COMP 105 Homework: Type Systems

Due **Thursday**, March 14, at 11:59 PM.

The purpose of this assignment is to help you learn about type systems.

Setup and Guidelines

Make a clone of the book code:

```
  git clone linux.cs.tufts.edu:/comp/105/book-code
```

You will need copies of `book-code/bare/tuscheme/tuscheme.sml` and `book-code/bare/timpcore/timpcore.sml`.
The code for Typed μScheme has changed, so **if you have an old clone, you will need to run `git pull`**.

As in the ML homework, use function definition by pattern matching. In particular, do not use the functions `null`, `hd`, and `tl`.

Two problems to do by yourself

1. Do Exercise 1 on page 274 of Ramsey: add lists to Typed Impcore. The exercises requires you to **design new syntax** and to **write type rules** for lists. I expect your type rules to be *deterministic*. Turn in this exercise in PDF file `lists.pdf`.

I recommend that you **do Exercise 2 first** (with your partner). It will give you more of a feel for monomorphic type systems.

Hint: Exercise 1 is more difficult than it first appears. I encourage you to scrutinize the lecture notes for similar cases, and to remember that you have to be able to type check every *expression* at compile time.

Here are some things to watch out for:

- It's easy to conflate syntax, types, and values. In this respect, doing theory is significantly harder than doing implementation, because there's no friendly ML compiler to tell you that you have a type clash among `exp`, `tyex`, and `value`.
- It's especially easy to get confused about `cons`. You need to create a **new syntax** for `cons`. This syntax needs to be *different* from the `PAIR` constructor that is what `cons` *evaluates* to.
- Dealing with the empty list presents a real challenge. Typed Impcore is monomorphic, which implies that any given piece of syntax has at most one type. But you want to allow for empty lists of *different* types. The easy way out is to design your syntax so that you have many different expressions, of different types, that all evaluate to empty lists.
- A good mental test case is that if `ns` is a list of integers, then `(car (reverse ns))` is an expression that probably should have a type.
- You might want to see what happens to an ML program when you try to type-check operations on empty and nonempty lists. For this exercise to be helpful, you *really* have to understand the phase distinction between a type-checking error and a run-time error.
- It's easy to write a type system that requires nondeterminism to compute the type of any term involving the empty list. But the problem requires a *deterministic* type system, because deterministic type systems are much easier to implement. You might consider whether similar problems arise with other kinds of empty data structures or whether lists are somehow unique.

11. Do Exercise 11 on page 275 of Ramsey: write `exists?` and `all?` in Typed uScheme. Turn in this exercise in file `11.scm`.

Four problems to do with a partner

2. Do Exercise 2 on page 274 of Ramsey: finish the type checker for Typed Impcore by implementing the rules for arrays. Turn in this exercise in file `timpcore.sml`.
My solution to this problem is 21 lines of ML.

15. Do Exercise 15 on page 276 of Ramsey: extend Typed uScheme with a type constructor for queues and with primitives that operate on queues. As it says in the exercise, **do not change** the abstract syntax, the values, the `eval` function, or the type checker. If you change any of these parts of the interpreter, your solution will earn **No Credit**.

I've made one change to the exercise: instead of implementing `get`, implement the two functions `get-first` and `get-rest`. This change will save you from having to deal with pair types. The change is reflected in two updated pages from the book.

I recommend that you represent each queue as a list. If you do this, you will be able to use the following primitive implementation of `put`:

```
let fun put (x, NIL)           = PAIR (x, NIL)
      | put (x, PAIR (y, ys)) = PAIR (y, put (x, ys))
      | put (x, _)            = raise BugInTypeChecking "non-queue passed to put"
in  put
end
```

Turn in the code for this exercise in file `tuscheme.sml`, which should also include your solution to Exercise 13 below. Please include the answers to parts (a) and(b) in your README file.

Hint: because `empty-queue` is *not* a function, you will have to modify the `initialEnvs` function on page 270b. The updated pages from the book show two places you will update: the *<primitive functions for Typed μScheme ::>* and the *<primitives that aren't functions, for Typed μScheme ::>*.
My solution to this problem, including the implementation of `put` above, is under 20 lines of ML.

