Computer Science 131, Fall 2000
Assignment 6: Type Inference

Out: Friday, November 3

Due: Friday, October 10, 5:00pm

To complete this assignment, make a new directory and copy all of the files from the directory `/cs/cs131/src/6`. Complete the `assign6.sml` file and turn it in using `cs131submit`.

For this assignment, instead of loading code with the command `use`, you are to use the SML/NJ compilation manager. This is similar to Unix command `make`, except that dependency information is computed automatically. If you invoke the function `CM.make()` then the compilation manager will read the list of filenames in the file `sources.cm`, figure out dependencies, and automatically compiles the necesary files in the right order.

Many structures have been predefined for you: `Metavar` implements metavariables, `Var` implements type and term variables, `Varmap` implements mappings from variables to whatever (used for typing contexts), `Metavarset` implements sets of metavariables. You should not need to look at the code for these; the signatures are given in the files `metavar.sig.sml`, `var.sig.sml`, `ord-map-sig.sml`, and `ord-set-sig.sml` respectively. Note that each variable and metavariable has a unique number, and that the metavariable n is printed as $\{n{:}t\}$ when it has contents t.

The definition of un-annotated program syntax is given in the structure `S`, whose signature is in the file `source.sig.sml`. Types and annotated syntax are defined in the structure `T`, whose signature is in the file `target.sig.sml`.

Your code should all be placed in the structure `Infer` defined in the file `assign6.sml`, as shown there.

Feel free to implement any helper functions you find convenient; you are required to include comments sufficient for the grader to be able to understand your code.

1 Implementing Unification (30%)

In CS 80 you were introduced to the idea of *unification*, which was used there for the purposes of resolution theorem proving. Unification comes up in several other contexts, including SML type inference and in the execution of logic programming languages like Prolog.

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.) Normally one wants the "most general unifier" — the subsitution that makes the fewest commitments.

In this assignment, we will take a more "imperative" view, which is commonly used in implementations of type inference. Metavariables will be assignable (implemented with SML `ref`s), and the unification procedure will do the necessary assignments to metavariables, rather than carrying around and applying substitutions.

Since the goal of this assignment is to implement type inference, the phrases being unified will be types. For example,

- If m_1 and m_2 are metavariables whose contents are unknown, then unifying m_1`->Bool` with `Int->`m_2 will require setting m_1 to be `Int` and m_2 to be `Bool`.
- If m_1 and m_2 are metavariables whose contents are unknown, then unification of m_1`->`m_2 with `Int->`m_2 will require setting m_1 to be `Int` and will not affect m_2. (It is true that by setting m_2 to be `Bool`, say, the two terms would still be equal, but this would not be the most general result possible.)
- If m_1 and m_2 are metavariables whose contents are unknown, then unification of m_1`->`m_2 with m_1`->`m_1 will require setting m_1 to be m_2. (Or vice-versa...it doesn't matter.) This records the fact that whatever m_2 turns out to be, m_1 will have the same value.
- If m_1 and m_2 are metavariables whose contents are unknown, then unification of m_1`->`m_2 with m_2`->Bool` might plausibly first set m_1 to be m_2 (by unifying the domains of the function types) and then set m_2 to be `Bool` (by unifying the codomains of the function types). Note that this may yield "chains" of metavariables: after this unification we know that making the two given types equal requires plugging in `Bool` for m_1, but this is represented by the fact that the value of m_1 is equal to that of the metavariable m_2, whose value in turn is `Bool`.

