Computer Science 320 (Fall, 2016)
Concepts of Programming Languages

Homework Assignment 8: Type Inference

Out: December 1, 2016
Due: Wednesday, December 14, 2016, by 11:59 pm

In this assignment, you will implement a type inference algorithm for a small subset of Haskell. You will make changes to the `Ty` module. The only modules which are different from previous versions of the interpreter are the `Main` and `Ty` modules, so you may swap in either your own versions of the `Eval`, `Exp`, and `Env` modules, or the versions provided as solutions for Homework Assignments #6 and #7.

You may not modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: Problem 5 is a bonus problem, a solution for which is worth up to 30 points of extra credit.

Problem 1. (25 pts)

You will implement a type checker for a subset of mini-Haskell. Familiarize yourself with the abstract syntax for types (see, for example, Section 7.5 in Lapets' Lecture Notes, or the posted handouts following the mid-term exam). Note that there are base types, and a constructor `Arrow::Ty -> Ty -> Ty` to construct a function type.

(a) Implement a function `tyOp::Oper -> Ty` which returns the type of an operator. You may assume that `(==)` can only be applied to integers, and that only integer lists can be constructed (which means, for example, that the type of `[]` is an integer list).

(b) Implement a function `ty0::Exp -> Error Ty` which can successfully type check all primitives (unit, empty list, operator, integer, and boolean), `if` statements, applications, and unit lambda abstractions (these are all the cases which do not have variables).

Since there are no type variables, you may use derived equality (`==`) on types. Remember that both branches of an `if` statement must have the same type, and that an `if` condition must have a boolean type. For application, you may need to pattern

match on the type of the function in order to check that its type matches the type of its argument.

Your solution should be able to type check `tests4.mhs` successfully, and should reject `tests1.mhs` (provided with previous assignments) for not being well-typed.

Problem 2. (10 pts)

Performing type inference on expressions which may contain variables requires *type* variables (in terms of the syntax for types, it is a type part of which is built using the `TyVar::String -> Ty` constructor). We cannot simply use derived equality on types when type variables are present, so we will need to define a unification algorithm which attempts to find a minimal substitution of type variables which makes two types equal. Informally, two types which contain variables are equal if we can somehow replace those variables in both expressions and obtain two equal expressions. For example, given the types `Int -> b` and `a -> Bool`, the minimal substitution which makes them equal is one which replaces all occurrences of `a` with `Int` and all occurrences of `b` with `Bool`.

In order to implement this algorithm, we will need to define substitutions, which are simply functions that substitute occurrences of a type variable with some other type. Thus, the type `Subst` is defined inside the `Ty` module to be

```
type Subst = Ty -> Ty
```

You will define a few functions for constructing and manipulating substitutions.

(a) Define a function `idsubst::Subst` which makes no changes to a type, and a function `o: Subst -> Subst -> Subst` which takes two substitutions and applies them in sequence, one after another.

(b) Define a function `subst::String -> Ty -> Subst` which takes a string x representing a type variable name, and a type t which will be used to substitute that variable. It should then return a substitution which, when given a type t', will substitute all occurrences of x in that type with t. Note that it *should not* substitute any other type variables which might be found in the type t'.

(c) Notice the type definition `type FreshVars = [Ty]`. Define a value `freshTyVars::FreshVars`, an infinite list of type variables in which no type variable is ever repeated.

Problem 3. (25 pts)

You will implement the unification algorithm for types. We say a substitution `s` *unifies* two types `t1` and `t2` if `s t1 == s t2`, where `==` is derived equality on types.

(a) Define a function `unify::Ty -> Ty -> Error Subst` which takes two types and finds the minimal substitution which makes them equal if one exists, and returns an error otherwise.

Note that given two types with no type variables, as long as they are equivalent in terms of derivable equality, a trivial substitution which makes no changes to the types is sufficient to unify them. On the other hand, if two types without type variables are not equivalent, there exists no substitution which can unify them.

