Lecture 12

Unification algorithm

On of the key steps involved in type inference is the unification algorithm. Given two type τ_1 and τ_2 a *unifier* if it exists is a type τ such that τ is a specialisation of both τ_1 and τ_2. The *most general unifier* of τ_1 and τ_2, if it exists, is the unifier which is most general, i.e. it is the unifier τ^* of τ_1 and τ_2 such that all other unifiers τ is a specialisation of τ^*. In this lecture we develop an algorithm to compute the most general unifier of two types τ_1 and τ_2.

```
> {- | Module defining the unification algorithm -}
>
> module Unification where
> import Lambda                 -- See the last lecture
> import qualified Data.Map as M -- To define specialisation.
> import Data.Either
> import Data.Maybe
>
> type Map = M.Map
```

Although we defined the unifier as a type, it is convenient when computing the unifier of τ_1 and τ_2 to compute the type specialisation Σ that unifies them to their most general unifier. We will therefore abuse the term most general unifier to also mean this specialisation. Also for simplicity we drop the adjective "most general" and just use unifier to mean the most general unifier.

```
> type Specialisation = Map String Type -- ^ A type specialisation
> type Result         = Either String   -- ^ The error message if it failed
>                        Specialisation   -- ^ The unifier
>
> unify               :: Type -> Type -> Result
```

A type specialisation is captured by a `Map` from type variable names to the corresponding specialisation. We given an function to compute $\tau[\Sigma]$ given a specialisation Σ.

```
> specialise :: Specialisation -> Type -> Type
> specialise sp t@(TV x) = maybe t (specialise sp) $ M.lookup x sp
> specialise sp (TA s t) = TA (specialise sp s) (specialise sp t)
> specialise sp  i       = i
```

We generalise the unification in two ways:

1. We consider unification of types under a particular specialisation Σ. The unifier of τ_1 and τ_2 *under the specialisation* Σ is nothing but the unifier of $\tau_1[\Sigma]$ and $\tau_2[\Sigma]$. The unification of two types can then be seen as unification under empty specialisation.

```
> genUnify  :: Specialisation -> Type -> Type -> Result
> unify     = genUnify M.empty
```

2. Instead of considering unifiers of pairs of types τ_1 and τ_2, we consider the simultaneous unifier of a sequence of pairs $\{(\tau_1, \sigma_1), \ldots, (\tau_n, \sigma_n)\}$. A unifier of such a sequence is a specialisation Σ such that $\tau_i[\Sigma] = \sigma_i[\Sigma]$ for all $1 \leq i \leq n$. Of course we want to find the most general of such unifier. We call this function `genUnify'`. Given, `genUnify` it is easy to define `genUnify'`.

```
> genUnify'  :: Specialisation  -- ^ pair of types to unify
>            -> [(Type,Type)]   -- ^ the specialisation to unify under
>            -> Result          -- ^ the resulting unifier
> genUnify' = foldl fld . Right
>           where fld (Right sp) (tau,sigma) = genUnify sp tau sigma
>                 fld err            _       = err
```

What is left is the definition of `genUnify`.

```
> genUnify sp (TV x)   t       = unifyV sp x t
> genUnify sp t        (TV x)  = unifyV sp x t
> genUnify sp INTEGER  t       = unifyI sp t
> genUnify sp t        INTEGER = unifyI sp t
> genUnify sp ap1      ap2     = unifyA sp ap1 ap2
>
```

The order of the pattern matches are important. For example notice that in line 6, both `ap1` and `ap2` have to be an arrow type (why?) Also in lines 3 and 4, t can either be `INTEGER` or an arrow type.

So our rules of unification can are split into unifying a variable α with a type τ, a constant types (here Integer) with a type τ and unifying two arrow types. We capture these rules with functions `unifyV`, `unifyI` and `unifyA` respectively.

```
> unifyV :: Specialisation -> String -> Type -> Result
> unifyI :: Specialisation -> Type   -> Result
> unifyA :: Specialisation -> Type   -> Type -> Result
```

