HW 18 **Prof. Caldwell**
Due: 3 December 2013 **COSC 3015**

1 Type Inference

Recall the type of terms.

```
data Term = V String
          | Ap Term Term
          | Abs String Term
```

The data-type `Type` with products is:

```
data Type = TyVar String | Arrow Type Type deriving Eq
```

1.1 Proof Rules

Sequents in the system (which represent the state of a derivation) are of the form
Sequents in the system (which represent the state of a type derivation) are of the form:

$$\Gamma, E \vdash M : T$$

In this structure, Γ is a *context* representing a state of knowledge about the types of some variables. Contexts have the form:

$$\Gamma = [x_1 : \tau_1, \cdots, x_k : \tau_k]$$

where the x_i's are variables and τ_i's are types.

E is a list of constraints between pairs of types and in the rules is presented as follows:

$$E = \{\tau_{(1,1)} = \tau_{(1,2)}, \cdots, \tau_{(k,1)} = \tau_{(k,2)}\}$$

wher $\tau_{i,j}$'s are types.

We write $\Gamma \backslash x$ to denote the list obtained from Γ by deleting all pairs whose first element is x.

As presented in the last homework, The proof rules for Wand's type inference system are given as follows:

$$\frac{}{\Gamma, \{\alpha = \tau\} \quad \vdash \quad x : \tau}(\text{Ax}) \qquad \text{if } (x, \alpha) \in \Gamma.$$

$$\frac{[x : \alpha]{+\!+}(\Gamma \backslash x),\, E \quad \vdash \quad M : \beta}{\Gamma, E \cup \{\tau = \alpha \rightarrow \beta\} \quad \vdash \quad \lambda x.M : \tau}(\text{Abs}) \qquad \text{where } \alpha \text{ and } \beta \text{ are fresh.}$$

$$\frac{\Gamma, E_1 \vdash M : \alpha \rightarrow \tau \qquad \Gamma, E_2 \vdash N : \alpha}{\Gamma, E_1 \cup E_2 \vdash MN : \tau}(\text{App}) \qquad \text{where } \alpha \text{ is fresh.}$$

A derivation in this system is a tree of instances of these rules where the leaves of the tree are all instances of the (Ax) rule. To construct a proof that a closed term (no free variables) (say M) has a type, we postulate that M has some type (say α) and proceed by recursion on the structure of M to show

$$\exists E.[(Type, Type)]. \text{ such that the sequent } [], E \vdash M : \alpha \text{ is derivable.}$$

To find E, we use the proof rules above to try to construct a derivation (leaving the E's blank to start) and then propagate the constraints in the E's back down through the derivation tree from the leaves.

Example 1.1. Here is an example of a derivation that $\lambda x.x$ has a type by starting with the sequent of the form $[], \{??\} \vdash (\lambda x.x) : \tau$. The term is an abstraction so we apply the rule (Abs).

$$\dfrac{[x:\alpha], E \vdash x : \beta}{[], \{\tau = \alpha \to \beta\} \cup E \vdash (\lambda x.x) : \tau}\ (\text{Abs})$$

But if we fill in the set E with the constraint $\tau = \alpha$, we have an instance of the Axiom rule.

$$\dfrac{\dfrac{E = \{\beta = \alpha\}}{[x:\alpha], E \vdash x : \beta}\ (\text{Ax})}{[], \{\tau = \alpha \to \beta\} \cup E \vdash (\lambda x.x) : \tau}\ (\text{Abs})$$

If we completely instantiate the sets E we get the following complete derivation.

$$\dfrac{\dfrac{}{[x:\alpha], \{\beta = \alpha\} \vdash x : \beta}\ (\text{Ax})}{[], \{\tau = \alpha \to \beta, \beta = \alpha\} \vdash (\lambda x.x) : \tau}\ (\text{Abs})$$

The fact that there is a derivation indicates that the term $(\lambda x.x)$ does have a type. We use the constraint set E to actually determine the type of $\lambda x.x$. To do this, we unify the set E and apply the resulting substitution to the type τ. For this case, when we unify E we get the substitution $[\tau := \alpha \to \alpha, \beta := \alpha]$. Applying this substitution to τ we determine that $(\lambda x.x) : \alpha \to \alpha$.

We can also do type derivations for terms containing free variables if we assume those free variables do have types.

Example 1.2. Consider the term $y(\lambda x.x)$. This should have a type if $y : (\alpha \to \alpha) \to \beta$.

We start by trying to show there is some E such that there is a derivation of the sequent

$$[y : (\alpha \to \alpha) \to \beta], E \vdash y(\lambda x.x) : \tau$$

Since the term is an application, we use the (Ap) rule.

$$\dfrac{[y : (\alpha \to \alpha) \to \beta], E_1 \vdash y : \alpha' \to \tau \qquad [y : (\alpha \to \alpha) \to \beta], E_2 \vdash (\lambda x.x) : \alpha'}{[y : (\alpha \to \alpha) \to \beta], E_1 \cup E_2 \vdash y(\lambda x.x) : \tau}\ (\text{Abs})$$

The left branch is an instance of an axiom because there is an entry for the variable y in the context.

