HW 12 **Prof. Caldwell**
Due: 11 October 2011 **COSC 3015**

1 Unifying a list of constraints

We represent constraints in Haskell as pair of types.

type Constraint = (Type, Type)

We will write $\tau_1 \stackrel{c}{=} \tau_2$ to denote the constraint pair (τ_1, τ_2).

Definition 1.1 (Satisfiability of a constraint list) Given a list of constraints of the the form $[\tau_1 \stackrel{c}{=} \tau_1', \cdots, \tau_k \stackrel{c}{=} \tau_k')]$ we say the list is *satisfiable* if there is a single substitution s that unifies all constraints in the list.

Recall that the function *unify* has type *Type* → *Type* → *Substitution*. building on the definitions from the previous assignment we give the definition for a function to unify a list constraints.

```
unifyL :: [Constraint] → Substitution
unifyL xs = unifyAll xs idSubst
   where unifyAll [] ans = ans
         unifyAll ((t1,t2):cs) ans = unifyAll (map substpair cs) ((subst s) . ans)
            where s = unify t1 t2
                  substpair (t1,t2) = (subst s t1, subst s t2)
```

2 Type Inference

Given a term (a program) the type inference algorithm determines if there is a type for that term and if so, what the type is.

There are two stages. The first stage is to build a set of constraints and the second is to solve them (if possible) yielding a substitution which is then applied to determine the type of the term. The constraint building phase is defined by a set of proof rules used to build a type derivation. These rules decompose the term and build a set of constraints which, if satisfiable, yields a substitution that can be used to find the type of the term.

We will present the set of proof rules (due to Michell Wand) to build a derivation (if one exists) which yields a list of constraints. We define a Haskell function which "implements" these rules. We then use the function *unifyL* to find a solution, if one exists.

2.1 The Proof Rules

The state of a type derivation is recorded in a structure of the following form:

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

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.

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

$$C = \{\tau_1 \stackrel{c}{=} \tau_1', \cdots, \tau_k \stackrel{c}{=} \tau_k'\}$$

where τ_i's are types.

We write $(x : \tau) : \Gamma$ to denote the list obtained from Γ by consing the pair $(x : \tau)$ on the left end. Haskell's `lookup` function can be used to find the type τ paired with the leftmost occurrence of a pair having x as its first element.

The proof rules for Wand's type inference system are given as follows:

$$\frac{}{\Gamma, \{\tau = \tau'\} \quad \vdash \quad x : \tau}(\text{Var}) \qquad \text{if } \mathit{lookup}\ x\ \Gamma = \mathit{Just}(\tau')$$

$$\frac{\Gamma, C_1 \vdash M : \alpha \to \tau \qquad \Gamma, C_2 \vdash N : \alpha}{\Gamma, C_1 \cup C_2 \vdash MN : \tau}(\text{App}) \qquad \text{where } \alpha \text{ is fresh.}$$

$$\frac{(x : \alpha) : \Gamma, C \quad \vdash \quad M : \beta}{\Gamma, \{\tau = \alpha \to \beta\} \cup C \quad \vdash \quad \lambda x.M : \tau}(\text{Abs}) \qquad \text{where } \alpha \text{ and } \beta \text{ are fresh.}$$

A type 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 derivation 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:

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

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

Example 2.1. 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).

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

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

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

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

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

The fact that there is a derivation indicates that the term $(\lambda x.x)$ may have a type. We use the constraint set C to actually determine the type of $\lambda x.x$. To do this, we unify the set C and apply the resulting substitution to the type τ. For this case, when we unify C we get the substitution $[\tau \mapsto \alpha \to \alpha, \beta \mapsto \alpha]$. 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 2.2. 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 C such that there is a derivation of the sequent

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

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

$$\dfrac{[y : (\alpha \to \alpha) \to \beta], C_1 \vdash y : \alpha' \to \tau \qquad [y : (\alpha \to \alpha) \to \beta], C_2 \vdash (\lambda x.x) : \alpha'}{[y : (\alpha \to \alpha) \to \beta], C_1 \cup C_2 \vdash y(\lambda x.x) : \tau}\ (\text{App})$$

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

$$\dfrac{\dfrac{C_1 = \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\}}{[y : (\alpha \to \alpha) \to \beta], C_1 \vdash y : \alpha' \to \tau}\ (\text{Var}) \qquad [y : (\alpha \to \alpha) \to \beta], C_2 \vdash (\lambda x.x) : \alpha'}{[y : (\alpha \to \alpha) \to \beta], C_1 \cup C_2 \vdash y(\lambda x.x) : \tau}\ (\text{App})$$

On the right branch we rebuild the proof given above.

