MP 6 – Unification Algorithm

CS 421 – Fall 2011

Revision 1.0

Assigned October 4, 2011
Due October 18, 2011, at 23:59pm
Extension 48 hours (20% penalty)

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.

3 Preliminaries

In MP5 you have implemented the first part of the type inferencer for the PicoML language. In this MP we will implement the second step of the inferencer: the algorithm that solves the constraints. This process is known as unification. In MP5 you were given the unifier as a black box that gave you the solution when fed with the constraints generated by your implementation.

It is recommended that you go over lecture notes covering type inference and unification to have a good understanding of how types are inferred.

4 Given Code

As usual, you are given some code for your use in the `Mp6common` module. This module includes the following data types to represent types of PicoML. You may use any of the functions in the List module for this MP.

```
type typeVar = int

type constTy = {name : string; arity : int}

type monoTy = TyVar of typeVar | TyConst of (constTy * monoTy list)
```

Note that we are defining a language (PicoML) using a language (OCaml). To represent the types, expressions and values in PicoML, we will use OCaml data types, which have their own OCaml types. Do not confuse OCaml types with PicoML types. When we refer to PicoML types, we will simply use the term "type". Should we need to refer to OCaml types, we will use "meta-type". E.g: PicoML expression 10 has the *type* `TyConst(name = "int"; arity = 0,[])`, which has the *meta-type* `monoTy`.

Recall that you can use `string_of_monoTy` from `Mp6common` to convert your types into a readable concrete syntax for types.

5 Substitutions

A substitution function returns a replacement type for a given type variable. Here, we use the the type `int` for type variables, which we give the synonym `typeVar`. Our substitutions will have the type `(typeVar * monoTy) list`. The first component of a pair is the index of a type variable. The second is the type that should be substituted for that type variable. If an entry for a type variable index does not exist in the list, the identity substitution should be assumed for that type variable (i.e. the variable is substituted with itself). For instance, the substitution

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

would be implemented as

```
# let phi = [(5, mk_fun_ty bool_ty (TyVar(2)))];;
val phi : (typeVar * monoTy) list =
  [(5,
    TyConst
     ({name = "->"; arity = 2},
      [TyConst ({name = "bool"; arity = 0}, []); TyVar 2]))]
```

Throughout this MP you may assume that substitutions we work on are always well-structured: there are no two pairs in a substitution list with the same index.

Given a substitution list, we can convert it to a function that takes an `typeVar` and returns a `monoTy`.

```
# let subst_fun s = ...
val subst_fun : (typeVar * monoTy) list -> typeVar -> monoTy = <fun>
# let subst = subst_fun phi;;
val subst : typeVar -> monoTy = <fun>
# subst 1;;
- : monoTy = TyVar 1
# subst 5;;
- : monoTy =
TyConst
 ({name = "->"; arity = 2},
  [TyConst ({name = "bool"; arity = 0}, []); TyVar 2])
```

We can also *lift* a substitution to operate on types. So a substitution ϕ, when lifted, replaces all the type variables occurring in its input type with the corresponding types.

```
# let rec monoTy_lift_subst s = ...
val monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy = <fun>
# let lifted_sub = monoTy_lift_subst phi;;
val lifted_sub : monoTy -> monoTy = <fun>
# lifted_sub (TyConst ({name = "->"; arity = 2}, [TyVar 1; TyVar 5]));;
- : monoTy =
TyConst
 ({name = "->"; arity = 2},
  [TyVar 1;
   TyConst
    ({name = "->"; arity = 2},
     [TyConst ({name = "bool"; arity = 0}, []); TyVar 2])])
```

You will need to implement the `subst_fun` and `monoTy_lift_subst` functions (see the Problems section).

6 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 set

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

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).

You will remember 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.

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 $s =$ `TyConst` $(name, [s_1; \ldots; s_n])$ and $t =$ `TyConst` $(name, [t_1; \ldots; t_n])$, then discard (s, t), and unify $C' \cup \bigcup_{i=1}^{n} \{(s_i, t_i)\}$.
   (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 the `None` option in this case).

In our system, function, integer, list, etc. types are the terms; `TyVars` are the variables.

7 Problems

1. (0 pts) Make sure that you understand the `monoTy` data type. You should be comfortable with how to represent a type using `monoTy`. MP5 should have given you enough practice of this. If you still do not feel fluent enough, do the exercise below. This exercise will not be graded; it is intended to warm you up.

   In each item below, define a function `asMonoTy`X : `unit -> monoTy` that returns the `monoTy` representation of the given type. In these types, $\alpha, \beta, \gamma, \delta, ...$ are type variables.