13. Do Exercise 13 on page 275 of Ramsey: write a type checker for Typed uScheme. Turn in this exercise in file `tuscheme.sml`, which should also include your solution to Exercise 15 above. Don't worry about the quality of your error messages, but do remember that your code must compile *without errors or warnings*.
My solution to this problem is about 120 lines of ML. It is very similar to the type checker for Typed Impcore that appears in the book. If I had given worse error messages, it could have been a little shorter.

T. Create three test cases for Typed uScheme type checkers. In file `type-tests.scm`, please put three `val` bindings to names `e1`, `e2`, and `e3`. (A `val-rec` binding is also acceptable.) **After** each binding, put in a comment the words "type is" followed by the type you expect the name to have. If you expect a type error, instead put a comment saying "type error". Here is an example (with more than three bindings):

```
(val e1 cons)
; type is (forall ('a) (function ('a (list 'a)) (list 'a)))

(val e2 (@ car int))
; type is (function ((list int)) int)

(val e3 (type-lambda ('a) (lambda (('a x)) x)))
; type is (forall ('a) (function ('a) 'a))

(val e4 (+ 1 #t)) ; extra example
; type error

(val e5 (lambda ((int x)) (cons x x))) ; another extra example
; type error
```

If you submit more than three bindings, we will use the *first* three.
Each binding must be completely independent of all the others. In particular, you cannot use values declared in earlier

bindings as part of your later bindings.

How to build a type checker

For your type checker, it would be a grave error to copy and paste the Typed Impcore code into Typed uScheme. You are much better off simply adding a brand new type checker to the `tuscheme.sml` interpreter. Use the techniques presented in class, and **start small**.

The only really viable strategy for building the type checker is one piece at a time. **Writing the whole type checker before running any of it will make you miserable.** Instead, start with small pieces similar to what we'll do in class:

1. It's OK to write out a complete case analysis of the syntax, but have every case raise the `LeftAsExercise` exception. This trick will start to get you a useful scaffolding, in which you can gradually replace each exception with real code. And of course **you'll test each time**.
2. You can begin by type-checking literal numbers and Booleans.
3. Add IF-expressions as done in class.
4. Implement the VAL rule for definitions, and maybe also the EXP rule. Now you can test a few IF-expressions with different types, but you'll need to disable the initial basis as shown below.
5. Implement the rule for function application. You should be able to test all the arithmetic and comparisons from class.
6. Implement LET binding. The Scheme version is slightly more general than we covered in class. Be careful with your contexts. Implement VAR.
7. Once you've got LET working, LAMBDA should be quite similar. To create a function type, use the `funtype` function in the book.
8. Knock off SET, WHILE, BEGIN.
9. Because of the representation of types, function application is a bit tricky. Study the `funtype` function and make sure you understand how to pattern match against its representation.
10. There are a couple of different ways to handle LET-STAR. As usual, the simplest way is to treat it as syntactic sugar for nested LETs.
11. Knock off the definition forms VALREC and DEFINE. (Remember that DEFINE is syntactic sugar for VALREC.)
12. Save TYAPPLY and TYLAMBDA for after the last class lecture on the topic. (Those are the *only* parts that have to wait until the last lecture; you can have your entire type checker, except for those two constructs, finished before the last class.)

Finally, don't overlook section 6.6.4 in the textbook. It is only a few pages, but it is chock full of useful functions and representations that are already implemented for you.

Let me suggest that you replace the line in the source code

```
val basis   = (* ML representation of initial basis *)
```

with

```
val basis_included = false
val basis = if not basis_included then [] else
```

With luck this will enable you to test things.

Before submitting, **turn the basis back on**.

Other advice

Here's some generic advice for writing any of the type-checking code, but especially the queues:

1. Compile early.
2. Compile insanely often.

3. Use an editor that jumps straight to the location of the error.
4. Come up with examples in Typed uScheme.
5. Figure out how those examples are *represented* in ML.
6. Keep in mind the distinction between the term language (values of queue type, values of function type, values of list type) and the type language (queue types, function types, list types).
7. If you're talking about a thing in the term language, you should be able to give its type.
8. If you're talking about a thing in the type language, you should be able to give its kind.

