tau'$ with these properties:
    >
    > $$\forall\beta_1, \ldots, \beta_n.\ \tau' \equiv \forall\alpha_1, \ldots, \alpha_n.\ \tau$$
    >
    > $$\{\beta_1, \ldots, \beta_n\} \cap C = \emptyset$$

    Both τ and τ' are type expressions, corresponding to the SML `tyex` type (defined on p. 357 and lines 1342–1347). When `renameForallAvoiding`$([\alpha_1, \ldots, \alpha_n], \tau, C)$ returns $\forall\beta_1, \ldots, \beta_n.\ \tau'$, both τ and τ' will have the same "shape" (e.g., if τ was a function type (`FUNTY`), then τ' will also be a function type). The difference between τ and τ' is that if τ uses some type variable α_i that is in the set C (the set of type variables that we want to "avoid"), then τ' replaces that α_i by some type variable β_i that is chosen to not conflict with any of the type variables in C (and is different from all of the other βs).

For example:

```
renameForallAvoiding(["'a","'b","'c"],
                     FUNTY([TYVAR "'a",TYVAR "'b",TYVAR "'d"],TYVAR "'e"),
                     ["'a","'c","'e"])
```

where (using Typed μScheme syntax for readability)

$$[\alpha_1, \alpha_2, \alpha_3] = [\texttt{'a}, \texttt{'b}, \texttt{'e}]$$
$$\tau = \texttt{('a 'b 'd -> 'e)}$$
$$C = [\texttt{'a}, \texttt{'c}, \texttt{'e}]$$

might return

```
FORALL(["'f","'b","'g"],FUNTY([TYVAR "'f",TYVAR "'b",TYVAR "'d"],TYVAR "'g"))
```

where (using Typed μScheme syntax for readability)

$$[\beta_1, \beta_2, \beta_3] = [\texttt{'f}, \texttt{'b}, \texttt{'g}]$$
$$\tau' = \texttt{('f 'b 'd -> 'g)}$$

Note that `'a` was replaced by `'f` and `'e` was replaced by `'g`, but `'b` was left alone.

We can't replace `'a`

* by `'b`, because that would violate both "is not free in τ" (because ftv(`('a 'b 'd -> 'e)`) = {`'a`, `'b`, `'d`, `'e`}) and "is different from every α_i".
* by `'c`, because that would violate "must be a new variable that does not appear in C".
* by `'d`, because that would that would violate "is not free in τ".
* by `'e`, because that would violate all of "must be a new variable that does not appear in C", "is not free in τ", and "is different from every α_i".

So, the first type variable that satisfies all of the conditions is `'f`.

On the other hand, we can leave `'b` alone, because `'b` does not appear in C.

We can't replace `'e`

* by `'a`, because that would violate all of "must be a new variable that does not appear in C", "is not free in τ", and "is different from every α_i".
* by `'b`, because that would violate both "is not free in τ" and "is different from every α_i".
* by `'c`, because that would violate "must be a new variable that does not appear in C".
* by `'d`, because that would that would violate "is not free in τ".
* by `'f`, because that would violate the implicit assumption that every β_i is different (and `'f` was chosen for β_1).

So, the first type variable that satisfies all of the conditions is `'g`.

Hint: The code for `renameForallAvoiding` is somewhat simpler than might be implied by the above. The essence is that for each α_i that must be replaced, construct an appropriate set of type variables to avoid and call `freshName`.

Hint: There is a reason that `renameForallAvoiding` is defined within the body of `tysubst`, rather than before the definition of `tysubst`. What function is in scope when `renameForallAvoiding` is defined within the body of `tysubst` that would not be in scope if `renameForallAvoiding` was defined before the definition of `tysubst`?

- 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 20pts) Complete Exercises 11 and 12 of Chapter 6 from *Programming Languages: Build, Prove, and Compare* (p. 390). The exercises asks you to implement the `pair` and `sum` type constructors and the polymorphic functions `pair`, `fst`, `snd`, `left`, `right`, and `either` *without* changing the abstract syntax, values, type checker, or evaluator of Typed μScheme. You will modify the provided `tuscheme.sml` interpreter.

You need only add to the `kinds` (Δ), `types` (Λ), and `values` (ρ) components of the `val primBasis` definition (lines 2815–2957).

These extensions are similar in spirit to the technique described in the last problem of Recitation 07. In fact, the Typed μScheme interpreter has already been extended with the `btree` type constructor and the polymorphic functions `leaf`, `node`, `leaf?`, `node?`, `node-l`, `node-x`, `node-r`.

2.1 Interpreter Source Code

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

`/usr/local/pub/mtf/plc/programming/prog05-typesys`

and packaged as an archive at:

`/usr/local/pub/mtf/plc/programming/prog05-typesys.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/programming/prog05-typesys.tar
```

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

2.2 Tests

The `prog05-typesys/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-typesys` 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). 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-typesys` 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). 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`, and `tuscheme.sml` 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. `http://doi.acm.org/10.1145/332833.332848`.