MP 5 – A Unification-based Type Inferencer

CS 421 – Spring 2016
Revision 1.0

Assigned April 15, 2016
Due April 27, 2016, 11:59:59pm

1 Change Log

1.0 Initial Release.

2 Objectives

Your objective for this assignment is to understand the details of the basic algorithm for first order unification.

In this MP you will implement one step of type inferencing: the unification algorithm *unify* that solves constraints generated by the inferencer.

It is recommended that before or in tandem with completing this assignment, you go over lecture materials covering type inference and unification to have a good understanding of how types are inferred.

3 Our Datatypes

Below is the datatype that we'll be using throughout this MP. `TyCon` represents the types that we have in our arsenal - in particular, `BoolTy` to represent booleans, `IntTy` to represent integers, `StringTy` to represent strings, `PairTy` to represent a pair of elements (also known as a tuple), `FnTy` to represent a function, `ListTy` to represent a list type, and finally, `TyVar` to represent types that we (potentially) need to figure out the type for. Note that `TyVar`s take integers; this is because we're treating them as variable "ids" behind the scenes - or alternatively, you can think of them as v_i, where i is the integer fed into the `TyVar`. They just represent variables, nothing special.

```
data TyCon = BoolTy
           | IntTy
           | StringTy
           | PairTy TyCon TyCon
           | FnTy TyCon TyCon
           | ListTy TyCon
           | TyVar Integer
  deriving Eq

instance Show TyCon where
  show BoolTy = "Bool"
  show IntTy = "Integer"
  show StringTy = "String"
  show (PairTy a b) = "(" ++ (show a) ++ ", " ++ (show b) ++ ")"
  show (FnTy a b) = (show a) ++ " -> " ++ (show b)
  show (ListTy a) = "[" ++ (show a) ++ "]"
  show (TyVar s) = "v" ++ (show s)
```

We've (as you can see above) already defined their `Show` instances, so you won't have to worry about that. If you're still confused, take a look at them as they might help you understand what a `TyCon` represents.

We have also defined two type synonyms to help make things a little clearer:

```
type SubstEnv = (H.HashMap Integer TyCon)
type EqnSet = [(TyCon, TyCon)]
```

`SubstEnv` is the environment that holds all our substitutions (defined below). Our goal in using a `SubstEnv` is to take a `TyVar` and map it to a concrete type (i.e., not containing any `TyVars`) - be it a `FnTy` or a `BoolTy`, for example.

`EqnSet` is the set of equations (or *constraints*) that our unification problem is based upon. Recall that unification problems are, at their core, solving a series of constraints. If this is confusing, take a break here to look at the Unification lecture slides.

4 Substitutions

A substitution is a mapping from a type variable to a term - which, in our case, is a mapping from a `TyVar` to a concrete type. In other words, for every instance of the given `TyVar`, we replace it with the corresponding concrete type. In our world, substitutions are stored in a hashmap where our key is the integer given in our `TyVar` and the value is the given concrete type. You can think about them as a set of equations of the form $\{v_n \mapsto \beta\}$, where v_n is a type variable (e.g. `(TyVar n)`) and β a concrete type.

In this MP, you will implement a function `substFun` that takes a substitution and returns a *substitution function*, which is a function that takes a type variable (i.e. a `TyVar`) as input and returns the replacement (concrete) type as output, as defined by the substitution. If it is the case that the given type variable doesn't exist in the given substitution, it is assumed that the type variable has the identity function applied - that is, the variable is substituted with itself.

Your function `substFun` should, given a substitution, return the function it represents. This should be a function that takes a `SubstEnv` and a `TyCon` and returns a `TyCon`. In particular, it should only handle the case where the given `TyCon` is a `TyVar`. (We've handled the other cases by throwing an error.)

In particular, suppose

```
phi = (H.insert 5 (FnTy BoolTy (TyVar 2)) H.empty)
```

which is considered to represent the substitution function

$$\phi(\tau_i) = \begin{cases} \texttt{bool} \to \tau_2 & \text{if } i = 5 \\ \tau_i & \text{otherwise} \end{cases}$$

```
*Main> :t substFun
substFun :: SubstEnv -> TyCon -> TyCon
*Main> substFun phi (TyVar 1)
v1
*Main> substFun phi (TyVar 5)
(Bool -> v2)
```

We can also *lift* a substitution to operate on types. A substitution ϕ, when lifted, replaces all the type variables occurring in its input type with the corresponding types in the given substitution. In this MP, you will implement a function `monoTyLiftSubst` for lifting substitutions to generic `TyCon`s.

```
*Main> :t monoTyLiftSubst
monoTyLiftSubst :: SubstEnv -> TyCon -> TyCon
*Main> monoTyLiftSubst phi (FnTy (TyVar 1) (TyVar 5))
(v1 -> (Bool -> v2))
```

5 Unification

The unification algorithm takes a set of pairs of types that are supposed to be equal. A system of constraints looks like the following:

$$\{(s_1, t_1), (s_2, t_2), ..., (s_n, t_n)\}$$

Each pair is called an *equation*. A (lifted) substitution ϕ *solves* an equation (s, t) if $\phi(s) = \phi(t)$. It solves a constraint set if $\phi(s_i) = \phi(t_i)$ for every (s_i, t_i) in the constraint set. The unification algorithm will return a substitution that solves the given constraint set (if a solution exists).