Avoid common mistakes

Here are some common mistakes:

- In Exercise 1, it's a relatively common mistake to try to create a type system that prevents programmers from applying `car` or `cdr` to the empty list. Don't do this! It can be done, but by the standards of COMP 105, such type systems are insanely complicated. As in ML, taking `car` or `cdr` of the empty list should be a well-typed term that causes an error at run time.
- In Exercise 1, it's quite common to write a nondeterministic type system by accident. The rules, typing context, and syntax have to work together to determine the type of every expression. But you're free to choose whatever rules, context, and syntax you want.
- In Exercise 1, it's inexplicably common to forget to write a typing rule for the construct that tests to see if a list is empty.
- There are already interpreters on your PATH with the same name as the interpreters you are working on. So remember to get the version from your current working directory, as in

  ```
  ledit ./timpcore
  ```

  Just plain `timpcore` will get the system version.
- **ML equality is broken!** The = sign gives equality of *representation*, which may or may not be what you want. For example, in Typed uScheme, you must use the `eqType` function to see if two types are equal. If you use built-in equality, **you will get wrong answers**.
- The `val-rec` form requires an extra side condition; in

  ```
  (val-rec x e)
  ```

  it is necessary to be sure that `e` can be evaluated without evaluating `x`. Many students forget this side condition, which can be implemented very easily with the help of the function `appearsUnprotectedIn`, which should be listed in the ``code index" of your book.
- It's a common mistake to call `ListPair.foldr` and `ListPair.foldl` when what was really meant was `ListPair.foldrEq` or `ListPair.foldlEq`.

A few words about difficulty and time

In Exercise 1 on page 274, I am asking you to create new type rules on your own. Many students find this exercise easy, but many find it very difficult. My sympathy is with the difficult camp; you haven't had much practice *creating* new rules on your own.

Exercise 11, writing `exists?` and `all?` in Typed μScheme, requires that you really understand *instantiation* of polymorphic values. Once you get that, the problem is not at all difficult, but the type checker is very, very persnickety. A little of this kind of programming goes a long way.

Exercise 2, type-checking arrays in Typed Impcore, is probably the easiest exercise on this homework. You need to be able to duplicate the kind of reasoning and programming that we did in class for the language of expressions with LET and VAR.

Exercise 15, adding queues to Typed μScheme, requires you to understand how primitive type constructors and values are added to the initial basis. And it requires you to write *ML code* that manipulates *μScheme representations*. Study the way the existing primitives are implemented!

Exercise 13, the full type checker for Typed μScheme, presents two kinds of difficulty:

- You really have to understand the connection between typing judgments, typing rules, and code.
- You have to understand a moderately sophisticated ML program (the interpreter) and then build a relatively big and independent extension of it.

For the first item, we'll talk a lot in class about the concepts and the connection between type theory and type checking. For the second item, it's not so difficult **provided** you remember what you've learned about building big software: don't write it all in one go. Instead, start with a tiny language and grow it very slowly, testing at each step.

What to submit for your joint work with your partner

For your joint work with your partner, run `submit105-typesys-pair` from a directory that contains the following files: `timpcore.sml`, `tuscheme.sml`, and `type-tests.scm`. In addition to your code, please provide a short `README` file which describes, at a high level, the design and implementation of your solutions.

What to submit for your individual work

Provide another `README` file containing the following information:

1. Your name
2. The names of any collaborators
3. The number of hours you spent on the assignment

When you are ready, run `submit105-typesys-solo` to submit your work, which should include `README`, `lists.pdf` and `11.scm`. All files are mandatory.

How your work will be evaluated

Acute observers will have noticed that when it comes to writing detailed criteria for evaluation, I am falling behind. I am now very behind. But watch this space.

We have the same new handlers as in Typed Impcore.

270a ⟨*more read-eval-print handlers* 270a⟩≡ (269b 220b)
```
  | TypeError         msg => continue ("type error: " ^ msg)
  | BugInTypeChecking msg => continue ("bug in type checking: " ^ msg)
```

Initializing the interpreter

To put everything together into a working interpreter, we need an initial kind environment as well as a type environment and a value environment.