1. The `assign6.sml` function contains the definition of a function

```
follow : T.monotype -> T.monotype
```

   This function acts as the identity on any non-metavariable type. However, given a chain of metavariables it will follow this to return an equivalent non-metavariable type or an un-set metavariable. When doing a lot of unifications, such chains can become long and may be traversed repeatedly. Modify 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 : T.monotype * T.monotype Metavar.metavar -> bool
```

that determines whether the given metavariable appears anywhere in the given monotype. (Note that you will also need to check inside the definitions of metavariables in the given monotype.) It may be helpful to assume that the given metavariable has not yet been assigned to; you may also wish to use the `follow` function. In your comments, be sure to note any preconditions required by your function.

3. The `assign6.sml` file also contains a definition for the function

```
unify : T.monotype * T.monotype -> unit
```

which takes its two arguments, calls `follow` on each type, and then passes the result to `unify'` which actually does the unification. Complete the definition of `unify'`. This function should raise the exception `Unify` if the arguments cannot be unified. Remember:

- Unifying a constant, or type variable, or a type metavariable with itself requires no work.
- Otherwise, unifying an unset metavariable with any other type can be done just by assigning the type to the metavariable, unless the metavariable appears in that type. (So, for example, one cannot unify m_1 with `Int->`m_1.)
- Unifying two arrow or product or list types can be done with recursive calls. (Careful...what function needs to be recursively called?)

2 Monomorphic Type Inference (40%)

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

```
minfer : (T.monotype Varmap.map * S.exp) -> (T.exp * T.monotype)
```

This function will take a typing context (mapping variables to monotypes) and an unannotated expression, and will return the corresponding annotated expression and its type.

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.

3 Polymorphic Type Inference (30%)

1. Write the function

   ```
   occursInContext : T.polytype Varmap.map * T.monotype Metavar.metavar -> bool
   ```

   which checks whether the given metavariable occurs in any type assigned to a variable by the given typing context.

2. Write the function

   ```
   findUnconstrained : T.polytype Varmap.map * T.monotype ->
                               Metavarset.set
   ```

   which returns the set of all unset metavariables appearing in the given monotype which do not occur in the given context.

3. Write the function

   ```
   generalize : T.polytype Varmap.map * T.exp * T.monotype ->
                     T.exp * T.polytype
   ```

   This function is given a typing context mapping variables to *polytypes*, an annotated expression e and its inferred type u, and makes the expression as polymorphic as possible. This requires performing operations equivalent to the following (though not necessarily in this order):

   - It should find the unconstrained metavariables $\{m_1, \ldots, m_n\}$ in u
   - Allocate n fresh type variables $\{\alpha_1, \ldots, \alpha_n\}$
   - Set m_i to be α_i for $1 \leq i \leq n$
   - Return the polymorphic expression `Fn` α_1`=>`$\cdots$`=>Fn` α_n`=>`e and its type $\forall\alpha_1.\ldots.\forall\alpha_n.u$.

   (In the case that $n = 0$, of course, the returned expression is just the same as the given expression.)

4. Write the function

   ```
   instantiate : T.exp * T.polytype -> T.exp * T.monotype
   ```

   which applies the given expression with as many fresh type metavariables as is necessary to make it no longer polymorphic. That is, this function takes an expression e and its polymorphic type $\forall\alpha_1.\ldots.\forall\alpha_n.u$, and the function returns the expression $e[m_1]\cdots[m_n]$ and this expression's (mono)type, where $m_1, \ldots, m_n$ are fresh metavariables. If $n = 0$ then `instantiate` should simply return the given expression.

You may use the function `substClosedInPolytype` which substitutes a monotype containing no free *type variables* for a type variable in a given polytype. (Conveniently, an unset metavariable contains no free type variables.)

5. Finally, define the function

```
pinfer : (T.polytype Varmap.map * S.exp) -> (T.exp * T.monotype)
```

to do polymorphic type inference. This should be basically the same code as for `minfer`, except that:

- The function's name is different
- When typechecking `let` x `be` e_1 `in` e_2, if e_1 can be made polymorphic, then after doing the type inference use `generalize` to make this definition as polymorphic as possible. Use the function `S.generalizable` to determine whether e_1 can safely be made polymorphic.
- When given a context Γ and an ordinary term variable x, rather than returning x and $\Gamma(x)$, return `instantiate(`x`,`$\Gamma(x)$`)`
- A few minor changes may be necessary when extending the typing context, since it now contains polytypes rather than monotypes.