HW 11 **Prof. Caldwell**
Due: 21 November 2013 **COSC 3015**

1 Types and Terms

Any discussion of type inference for a programming language involves two languages: a language of type expressions and the programming language itself.

We start with a simple programming language – the λ-calculus with pairs. Let

$$\mathcal{V} = \{x, y, z, w, x_1, y_1, z_1, w_1, \cdots\}$$

be an unbounded set of variables. The, the following grammar presents the language of λ-terms.

$$\begin{array}{l} \Lambda ::= x \mid (M\,N) \mid \lambda x.M \mid (M, N) \mid \mathit{fst}\ M \mid \mathit{snd}\ M \\ \quad \text{where } x \in \mathcal{V} \text{ is a variable.} \\ \qquad\quad M, N \in \Lambda \ \text{ are previously constructed } \lambda - \text{terms.} \\ \qquad\quad \mathit{fst}, \mathit{snd} \ \text{ are constant symbols.} \end{array}$$

The term $(M\,N)$ is an *application* (of M to N.) The term $\lambda x.M$ is an *abstraction* and is how functions are defined. The term (M, N) is a pair and *fst* and *snd* are the projections functions for pairs. Eventually we will discuss computation in the language, but for now, we are interested in determining if a term is well typed or not.

We represent the λ-terms in Haskell as follows:

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

Let $\mathcal{VT} = \{\alpha, \beta, \gamma, \alpha_1 \cdots\}$ be an unbounded set of type variables. The language of types is give by the following grammar.

$$\begin{array}{l} T ::= \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\ \quad \text{where } \alpha \in \mathcal{VT} \text{ is a type variable.} \\ \qquad\quad \tau_1, \tau_2 \in T \ \text{ are previously constructed type expressions.} \end{array}$$

The type $\tau_1 \to \tau_2$ denotes the type of functions from type τ_1 to τ_2. The type $\tau_1 \times \tau_2$ denoted the type of pairs where the first element is of type τ_1 and the second is of type τ_2.

The Haskell representation of type expressions can be given as follows:

```
data Op = Arrow | Product    deriving (Eq,Show)
data Type = TVar String | BinType Op Type Type    deriving Eq
```

In this representation the type $\alpha \to (\beta \times \gamma)$ would have the representation

```
BinType Arrow (TVar "a") (BinType Product (TVar "b") (TVar "c"))
```

Note that type variables occur as the leafs in the syntax trees of type expressions.

We choose to instantiate the Haskell type *Type* as an instance of the show type class as follows:

```
instance Show Type where
  show (TVar x) = x
  show (BinType Arrow t1 t2) = "(" ++ show t1 ++ " -¿ " ++ show t2 ++ ")"
  show (BinType Product t1 t2) = "(" ++ show t1 ++ " X " ++ show t2 ++ ")"
```

The list of variables in a type expression can be computed by the following Haskell function.

```
vars :: Type → [String]
vars ty = nub (v ty)
   where v (TVar x) = [x]
         v (BinType op t1 t2) = v t1 ++ v t2
```

Note that *nub* is in the library *Data.List* and eliminates duplicate entries in a list.

2 Substitutions

A *type substitution* is a function of type $\mathcal{VT} \to T$ mapping type variables to types. Since the Haskell datatype *Type* uses `String` to represent variables substitutions can be defined in Haskell by the following type:

```
Substitution :: String → Type
```

We define the following function to recursively apply a substitution to a type.

```
subst :: Substitution → Type → Type
subst s (TVar x) = s x
subst s (BinType op t1 t2) = BinType op (subst s t1) (subst s t2)
```

Note that applying a substitution to a type can only add structure at the leaves *i.e.* applying a substitution can not change the top-level shape of the syntax tree of the type, it can only change a variable which is a leaf.

The identity substitution is the one that maps strings x to types of the tome *TVar x*.

```
idSubst :: Substitution
idSubst x = (TVar x)
```

We have the following theorem.

Theorem 2.1.

$$\forall t : \textit{Type}.\ \ \textit{subst idSubst}\ t = t$$

You could prove this theorem by induction on the structure of the type t.

We can write down substitutions by enumerating the points where they differ from *idSubst*. For example the substitution that behaves like *idSubst* except on variables α and β could be written as

$$s = \{\alpha \mapsto \tau_1, \beta \mapsto \tau_2\}$$

We define the pointwise update of a function as follows:

```
update (x,v) f = (\ y →  if y == x then v else f y)
```

Thus, the substitution s enumerated above could be computed in Haskell by the following expression:

```
update (β, τ2) (update (α,τ2) idSubst)
```

3 Unification

Given a pair of types τ_1 and τ_2 we say they are *unifiable* if there is a substitution (call it σ) such that

$$subst\ \sigma\ \tau_1 = subst\ \sigma\ \tau_2$$

For example, the types $\alpha \to \alpha$ and $\beta \times \gamma \to \delta$ is unifiable by a substitution of the following form:

$$\{\alpha \mapsto \beta \times \gamma,\ \delta \mapsto \alpha\}$$

Type expressions that do not share the same top level shape can not be unified. For example, there is no substitution that unifies $\alpha \to \beta$ with $\alpha \times \beta$ because $\to$ and $\times$ do not match. There is also a problem with a case like α and $\alpha \to \beta$. If you try to figure out a way to do this, you see that α must become something like $\alpha \to \beta$. Consider what happens if we apply the substitution $s = \{\alpha \mapsto (\alpha \to \beta)\}$.

$$\begin{aligned} &subst\ s\ \alpha = (\alpha \to \beta) \\ &subst\ s\ (\alpha \to \beta) = (\alpha \to \beta) \to \beta \end{aligned}$$

So this leads to a kind of loop. We keep expanding α to a term that has α in it so we'll never get a match on both sides. This is called an *occurs check failure*.

The unification algorithm takes two terms and, if they are unifiable, returns a substitution that will make them identical. Here is the unification algorithm described mathematically where $\otimes$ is one of the type constructors $\{\to, \times\}$.

$$\begin{array}{l}
unify\ \alpha\ \alpha\ =\ idSubst \\
unify\ \alpha\ \beta\ =\ update\ (\alpha,\ \beta)\ idSubst \\
unify\ \alpha\ (\tau_1 \otimes \tau_2)\ = \\
\qquad\quad if\ \alpha \in vars(\tau_1 \otimes \tau_2)\ then \\
\qquad\qquad error\ "Occurs\ check\ failure" \\
\qquad\quad else \\
\qquad\qquad update\ (\alpha,\ \tau_1 \otimes \tau_2)\ idSubst \\
unify\ (\tau_1 \otimes \tau_2)\ \alpha\ =\ unify\ \alpha\ (\tau_1 \otimes \tau_2) \\
unify\ (\tau_1 \otimes_1 \tau_2)\ (\tau_3 \otimes_2 \tau_4)\ =\ if\ \otimes_1\ ==\ \otimes_2\ then \\
\qquad\qquad\qquad\qquad\qquad\qquad (subst\ s2)\ .\ s1 \\
\qquad\qquad\qquad\qquad\qquad\quad else \\
\qquad\qquad\qquad\qquad\qquad\qquad error\ "not\ unifiable." \\
\qquad\quad where\ s1 = unify\ \tau_1\ \tau_3 \\
\qquad\qquad\qquad s2 = unify\ (subst\ s1\ \tau_2)\ (subst\ s1\ \tau_4)
\end{array}$$

Exercise 3.1. Using the base code provided on the web-page implement the *unify* function in Haskell.