270b ⟨*initialization for Typed μScheme* 270b⟩≡ (271a) 270c▷
```
  val initialEnvs =
    let fun addPrim ((name, prim, funty), (types, values)) =
          ( bind (name, funty, types)
          , bind (name, ref (PRIMITIVE prim), values)
          )
        val (types, values) = foldl addPrim (emptyEnv, emptyEnv)
                                (⟨primitive functions for Typed μScheme :: 676b⟩ nil)
        fun addVal ((name, v, ty), (types, values)) =
          ( bind (name, ty, types)
          , bind (name, ref v, values)
          )
        val (types, values) = foldl addVal (types, values)
                                (⟨primitives that aren't functions, for Typed μScheme :: (automatically generated)⟩ nil)
        fun addKind ((name, kind), kinds) = bind (name, kind, kinds)
        val kinds   = foldl addKind emptyEnv
                        (⟨primitive type constructors for Typed μScheme :: 254b⟩ nil)
        val envs    = (kinds, types, values)
        val basis   = ⟨ML representation of initial basis (automatically generated)⟩
        val reader =
          defreader (false, stringsreader ("initial basis", basis), tuschemeSyntax)
    in  readCheckEvalPrint (reader, fn _ => (), fn _ => ()) envs
    end
```

kinds	: kind	env
types	: tyex	env
values	: value ref	env
envs	: env_bundle	

bind	212
BugInTypeChecking	269a
continue	269b
defreader	658b
emptyEnv	212
filereader	657a
PRIMITIVE	264a
readCheckEvalPrint	269b
stringsreader	657a
tuschemeSyntax	673b
TypeError	271a

The code for the primitives appears in Appendix G. It is similar to the code in Chapter 5, except that it supplies a type, not just a value, for each primitive.

The function `runInterpreter` takes one argument, which tells it whether to prompt.

⟨*initialization for Typed μScheme* 270b⟩+≡ (271a) ◁270b
```
  fun runInterpreter noisy =
    let fun writeln s = app print [s, "\n"]
        fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
        val reader =
          defreader (noisy, filereader ("standard input", TextIO.stdIn), tuschemeSyntax)
    in  ignore (readCheckEvalPrint (reader, writeln, errorln) initialEnvs)
    end
```

6.10.3 Learning about polymorphic type systems

13. Write a type checker for Typed μScheme. That is, implement `elabdef` in code chunk 268a. Although you could write this checker by cloning and modifying the type checker for Typed Impcore, you may have better luck building a checker from scratch by following the type rules for Typed μScheme.

    When you type-check VAL-REC(x, τ, e), be sure to check that x is not evaluated in e, as described on page 267.

    When you type-check literals, use the rules on page 265. Although these rules are incomplete, they should suffice for anything the parser can produce. If a literal `PRIMITIVE` or `CLOSURE` reaches your type checker, the impossible has happened, and your code should raise an appropriate exception.

14. Suppose we get sick and tired of writing `@` signs everywhere, so we decide to extend Typed μScheme by making PAIR, FST, and SND abstract syntax instead of functions.

    (a) What is the type of the following function?

    ```
    (type-lambda ('a) (type-lambda ('b)
       (lambda (((pair a b) p)) (pair (snd p) (fst p)))))
    ```

    (b) Using the type rules from the chapter, give a derivation tree proving the correctness of your answer to part 14a.

15. A great advantage of a polymorphic type system is that the language can be extended without touching the abstract syntax, the values, the type checker, or the evaluator. Without changing any of these parts of Typed μScheme, extend Typed μScheme with a `queue` type constructor and the polymorphic values `empty-queue`, `empty?`, `put`, `get-first`, and `get-rest`. (A more typical functional queue provides a single `get` operation which returns a pair containing both the first element in the queue and any remaining elements. If instead you write the two functions `get-first` and `get-rest`, you won't have to fool with pair types.)

    (a) What is the kind of the type constructor `queue`? Add it to the initial Δ for Typed μScheme.

    (b) What are the types of `empty-queue`, `empty?`, `put`, `get-first`, and `get-rest`?

    (c) Add them to the initial Γ and ρ of Typed μScheme. You will need to write implementations in ML.

16. *Without* changing the abstract syntax, values, type checker, or evaluator of Typed μScheme, extend Typed μScheme with the `pair` type constructor and the polymorphic functions `pair`, `fst`, and `snd`.

    (a) What is the kind of the type constructor `pair`? Add it to the initial Δ for Typed μScheme.

    (b) What are the types of `pair`, `fst`, and `snd`?

    (c) Add them to the initial Γ and ρ of Typed μScheme. As you add them to ρ, you can use the same implementations that we use for `cons`, `car`, and `cdr`.