Also, remember that a type variable can be substituted for any type, including another type variable. Finally, be very careful when unifying two function types (constructed using `Arrow::Ty -> Ty -> Ty`. First, try to unify the argument types. If a substitution is obtained, be sure to apply it to the result types *before* trying to unify them. For example, consider the two types `a -> a` and `Int -> Bool`, which cannot be unified. If we try to unify `Int` and `a`, we will obtain a substitution which replaces all instances of `a` with `Int`. This should be taken into account when trying to unify `Bool` and `a`.

Problem 4. (40 pts)

You will complete the already partially-implemented type inference function `ty::Env Ty -> FreshVars -> Exp -> Error (Ty, Subst, FreshVars)`. This function maintains an environment which maps variables in expressions to their types, and binds fresh *type* variables to any new variables it encounters in lambda abstractions or let-bindings. Each time the function processes some node in an expression tree, it may need to unify the types of the children of that node, and so, it accumulates these substitutions, returning the to the caller.

Note carefully the base cases for primitives, which return the trivial substitution, along with the types

(a) Add the base cases for expressions (unit, the empty list, integers, boolean values, and operators) to the definition of the function `ty`. You may simply return the fresh list of variables unchanged, and a trivial substitution which makes no changes to the types.

(b) The case for `if` expressions in the definition of `ty` is not complete. Notice that the function

```
tys::Env Ty -> FreshVars -> [Exp] -> Error ([Ty], Subst, FreshVars)
```

is used to obtain the types for all three subexpressions simultaneously. Perform the remaining checks and unifications in order to complete this part of the definition of the function. Remember that any substitutions you might generate in the process (including the one that has already been generated when type checking the three subexpressions) need to be considered in every subsequent unification. Also, when returning the substitution(s) you generated, remember to use `o::Subst -> Subst -> Subst` to combine them in the proper order.

(c) The case for unit lambda abstractions in the definition of `ty` is not complete. Complete this part of the definition. Remember to return the type of the abstraction (it takes a value of type unit as an argument).

(d) The case for general lambda abstractions in the definition of `ty` is not complete. Complete this part of the definition. Because a new variable is encountered, you will need to obtain a fresh *type* variable, and bind this expression variable to the fresh type variable in the environment. The subexpression should be type-checked under this new environment.

Remember that the argument type of the abstraction is the same as the type of the variable over which the abstraction is defined. You may want to look at the part of the definition of `ty` for let bindings for some guidance. The cases for variables and application have already been completed for you.

Problem 5. (*30 extra credit pts)

(a) Notice that the mini-Haskell type inference algorithm is able to generate polymorphic types for individual values. For example, given the expression `\ x -> x`, the type should look something like `t1 -> t1`, where `t1` is a fresh variable generated using `freshTyVars`. Define the function `freevars::Ty -> [String]`, which returns the list of free type variables in a type.

(b) In the `Ty` module, notice the definition of a syntax for polymorphic types, `data PolyTy = ForAll [String] Ty`. Use your solution from part (a) to implement the function `canon::Ty -> PolyTy` which takes a type $t(\alpha_1, \ldots, \alpha_n)$ and returns a polymorphic type $\forall \alpha_1 \ldots \alpha_n . t(\alpha_1, \ldots, \alpha_n)$ in which every free variable has been universally quantified.

(c) The `Main` module uses values of type `data AnnotVal = AnnotVal Val Ty` to display values annotated with their types. Modify the `show` function for `AnnotVal` so that if a value's type is polymorphic, it is displayed in its explicitly quantified form.

(d) In Problems #1 and #3, we assumed that `(==)` can work only on integers. Modify the function(s) `ty0` and/or `ty` (as well as any others you may need to modify, such as `tyOp`) so that `(==)` will type check if it is applied to two boolean values as well as to two integers.

(e) Try using a similar approach to make `[]` and `cons` polymorphic. (Note: This problem is fairly open-ended and potentially challenging).