COMP 105 Homework: Type Systems

Due Thursday, March 17, 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`.

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 247 of Ramsey and Kamin: write type rules for lists. Turn in this exercise in PDF file `lists.pdf`.

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.

11. Do Exercise 11 on page 248 of Ramsey and Kamin: 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 247 of Ramsey and Kamin: 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.

16. Do Exercise 16 on page 249 of Ramsey and Kamin: add sum types to Typed uScheme. You need to do only parts (a), (b), and (d). The laws requested in part (c) are as follows:

```
(either (left  x) f g) = (f x)
(either (right y) f g) = (g y)
```

(You may recognize this interface as an instance of continuation-passing style; functions `f` and `g` are continuations.)

As a hint, remember that you can call `eval` from a primitive function. 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.
My solution to this problem is about 15 lines of ML.

13. Do Exercise 13 on page 248 of Ramsey and Kamin: write a type checker for Typed uScheme. Turn in this exercise in file `tuscheme.sml`, which should also include your solution to Exercise 16 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. It could have been a little shorter if I had given worse error messages.

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.

Sum types

Most statically typed programming languages have crappy support for sum types. For example, the closest you can get in C or C++ is a union. This is not very close. As a result, it's hard for you to have good intuitions about sum types. Help has arrived.

Example: produce a value of sum type

A common use of a sum type is a function that returns either a good value or an error message. For example, in a square-root routine, we might return a value of type `number + sym list`, returning a number when it makes sense and an error message otherwise. Here's an example in *untyped* uScheme:

<sqrt.scm>=

```
(define sqrt (n)
  (if (< n 0)
      (right '(square root of a negative number))
      (left  (findzero-between (lambda (x) (- (* x x) n)) 0 n))))
```

Consuming values of sum type

For a programmer, the most comfortable way to consume a value of sum type is with special syntax. The classic syntax is the case expression. Here's how we would do it in ML:

<sum.sml>= [D->]

```
datatype ('a, 'b) sum
  = LEFT  of 'a
  | RIGHT of 'b

...
    case sqrt n
      of LEFT x => x
       | RIGHT strings => raise RuntimeError (concat strings)
...
```

Defines sum (links are to index).

More generally we can write any consumption of sum type as follows:

<sum.sml>+= [<-D]

```
   case e
     of LEFT  x => e1
      | RIGHT y => e2
```

But what if we don't want to introduce new syntax? Then we can introduce the `either` function and convert the whole thing to continuation-passing style. In~the algebraic laws that are part of the sum problem above, `f`~is the left continuation and `g`~is the right continuation. We would CPS-convert the general code as follows:

```
(either e (lambda (x) e1) (lambda (y) e2))
```

This version is completely general, yet it requires no new syntax. Your mission is to do it in a typed way.

Advice

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

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 sum type, values of function type, values of list type) and the type language (sum 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:

- 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, if I have any luck, you will find in the index of your book.

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.

- *<sqrt.scm>*: D1
- *<sum.sml>*: D1, D2

- sum: D1