$$\dfrac{\dfrac{C_1 = \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\}}{[y : (\alpha \to \alpha) \to \beta], C_1 \vdash y : \alpha' \to \tau}\ (\text{Var}) \qquad \dfrac{\dfrac{C_3 = \{\beta' = \alpha''\}}{[x : \alpha'', y : (\alpha \to \alpha) \to \beta], C_3 \vdash x : \beta'}\ (\text{Var})}{[y : (\alpha \to \alpha) \to \beta], C_2 = (\{\alpha' = \alpha'' \to \beta'\} \cup C_3) \vdash (\lambda x.x) : \alpha'}\ (\text{Abs})}{[y : (\alpha \to \alpha) \to \beta], C = (C_1 \cup C_2) \vdash y(\lambda x.x) : \tau}\ (\text{App})$$

Putting together the constraints, we get the following set:

$$\begin{array}{rcl} C & = & C_1 \cup C_2 \\ & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\} \cup (\{\alpha' = \alpha'' \to \beta'\} \cup C_3) \\ & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta\} \cup (\{\alpha' = \alpha'' \to \beta'\} \cup \{\beta' = \alpha''\}) \\ & = & \{\alpha' \to \tau = (\alpha \to \alpha) \to \beta, \alpha' = \alpha'' \to \beta', \beta' = \alpha''\}) \end{array}$$

Unification of this results in the substitution:

$$s = [a' \mapsto (b' \to b'), \tau \mapsto b, a \mapsto b', a'' \mapsto b']$$

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

2.2 Implementation

We implemented the *infer* function in class. Recall that we encoded the contexts by the following type.

type Context = [(String, Type)]

The *infer* function essentially implements a derivation. To be able to choose *fresh* variables we need to keep track of the all the variable names used in building the derivation. To do this we will pass a list of variables used so far (a list of strings) and return them together with the list of constraints. In this way, we thread a list of the variables used so far through the computation.

The type of the `infer` function is as follows:

```
infer :: Context → Term → Type → [String] → ([Constraint], [String])
```

This function takes a context (denoted Γ in the rules above and represented by the type *Context* here), 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 the type variables used so far.

The following function is used to generate fresh variables. It is a bit more elegant than the quick and dirty version we sued in class.

```
fresh :: String → [String] → String
fresh x xs = if not(x ‘elem‘ xs) then x else fresh’ x 0
   where  fresh’ x i  = if not(x’ ‘elem‘ xs) then x’ else fresh’ x (i + 1)
       where x’ = x ++ (show i)
```

Here is the infer function we implemented in class.

```
infer ctx  (Var x) ty vars =
   case (lookup x ctx) of
      Just t → ([(ty,t)],vars)
      Nothing → error ”infer: Var-case  failure”

infer ctx (Ap t1 t2) ty vars = (c1 ++ c2, vars2)
    where (c1,vars1) = infer ctx t1 (BinType Arrow (TVar a) ty) (a : vars)
          (c2,vars2) = infer ctx t2 (TVar a) vars1
          a =  fresh ”a” vars

infer ctx (Abs x t) ty vars = ((BinType Arrow (TVar a) (TVar b), ty) : c, vars’)
   where (c,vars’) = infer ((x, TVar a) : ctx) t (TVar b) (a : b : vars)
         a = fresh ”a” vars
         b = fresh ”b” vars
```

The function is defined by recursion on the structure of the term whose type is being inferred. The case *(Var x)* implements the Var rule, the case labeled *(Ap m n)* implements the (Ap) rule and the case labeled *(Abs x m)* implements the (Abs) rule.

2.3 Adding product types.

We already included product types (pairs) in the type *Type* but we did not include any mends for forming pairs or destructing pairs in the programming language. To do so we extend the language as follows:

$$
\begin{aligned}
&\textit{data Term} = \textit{Var String}\\
&\qquad\quad \mid \textit{Ap Term Term}\\
&\qquad\quad \mid \textit{Abs String Term}\\
&\qquad\quad \mid \textit{Pair Term Term}\\
&\qquad\quad \mid \textit{Fst Term}\\
&\qquad\quad \mid \textit{Snd Term}
\end{aligned}
$$

Here is a comparison of the mathematical notation and the corresponding Haskell notation.

Mathematical Notation	Haskell Notation
x	*(Var x)*
(M N)	*(Ap m n)*
$\lambda x.M$	*(Abs x m)*
$\langle M, N\rangle$	*(Pair m n)*
fst M	*(Fst m)*
snd M	*(Snd m)*

Recall the computation rules for *fst* and *snd*.

$$
\mathit{fst}\ \langle m, n\rangle = m \qquad \mathit{snd}\ \langle m, n\rangle = n
$$

Here are the proof rules keyed to the new kinds of terms.

$$
\frac{\Gamma, C_1 \vdash M : \alpha \qquad\qquad \Gamma, C_2 \vdash N : \beta}{\Gamma, C_1 \cup C_2 \cup \{\tau = \alpha \times \beta\} \vdash \langle M, N\rangle : \tau}(\text{Pair}) \qquad \text{where } \alpha \text{ and } \beta \text{ are fresh.}
$$

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

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

Exercise 2.1. Using the base code provided extend the `infer` function to implement these additional type inference rules.