

## 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  $\lambda$ -calculus with pairs. Let

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

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

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

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

$$\begin{aligned} T ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \\ & \text{where } \alpha \in \mathcal{VT} \text{ is a type variable.} \\ & \tau_1, \tau_2 \in T \text{ are previously constructed type expressions.} \end{aligned}$$

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

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  $\alpha \rightarrow (\beta \times \gamma)$  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  $\mathcal{VT} \rightarrow T$  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 form *TVar x*.

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

We have the following theorem.

**Theorem 2.1.**

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

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  $\alpha$  and  $\beta$  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,τ1) idSubst)
```

### 3 Unification

Given a pair of types  $\tau_1$  and  $\tau_2$  we say they are *unifiable* if there is a substitution (call it  $\sigma$ ) such that

$$\text{subst } \sigma \tau_1 = \text{subst } \sigma \tau_2$$

For example, the types  $\alpha \rightarrow \alpha$  and  $\beta \times \gamma \rightarrow \delta$  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  $\alpha \rightarrow \beta$  with  $\alpha \times \beta$  because  $\rightarrow$  and  $\times$  do not match. There is also a problem with a case like  $\alpha$  and  $\alpha \rightarrow \beta$ . If you try to figure out a way to do this, you see that  $\alpha$  must become something like  $\alpha \rightarrow \beta$ . Consider what happens if we apply the substitution  $s = \{\alpha \mapsto (\alpha \rightarrow \beta)\}$ .

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

So this leads to a kind of loop. We keep expanding  $\alpha$  to a term that has  $\alpha$  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  $\{\rightarrow, \times\}$ .

```

unify  $\alpha$   $\alpha$  = idSubst
unify  $\alpha$   $\beta$  = update ( $\alpha$ ,  $\beta$ ) idSubst
unify  $\alpha$  ( $\tau_1 \otimes \tau_2$ ) =
    if  $\alpha \in \text{vars}(\tau_1 \otimes \tau_2)$  then
        error "Occurs check failure"
    else
        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
    (subst s2) . s1
    else
        error "not unifiable."
where s1 = unify  $\tau_1$   $\tau_3$ 
      s2 = unify (subst s1  $\tau_2$ ) (subst s1  $\tau_4$ )

```

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