Recall from lecture that the unification algorithm consists of four transformations. These transformations can be expressed in terms of how an action on the first element of the unification problem affects the remaining elements.

In particular, given a constraint set C:

1. If C is empty, return the identity substitution.
2. If C is not empty, pick an equation $(s, t) \in C$. Let C' be $C \setminus \{(s, t)\}$.
   (a) **Delete rule:** If s and t are are equal, discard the pair, and unify C'.
   (b) **Orient rule:** If t is a variable, and s is not, then discard (s, t), and unify $\{(t, s)\} \cup C'$.
   (c) **Decompose rule:** If s is a `PairTy`, `FnTy`, or `ListTy`, $s =$ `Con` $s_1 s_2$ and $t =$ `Con` $t_1 t_2$ (where `Con` is the corresponding constructor), then discard (s, t), and unify $C' \cup \{(s_1, t_1), (s_2, t_2)\}$
   (d) **Eliminate rule:** If s is a variable, and s does not occur in t, substitute s with t in C' to get C''. Let ϕ be the substitution resulting from unifying C''. Return ϕ updated with $s \mapsto \phi(t)$.
   (e) If none of the above cases apply, it is a unification error (your `unify` function should return `Nothing` in this case).

6 Handing In

You must modify `app/Unifier.hs`, as that is the only file we will be grading. You are allowed to (and in fact, are encouraged to) write your own tests. You can do so in `test/Tests.hs`. We have also provided you with a test suite that you can run via `stack test`.

Like previous MPs, you are not allowed to import other modules. You are allowed to use functions given to you in `Prelude`. You are encouraged to write helper functions, and are allowed to use functions written in earlier parts of this MP in later parts of the MP.

7 Problems

Problem 1. (5 pts) Implement the `substFun` function as described in Section 4.

```
*Main> :t substFun
substFun :: SubstEnv -> TyCon -> TyCon
*Main> substFun phi (TyVar 1)
v1
*Main> substFun phi (TyVar 5)
(Bool -> v2)
```

Problem 2. (5 pts) Implement the `monoTyLiftSubst` function as described in Section 4.

```
*Main> :t monoTyLiftSubst
monoTyLiftSubst :: SubstEnv -> TyCon -> TyCon
*Main> monoTyLiftSubst phi (FnTy (TyVar 1) (TyVar 5))
(v1 -> (Bool -> v2))
```

Problem 3. (5 pts) Write a function `occurs :: TyCon -> TyCon -> Bool`. The first argument of the function is a `TyVar`, the second a target expression, with the output indicating whether the variable occurs within the target. You may assume your input will be a `TyVar`.

```
*Main> :t occurs
occurs :: TyCon -> TyCon -> Bool
*Main> occurs (TyVar 0) (FnTy (TyVar 0) (TyVar 0))
True
*Main> occurs (TyVar 0) (FnTy (TyVar 1) (TyVar 2))
False
```

Problem 4. (65 pts) Now, you're ready to write the unification function. We will represent constraint sets via our list of equations, `EqnSet`. If there exists a solution to a set of constraints (i.e., a substitution that solves the set), your function should return `Just` of that substitution. Otherwise it should return `Nothing`. Here's a sample run.

```
*Main> :t unify
unify :: EqnSet -> Maybe SubstEnv
*Main > :{
let phi' =
           (unify [(TyVar 0, ListTy IntTy), ((FnTy (TyVar 0) (TyVar 0)),
            (FnTy (TyVar 0) (TyVar 1)))])
:}
*Main> phi'
Just (fromList [(0,[Integer]),(1,[Integer])])
*Main> substFun ((\(Just x) -> x) phi') (TyVar 1)
[Integer]
*Main> substFun ((\(Just x) -> x) phi') (TyVar 2)
v2
```

Hint: You will find the functions you implemented in Problems 1, 2, and 3 very useful in some rules.

Point distribution: Delete is 6 pts, Orient is 6 pts, Decompose is 16 pts, Eliminate is 37 pts. This distribution is approximate. Correctness of one part impacts the functioning of other parts.