Computer Science 131, Fall 2001
Assignment 8: Monomorphic Type Inference

Out: Friday, November 30, 2001

Due: Friday, December 7, 5:00pm

To complete this assignment, make a new directory and copy all of the files from the directory `/cs/cs131/src/8`. You must use the binary `/cs/cs131/bin/sml-cs131-a8` to load the file `assign8.sml`. Complete the `assign8.sml` file and turn it in using `cs131submit`. All the code you write should all be placed in the structure `Infer` defined in the file `assign8.sml`, as shown there. This is the only file you should need to load in SML; all the others are provided simply for reference.

Many structures have been predefined for you: `Env` implements mappings from variables to whatever (used for typing contexts, exactly as in Assignment 5), and the new structure `Metavar` implements metavariables. The signatures for these structures are are given in the files `env.sig.sml` and `metavar.sig.sml`. Note that each metavariable is identified by a unique number; the n-th metavariable prints out as $\{n{:}t\}$ when it has been defined equal to some type t.

The source program syntax is again given the the structure `A`, whose signature is in the file `absyn.sig.sml`. The differences from Assignment 5 are:

- Real numbers and the integer-to-real conversion operation have been removed; arithmetic and comparison operations now only work on integer arguments.
- Function definitions (`fn` and `fun`) are no longer annotated with type arguments.
- The `ty` datatype has been extended with type metavariables.

You are again given a function `M.parse` that can read programs written in ML-syntax and produce a value of type `A.exp`; the only change is that the parser no longer wants type annotations on function definitions. You may want to use the test function `Infer.test` which takes a filename, reads the file, and runs your type inference function on the code.

Feel free to implement any helper functions you find convenient. Be sure to comment your code so that the graders can follow it!

1 Implementing Unification (50%)

In most formal mathematical treatments, unification is the procedure which, given two "phrases" u_1 and u_2 from some fixed language that includes metavariables, attempts to find a substitution σ of terms for metavariables such that $u_1\sigma = u_2\sigma$. (Here $u_1\sigma$ means to apply the substitution σ to the term u_1.)

In this assignment, we will take a more "imperative" view, which is commonly used in implementations of type inference. Metavariables are mutable (implemented using SML `ref`s), and the unification procedure will assign definitions to metavariables, rather than carrying around and applying substitutions.

1. The `assign8.sml` function contains the definition of a function

   ```
   follow : A.ty -> A.ty
   ```

   This function takes a type and returns an equivalent type that is not a metavariable with a definition. Given any non-metavariable type or an unset metavariable, this function simply returns its argument; given a metavariable with a definition, this function takes the definition and recursively applies `follow` (because it might turn out that the definition itself is a metavariable with a definition).

   Unification often generates chains of metavariables whose definitions are other metavariables. When doing a lot of unifications, it can be expensive to repeatedly traverse long chains. Begin by modifying the function `follow` to do "path compression". That is, the result of the function should not change, but after the function returns all the intermediate metavariables in the chain should be set directly to the final result. For example, if m_1 is set to m_2, and m_2 is set to m_3, and m_3 is set to m_4, and m_4 is unset, then not only should `follow` applied to m_1 return m_4, but afterwards m_1, m_2, and m_3 should all be set to m_4 as well. Thus later calls to `follow` applied to the type m_1 (and m_2 for that matter) will execute more quickly.

2. Write a function

   ```
   occurs : A.ty * (A.ty Metavar.metavar) -> bool
   ```

   that determines whether the given metavariable appears anywhere in the given monotype, including inside the definitions of defined metavariables. You may assume that the metavariable you are checking for does not have a definition yet. If it seems useful you can use your `follow` function, though this is not required. (The type `A.ty Metavar.metavar` represents a metavariable that may or may not have an `A.ty` as its definition.)

3. The `assign8.sml` file also contains a definition for the function

   ```
   unify : A.ty * A.ty -> unit
   ```

which takes its two arguments, calls `follow` on each type, and then passes the result to a function `unify'` which actually does the unification and gives values to any unset metavariables if necessary to make the two types equal. Complete the definition of `unify'`. This function should raise the exception `Unify` if the arguments cannot be unified.

2 Monomorphic Type Inference (50%)

As a first step, you are to implement type inference for a monomorphic type system (no special treatment of `let`). Complete the function

```
minfer : (A.ty Varmap.map * A.exp) -> A.ty
```

This function will take a typing context (mapping variables to monotypes) and an expression, and return the resulting type if one exists (just like `typeof` in Assignment 5). See `ord-map-sig.sml` for signature of the `Varmap` structure.

For each case in `minfer` you should first recursively do type inference on sub-expressions, then consider what constraints among the types inferred for the subexpression and the type to be returned by `minfer` must hold. These can immediately be enforced by calls to `unify`.

If the given expression cannot be made to typecheck, `minfer` should raise an exception.