12.1 Unifying a variable and a type

Unification of variables can be tricky since specialisations are involved. Firstly, notice that if α occurs in a type τ, we cannot unify α with τ unless τ is α itself. We first give a function to check this.

```
> occurs :: Specialisation -> String -> Type -> Bool
> occurs sp x (TV y)   | x == y    = True
>                      | otherwise = maybe False (occurs sp x) $ M.lookup y sp
> occurs sp x (TA s t) = occurs sp x s || occurs sp x t
> occurs sp x  _       = False
```

The rules of unification of a variable α and a type τ are thus

1. If α has a specialisation σ, unify σ and tau,
2. If τ is α or can be specialised to α then we already have the specialisation.
3. If α is unspecialised then specialise it with τ provided there occurs no recursion either directly or indirectly.

```
> unifyV  sp x t = maybe (unifyV' sp x t) (genUnify sp t) $ M.lookup x sp
>
> unifyV' sp x INTEGER                 = Right $ M.insert x INTEGER sp
> unifyV' sp x var@(TV y)  | x == y    = Right $ sp
>                          | otherwise = maybe (Right $ M.insert x var sp)
>                                              (unifyV' sp x)
>                                              $ M.lookup y sp
> unifyV' sp x ap@(TA s t) | occurs sp x s = failV sp x ap
>                           | occurs sp x t = failV sp x ap
>                           | otherwise  = Right $ M.insert x ap sp
>
> failV sp x ap = Left $ unwords [ "Fail to unify", x, "with" , ppSp sp ap
>                                , ": recursion detected."
>                                ]
>
```

Notice here that `M.lookup x sp` returns `Nothing` if `x` is not specialised under the specialisation `sp`. The function $unifyV'$ unifies only an unspecialised variable with τ.

12.2 Unifying an integer and type

Notice here that the type τ is not a variable. Then it can only be Integer or an arrow type.

```
> unifyI sp INTEGER = Right sp
> unifyI sp t       = Left $ unwords [ "Failed to unify Integer with type"
>                                    , ppSp sp t
>                                    ]
```

12.3 Unifying two arrow types

```
> unifyA sp ap1@(TA a b) ap2@(TA c d) = either errmsg Right
>                                    $ genUnify' sp [(a,c),(b,d)]
>          where errmsg str = Left $ unwords [ "while unifying"
>                                            , ppSp sp ap1, "and"
>                                            , ppSp sp ap2
>                                            ] ++ "\n" ++ str
```

12.4 Helper functions.

We now document the helper functions used in the unification algorithm. First we give an algorithm to *pretty print* types. This makes our error messages more readable. Notice that this version does not put unnecessary brackets.

```
> -- | Pretty print a type
>
> pp :: Type -> String
> pp INTEGER  = "Integer"
> pp (TV x)   = x
> pp (TA s t) = bracket s ++ " -> " ++ pp t
>        where bracket t@(TA r s) = "(" ++ pp t ++ ")"
>              bracket s          = pp s
>
> -- | Pretty print a specialised type
> ppSp :: Specialisation -> Type -> String
> ppSp sp = pp . specialise sp
```

12.5 Testing this code using ghci

Since the lecture used the module of the previous lecture you need to give the following commandline arguments.

```
$ ghci src/lectures/Unification-algorithm.lhs src/lectures/Towards-type-inference.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 2] Compiling Lambda             ( src/lectures/Towards-type-inference.lhs, interpreted )
[2 of 2] Compiling Unification        ( src/lectures/Unification-algorithm.lhs, interpreted )
Ok, modules loaded: Unification, Lambda.
*Unification> let [a,b,c,d,e] = map TV $ words "a b c d e"
*Unification> unify (TA INTEGER a) b
Loading package array-0.3.0.2 ... linking ... done.
Loading package containers-0.4.0.0 ... linking ... done.
Right (fromList [("b",TA INTEGER (TV "a"))])
*Unification> unify (TA INTEGER a) (TA b (TA b c))
Right (fromList [("a",TA (TV "b") (TV "c")),("b",INTEGER)])
*Unification>
*Unification> let t = (TA a b)
*Unification> unify (TA INTEGER a) (TA c t)
Left "while unifying Integer -> a and c -> a -> b\nFail to unify a with a -> b : recursion detect
*Unification> let Left l = unify (TA INTEGER a) (TA c t)
*Unification> putStrLn l
while unifying Integer -> a and c -> a -> b
Fail to unify a with a -> b : recursion detected.
```

```
*Unification>
```