MP 6 – Unification Algorithm

CS 421 – su 2011
Revision 1.0

Assigned July 10, 2011
Due July 31, 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

Recall that type inferencing has two main steps:

1. Build a system of constraints.
2. Solve the system of constraints.

In MP5 you have implemented the first part of the type inferencer for the MicroML 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 set of 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. In this assignment, you will be asked to implement unification for a very generic setting of terms built from constructors and variables. The OCaml type representing just generic terms is:

```
type 'a term = TmVar of int | TmConst of ('a * 'a term list)
```

Here, variables are represented by `TmVar`(n), for integers n. The type is parametrized by a type of constructors. In the case we are using our unification algorithm for solving constraints from type inferencing, the constructors should represent the constructors for types. In examples, we will use the following type to represent those constructors:

```
type constTy = BoolTy | IntTy | RealTy | StringTy | UnitTy
             | FunTy | PairTy | ListTy | OptionTy
```

In this setting, we would represent the type `int -> bool` by `TmConst(FunTy, [TmConst(IntTy,[]); TmConst(BoolTy,[])])`. You can use `print_term` from `Mp6common` to neatly display terms for your convenience.

You may use any of the functions in the List module for this MP.

5 Substitutions

An important concept in any setting having terms with variables is that of a *substitution* of terms for variables and *substituting* a term for a variable. It is often important for algorithms to have a data structure representing the simultaneous substitution of terms for variables, and for that we will use the following type:

```
type 'a substitution = (int * 'a term) list
```

The main function you will implement in this MP is `unify`, which will return as its answer a substitution option.

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

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

would be implemented as

```
# let phi = [(5, TmConst(FunTy, [TmConst(BoolTy,[]); TmVar 2]))];;
val phi : (int * constTy term) list =
  [(5, TmConst (FunTy, [TmConst (BoolTy, []); TmVar 2]))]
```

Throughout this MP, you may assume that the substitutions on which we work 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 `int` and returns an `'a term`.

```
# let subst_fun s = ...
val subst_fun : (int * 'a term) list -> int -> 'a term = <fun>
# let subst = subst_fun phi;;
val subst : int -> constTy term = <fun>
# subst 1;;
- : constTy term = TmVar 1
# subst 5;;
- : constTy term = TmConst(FunTy, [TmConst(BoolTy, []); TmVar 2])
```

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

```
# let rec lift_subst s = ...
val lift_subst : (int * 'a term) list -> 'a term -> 'a term = <fun>
# let lifted_sub = lift_subst phi;;
val lifted_sub : constTy term -> constTy term = <fun>
# lifted_sub (TmConst (FunTy, [TmVar 1; TmVar 5]));;
- : constTy term =
TmConst (FunTy, [TmVar 1; TmConst (FunTy, [TmConst (BoolTy, []); TmVar 2])])
```

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

6 Unification

The unification algorithm takes a set of pairs of terms 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 $s =$ `TmConst` $(name, [s_1; \ldots; s_n])$ and $t =$ `TmConst` $(name, [t_1; \ldots; t_n])$, 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; `TmVar`s are the variables.

7 Convertion Functions

Out unification deals with generic terms, but to use it in MicroML, we need to be able to force it to work with MicroML types. We need to be able to convert `constTy term` to `tyExp` and vice versa. Recollect from MP5 the type `tyExp` for representing MicroML types:

```
type tyExp =
     VarType of int | BoolType | IntType | RealType | StringType | UnitType
   | FunType of tyExp * tyExp | PairType of tyExp * tyExp | ListType of tyExp
   | OptionType of tyExp
```

That is why we need these two function:

```
let rec term_of_tyExp (ty : tyExp) = ...
let rec tyExp_of_term (tm : constTy term) = ...
```

You will write these functions as a part of this MP.

8 Problems

1. (0 pts) Make sure that you understand the `constTy term` data type. You should be comfortable with how to represent a type using `constTy term`. This exercise will not be graded; it is intended to warm you up.

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

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

```
#  let asTerm1 () = ...
val asTerm1 : unit -> constTy term = <fun>
# print_newline(print_term(asTerm1()));;
bool -> int list
- : unit = ()
```

- $\alpha \to \beta \to \delta \to \gamma$

```
#  let asTerm2 () = ...
val asTerm2 : unit -> constTy term = <fun>
# print_newline(print_term(asTerm2()));;
'a -> 'b -> 'd -> 'c
- : unit = ()
```

- $(\alpha \to (\beta * \texttt{int})\texttt{list})$

```
#  let asTerm3 () = ...
val asTerm3 : unit -> constTy term = <fun>
# print_newline(print_term(asTerm3()));;
'a -> ('b * int) list
- : unit = ()
```

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

```
#  let asTerm4 () = ...
val asTerm4 : unit -> constTy term = <fun>
# print_newline(print_term(asTerm4()));;
string * 'b list -> 'a
- : unit = ()
```

2. (0 pts) Make sure that you understand the `tyExp` data type. You should be comfortable with how to represent a type using `tyExp`. 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 `asTyExp`X`:unit -> tyExp` that returns the `tyExp` representation of the given type. In these types, $\alpha, \beta, \gamma, \delta, ...$ are type variables.