$$\dfrac{\dfrac{E_1 = \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\}}{[y : (\alpha \to \alpha) \to \beta], E_1 \vdash y : \alpha' \to \tau}\ (\text{Ax}) \qquad [y : (\alpha \to \alpha) \to \beta], E_2 \vdash (\lambda x.x) : \alpha'}{[y : (\alpha \to \alpha) \to \beta], E_1 \cup E_2 \vdash y(\lambda x.x) : \tau}\ (\text{Abs})$$

On the right branch we rebuild the proof given above.

$$
\dfrac{\dfrac{E_1 = \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\}}{[y : (\alpha \to \alpha) \to \beta], E_1 \vdash y : \alpha' \to \tau}\,(\mathrm{Ax}) \qquad \dfrac{\dfrac{E_3 = \{\beta' = \alpha''\}}{[x : \alpha'', y : (\alpha \to \alpha) \to \beta], E_3 \vdash x : \beta'}\,(\mathrm{Ax})}{[y : (\alpha \to \alpha) \to \beta], E_2 = (\{\alpha' = \alpha'' \to \beta'\} \cup E_3) \vdash (\lambda x.x) : \alpha'}\,(\mathrm{Abs})}{[y : (\alpha \to \alpha) \to \beta], E = (E_1 \cup E_2) \vdash y(\lambda x.x) : \tau}\,(\mathrm{Abs})
$$

Putting together the constraints, we get the following set:

$$
\begin{array}{rcl}
E & = & E_1 \cup E_2 \\
 & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\} \cup (\{\alpha' = \alpha'' \to \beta'\} \cup E_3) \\
 & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\} \cup (\{\alpha' = \alpha'' \to \beta'\} \cup \{\beta' = \alpha''\}) \\
 & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta, \alpha' = \alpha'' \to \beta', \beta' = \alpha''\})
\end{array}
$$

Unification of this results in the substitution:

$$
s = [a' := (b' \to b'), t := b, a := b', a'' := b']
$$

When s is applied to τ we get the type β, as expected.

1.2 Implementation

In Haskell we encode contexts as list of type `[(String,Type)]`. Constraint sets are represented in the Haskell implementaiton as a list of type `[(Type,Type)]`. M denotes a lambda-term, and in Haskell is represented by elements of the data-type `Term`. T denotes a type and is represented in Haskell by elements of the data-type `Type`.

The implementation Here is the type of the `infer_type` function:

```
infer_type :: [(String, Type)]
              -> Term
              -> Type
              -> [String]
              -> ([(Type, Type)], [String])
```

This function takes a context (denoted Γ in the rules above and represented by a list of `String`, `Type` pairs.), a term to infer the type of, a type (denoted τ in the rules above and initially a type variable not occurring anywhere in the context), and a string list containing the names of all variables used so far.

```
infer_type context trm typ vars =
 case trm of
   (V x) ->
      case (lookup x context) of
        (Just a) -> ([(typ,a)],vars)
        Nothing ->  error ("infer_type: " ++ x ++ " not in context!")
   (Ap m n) ->
      let a = fresh "a" vars in
      let (e1,vars1) = infer_type context m (Arrow (TyVar a) typ)(a:vars) in
      let (e2,vars2) = infer_type context n (TyVar a) vars1 in
        (e1 ++ e2, vars2)
```

```
(Abs x m) ->
   let a = fresh "a" vars in
   let b = fresh "b" (a : vars) in
   let (e1,vars1) = infer_type ((x,(TyVar a)):context) m (TyVar b) (a:b:vars) in
     ( [(typ, Arrow (TyVar a) (TyVar b))] ++ e1 , vars1)
```

The case `V x` implements the Axiom rule, the case labeled `(Ap m n)` implements the (Ap) rule and the case labeled `(Abs x m)` implements the (Abs) rule.

1.3 Adding product types.

To add product types we extend the data-types `Type` and `term` as follows:

```
data Op = Arrow | Product
        deriving (Eq,Show)

data  Type =   TVar  String |  BinType Op  Type   Type
        deriving (Eq)

data  Term =   Var String
             | Abs  String Term
             | Ap Term Term
             | Pair Term Term
             | Fst Term
             | Snd Term
        deriving (Eq)
```

Mathematically we write $M \times N$ for the Haskell term `Prod A B` and render the Haskell term `(Pair M N)` as $\langle M, N, \rangle$.
Here are the additional proof rules:

$$\frac{\Gamma, E_1 \vdash M : \alpha \qquad\qquad \Gamma, E_2 \vdash N : \beta}{\Gamma, E_1 \cup E_2 \cup \{\tau = \alpha \times \beta\} \vdash \langle M, N \rangle : \tau}\text{(Pair)} \qquad \text{where } \alpha \text{ and } \beta \text{ are fresh.}$$

$$\frac{\Gamma, E \quad \vdash \quad M : \tau \times \alpha}{\Gamma, E \quad \vdash \quad \mathit{fst}\, M : \tau}\text{(Fst)} \qquad \text{where } \alpha \text{ is fresh.}$$

$$\frac{\Gamma, E \quad \vdash \quad M : \alpha \times \tau}{\Gamma, E \quad \vdash \quad \mathit{snd}\, M : \tau}\text{(Snd)} \qquad \text{where } \alpha \text{ is fresh.}$$

Exercise 1.1. Using the base code provided (which includes unification for products) extend the `infer_type` function to implement these additional type inference rules.

Exercise 1.2. Design a number of test cases to show that your extension works. You should at least include the test examples in the file `hw18_expected.txt`. Include at least six more.