   - `bool` $\rightarrow$ `int list`

```
#  let asMonoTy1 () = ...
val asMonoTy1 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy1());;
- : string = "bool -> int list"
```

   - $\alpha \rightarrow \beta \rightarrow \delta \rightarrow \gamma$

```
# let asMonoTy2 () = ...
val asMonoTy2 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy2());;
- : string = "'e -> 'd -> 'c -> 'b"
```

   - $\alpha \rightarrow (\beta *$ `int`$)$`list`

```
# let asMonoTy3 () = ...
val asMonoTy3 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy3());;
- : string = "'g -> ('f * int) list"
```

- ($\texttt{string} * (\beta\ \texttt{list} \rightarrow \alpha)$)

  ```
  #  let asMonoTy4 () = ...
  val asMonoTy4 : unit -> monoTy = <fun>
  # string_of_monoTy(asMonoTy4());;
  - : string = "string * 'i list -> 'h"
  ```

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

   ```
   # let subst_fun s = ...
   val subst_fun : (typeVar * monoTy) list -> typeVar -> monoTy = <fun>
   # let subst = subst_fun [(5, mk_fun_ty bool_ty (TyVar(2)))];;
   val subst : typeVar -> monoTy = <fun>
   # subst 1;;
   - : monoTy = TyVar 1
   # subst 5;;
   - : monoTy =
   TyConst
    ({name = "->"; arity = 2},
     [TyConst ({name = "bool"; arity = 0}, []); TyVar 2])
   ```

3. (4 pts) Implement the `monoTy_lift_subst` function as described in Section 5.

   ```
   # let rec monoTy_lift_subst s = ...
   val monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy = <fun>
   # monoTy_lift_subst [(5, mk_fun_ty bool_ty (TyVar(2)))]
                 (TyConst ({name = "->"; arity = 2}, [TyVar 1; TyVar 5]));;
   - : monoTy =
   TyConst
    ({name = "->"; arity = 2},
     [TyVar 1;
      TyConst
       ({name = "->"; arity = 2},
        [TyConst ({name = "bool"; arity = 0}, []); TyVar 2])])
   ```

4. (5 pts) Write a function `occurs : typeVar -> monoTy -> bool`. The first argument is the integer component of a `TyVar`. The second is a target expression. The output indicates whether the variable occurs within the target.

   ```
   # occurs 0 (TyConst ({name = "->"; arity = 2}, [TyVar 0; TyVar 0]));;
   - : bool = true
   # occurs 0 (TyConst ({name = "->"; arity = 2}, [TyVar 1; TyVar 2]));;
   - : bool = false
   ```

5. (64 pts) Now you are ready to write the unification function. We will represent constraint sets simply by lists. If there exists a solution, your function should return `Some` of that substitution. Otherwise it should return `None`. Here's a sample run.

```
# let rec unify eqlst = ...
val unify : (monoTy * monoTy) list -> (typeVar * monoTy) list option = <fun>
# let Some(subst) =
   unify [(TyVar 0,
            TyConst ({name = "list"; arity = 1},
              [TyConst ({name = "int"; arity = 0}, [])]));
           (TyConst ({name = "->"; arity = 2}, [TyVar 0; TyVar 0]),
            TyConst ({name = "->"; arity = 2}, [TyVar 0; TyVar 1]))];;
... (* Warning message suppressed *)
val subst : (typeVar * monoTy) list =
  [(0,
    TyConst
     ({name = "list"; arity = 1}, [TyConst ({name = "int"; arity = 0}, [])]));
   (1,
    TyConst
     ({name = "list"; arity = 1}, [TyConst ({name = "int"; arity = 0}, [])]))]
# subst_fun subst 0;;
- : monoTy =
TyConst
 ({name = "list"; arity = 1}, [TyConst ({name = "int"; arity = 0}, [])])
# subst_fun subst 1;;
- : monoTy =
TyConst
 ({name = "list"; arity = 1}, [TyConst ({name = "int"; arity = 0}, [])])
# subst_fun subst 2;;
- : monoTy = TyVar 2
```

Hint: You will find the functions you implemented in Problems 2,3,4 very useful in some rules.

Point distribution: Delete is 6 pts, Orient is 6 pts, Decompose is 16 pts, Eliminate is 36 pts. This distibution is approximate. Correctness of one part impacts the functioning of other parts. Machine grader will not be able to detect this, however the human grader will fix propagating errors.

6. **Extra Credit (10 pts)** Two types τ_1 and τ_2 are equivalent if there exist two substitutions ϕ_1, ϕ_2 such that $\phi_1(\tau_1) = \tau_2$ and $\phi_2(\tau_2) = \tau_1$. Write a function `equiv_types : monoTy -> monoTy -> bool` to indicate whether the two input type expressions are equivalent.

   Hint: find τ_3 such that τ_1 is equivalent to τ_3 and τ_2 is also equivalent to τ_3 by reducing τ_1 and τ_2 to a canonical form.

```
# let equiv_types ty1 ty2 = ...
val equiv_types : monoTy -> monoTy -> bool = <fun>
# equiv_types
    (TyConst
      ({name = "->"; arity = 2},
      [TyVar 4; TyConst ({name = "->"; arity = 2}, [TyVar 3; TyVar 4])]))
    (TyConst
      ({name = "->"; arity = 2},
      [TyVar 3; TyConst ({name = "->"; arity = 2}, [TyVar 4; TyVar 3])]));;
- : bool = true
# equiv_types
    (TyConst
      ({name = "->"; arity = 2},
      [TyVar 4; TyConst ({name = "->"; arity = 2}, [TyVar 3; TyVar 4])]))
    (TyConst
      ({name = "->"; arity = 2},
      [TyVar 4; TyConst ({name = "->"; arity = 2}, [TyVar 3; TyVar 2])]));;
- : bool = false
```