   - $\texttt{bool} \to \texttt{int list}$

```
#  let asTyExp1 () = ...
val asTyExp1 : unit -> tyExp = <fun>
# print_newline(print_tyExp (asTyExp1()));;
bool -> int list
- : unit = ()
```

   - $\alpha \to \beta \to \delta \to \gamma$

```
#  let asTyExp2 () = ...
val asTyExp2 : unit -> tyExp = <fun>
# print_newline(print_tyExp (asTyExp2()));;
'a -> 'b -> 'd -> 'c
- : unit = ()
```

   - $(\alpha \to (\beta * \texttt{int})\texttt{list})$

```
#  let asTyExp3 () = ...
val asTyExp3 : unit -> tyExp = <fun>
# print_newline(print_tyExp (asTyExp3()));;
'a -> ('b * int) list
- : unit = ()
```

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

```
#  let asTyExp4 () = ...
val asTyExp4 : unit -> tyExp = <fun>
# print_newline(print_tyExp (asTyExp4()));;
string * (('b list) -> 'a)
- : unit = ()
```

3. (3 pts) Implement the `term_of_tyExp` function as described in Section 7.

```
# let rec term_of_tyExp t = ...
val term_of_tyExp : tyExp -> constTy term = <fun>
# term_of_tyExp (FunType(BoolType, IntType));;
- : constTy term = TmConst (FunTy, [TmConst (BoolTy, []); TmConst (IntTy, [])])
```

4. (3 pts) Implement the `tyExp_of_term` function as described in Section 7.

```
# let rec tyExp_of_term t = ...
val tyExp_of_term : constTy term -> tyExp = <fun>
# tyExp_of_term (TmConst (PairTy, [TmVar 2; TmConst (RealTy, [])]));;
- : tyExp = PairType (VarType 2, RealType)
```

Since terms are more general than `tyExp`, some of them do not have a corresponding MicroML type. You will not be tested on these cases.

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

```
# let subst_fun s = ...
val subst_fun : (int * 'a term) list -> int -> 'a term = <fun>
# let subst = subst_fun [(5, TmConst(FunTy, [TmConst(BoolTy,[]); TmVar 2]))];;
val subst : int -> constTy term = <fun>
# subst 5;;
- : constTy term = TmConst (FunTy, [TmConst (BoolTy, []); TmVar 2])
```

6. (4 pts) Implement the `lift_subst` function as described in Section 5.

```
# let rec lift_subst s = ...
val lift_subst : (int * 'a term) list -> 'a term -> 'a term = <fun>
# lift_subst [(5, TmConst(FunTy, [TmConst(BoolTy,[]); TmVar 2]))]
             (TmConst(FunTy, [TmVar 1; TmVar 5]));;
- : constTy term =
TmConst (FunTy, [TmVar 1; TmConst (FunTy, [TmConst (BoolTy, []); TmVar 2])])
```

7. (5 pts) Write a function `occurs : int -> 'a term -> bool`. The first argument is the integer component of a `TmVar`. The second is a target expression. The output indicates whether the variable occurs within the target.

```
# occurs 0 (TmConst(FunTy, [TmVar 0; TmVar 0]));;
- : bool = true
```

8. (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 : ('a term * 'a term) list -> 'a substitution option = <fun>
# let Some(subst) =
   unify [(TmVar 0,
             TmConst (ListTy, [TmConst (IntTy, [])]));
          (TmConst (FunTy, [TmVar 0; TmVar 0]),
             TmConst (FunTy, [TmVar 0; TmVar 1]))];;
... (* Warning message suppressed *)
val subst : constTy substitution =
  [(0, TmConst (ListTy, [TmConst (IntTy, [])]));
   (1, TmConst (ListTy, [TmConst (IntTy, [])]))]
# subst_fun subst 0;;
- : constTy term = TmConst (ListTy, [TmConst (IntTy, [])])
```

Hint: You will find the functions you implemented in Problems 5,6,7 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.

9. **Extra Credit (10 pts)** Two terms 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_terms : 'constTy term -> 'constTy term -> bool` to indicate whether the two input terms 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_terms tm1 tm2 = ...
val equiv_terms : 'a term -> 'a term -> bool = <fun>
# equiv_terms
    (TmConst(FunTy, [TmVar 4; TmConst(FunTy, [TmVar 3; TmVar 4])]))
    (TmConst(FunTy, [TmVar 3; TmConst(FunTy, [TmVar 4; TmVar 3])]));;
- : bool = true
```