COMP 105 Homework: Type Systems

Due Tuesday, March 29, at 11:59 PM (updated)

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

Setup

Make a clone of the book code:

```
git clone linux.cs.tufts.edu:/comp/105/build-prove-compare
```

You will need copies of `build-prove-compare/bare/tuscheme/tuscheme.sml` and `build-prove-compare/bare/timpcore/timpcore.sml`.

Dire warnings

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

Individual Problems

Working on your own, please solve the exercises **Exercise 8 on page 429 and Exercise 14 on page 430** described below. These exercises are in Volume 2 of the book.

Problem Details

8. *Adding lists to Typed Impcore*. Do Exercise 8 on page 429 of Ramsey. The exercise requires you to **design new syntax** and to **write type rules** for lists. I expect your type rules to be *deterministic*.

*Hint: This exercise 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. I recommend that you **do Pair Exercise 1 first**. It will give you more of a feel for monomorphic type systems.

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.

14. *Polymorphic functions in Typed uScheme*. Do Exercise 14 on page 430 of Ramsey. Note that `all?` should return true when given an empty list (the predicte vacuously holds of all the elements of the list).

Pair Problems

Working with a partner, please solve the exercises **Exercise 1 on page 426, Exercise 20 on page 434, Exercise 18 on page 434, and Exercise T** described below.

Problem Details

1. *Type checking arrays*. Do Exercise 1 on page 426 of Ramsey. The course solution to this problem is 21 lines of ML.

20. *Type checking polymorphic queues*. Do Exercise 20 on page 434 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 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
```

Hint: because `empty-queue` is *not* a function, you will have to modify the definition of `initialEnvs` on page 423. That page shows two places you will update: the *<primitive functions for Typed μScheme ::>* and the *<primitives that aren't functions, for Typed μScheme ::>*.
The course solution to this problem, including the implementation of `put` above, is under 20 lines of ML.

18. *Type checking Typed uScheme*. Do Exercise 18 on page 434 of Ramsey: write a type checker for Typed uScheme. Don't worry about the quality of your error messages, but do remember that your code must compile *without errors or warnings*. The course 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 the code gave worse error messages, it could have been a little shorter.

T. *Test cases for type checkers*. Create three test cases for Typed uScheme type checkers. In your test file, 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) ('a (list 'a) -> (list 'a)))    (NEW NOTATION)

(val e2 (@ car int))
; type is ((list int) -> int)                          (NEW NOTATION)

(val e3 (type-lambda ('a) (lambda (('a x)) x)))
; type is (forall ('a) ('a -> 'a))                     (NEW NOTATION)

(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. **PLEASE MAKE SURE YOU FOLLOW THE ABOVE DIRECTIONS EXACTLY**.

What to submit: Individual Problems

You should submit three files:

- a `README` file containing
  - the names of the people with whom you collaborated
  - the numbers of the problems that you solved
  - the number of hours you worked on the assignment.
- a PDF file `lists.pdf` containing your work on 8.
- a file `typesys-forall.scm` containing your code for Exercise 14 on page 430.

How to submit: Individual Problems

When you are ready, run `submit105-typesys-solo` to submit your work. Note you can run this script multiple times; we will grade the last submission.

What to submit: Pair Problems

For your joint work with your partner, you should submit four files:

- a `README` file containing
  - your answers to parts (a) and (b) of Exercise 20
  - a high-level description of the design and implemenation of your solutions
  - the number of hours you worked on the pair portion of the assignment
- a file `timpcore.sml` containing your code for Exercise 1.
- a file `tuscheme.sml` containing your code for Excercises 20 and Exercise 18.
- a file `type-tests.scm` containing your tests for Exercise T. Please make sure your test cases **exactly** conform to the requested format.

How to submit: Pair Problems

When you are ready, run `submit105-typesys-pair` to submit your work. Note you can run this script multiple times; we will grade the last submission.

How your work will be evaluated

We will use auto-grading to evaluate the functional correctness of your code. We will use the test cases that you submit to evaluate everyone's type checker. You will get points for every bug that your test case triggers. As for structure and organization, we will use the same criteria as in previous homework assignments.

Hints

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 7.6.4 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
```

This should 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 8, 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 8, 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 8, 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 8 on page 429, 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 14, 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 1, 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 20, 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 18, 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.

Back to the class home page