
MP 4 – Unification Algorithm

CS 421 – Fall 2006

Revision 4.1

Assigned September 28, 2006

Due October 5, 2006, 11 AM

Extension 48 hours (20% penalty)

1 Change Log

4.2 Initial Release.

4.1 Modified on Friday, September 29, 10:00 PM.

2 Preliminaries

This MP is part of a series of assignments about defining a simple functional programming language. We shall refer to this language as the *object language*. Here we only define its types and you are required to solve problems regarding type equations and unification.

We shall define the types of the object language using an OCaml recursive datatype, whose name is `tp`. Notice the important distinction between the OCaml types (such as `int`, `bool` and `tp` below), and the object-language types, which are elements (that is, pieces of data) of type `tp`. To avoid confusion, we shall refer to the elements of `tp` as “types”, while if necessary using the term “meta-type” for OCaml types.

You are given the following meta-types:

```
type tp = TpVar of int | Bool | Nat | String | Fnc of (tp * tp) |
         Prod of (tp * tp) | List of tp
type tpEquation = Eq of (tp * tp)
type substitution = int -> tp
```

Above `tp` is the meta-type of types. A type is either a type variable `TpVar n` (indexed by an integer n), or `Bool`, or `Nat`, or `String`, or a functional type $t \rightarrow t'$ (generated with the constructor `Fnc`), or a product $t \times t'$ (generated with the constructor `Prod`), or the type `List t`.

The meta-type `tpEquation` contains type equations `Eq(t1, t2)`. The meta-type `substitution` contains substitutions, encoded as mappings between numbers (that is, indexes of type variables) and types.

We will be using the following function for testing some of your code:

```
let getProper opt =
  match opt with
  | Some x -> x |
  | None -> raise (Failure "")
```

`getProper` takes an argument `opt` of a polymorphic option type and returns `x` if `opt` has the form `Some x`, otherwise raises an exception.

3 Problems

In this MP, you are allowed to use any library functions you may wish, as well as any function previously defined in this or in a previous MP.

1. Write a function `subst` taking a number and two types such that `subst n t t'` returns the result of substituting type variable `TpVar (n)` with type `t` in type `t'`.

```
# let rec subst n t t' = ...
val subst : int -> tp -> tp -> tp = <fun>
# subst 0 (TpVar 1) (TpVar 0) ;;
- : tp = TpVar 1
# subst 1 (Prod(Fnc (TpVar 1, TpVar 0), String))
(Fnc(List (Fnc ((TpVar 1), TpVar 2)), Prod(List (TpVar 1), Bool))) ;;
- : tp =
Fnc
(List (Fnc (Prod (Fnc (TpVar 1, TpVar 0), String), TpVar 2)),
Prod (List (Prod (Fnc (TpVar 1, TpVar 0), String), Bool)) ) ;;
```

2. Write a function `substInEqList` taking a number, a type and a list of type equations such that `substInEqList n t eqs` substitutes the type variable `TpVar (n)` with the type `t` in both types of each type equation in the list `eqs`.

```
# let substInEqList n t eqs = ...
val substInEqList : int -> tp -> tpEquation list -> tpEquation list = <fun>
# substInEqList 0 (TpVar 2) [] ;;
- : tpEquation list = []
# substInEqList 3 (TpVar 2) [Eq(TpVar 3, TpVar 2)] ;;
- : tpEquation list = [Eq (TpVar 2, TpVar 2)]
# substInEqList 0 Bool [Eq(Fnc(Nat, TpVar 0), TpVar 0);
Eq(Prod(TpVar 0, TpVar 1), String) ] ;;
- : tpEquation list =
[Eq (Fnc (Nat, Bool), Bool); Eq (Prod (Bool, TpVar 1), String)]
```

3. Write a function `isInType` that takes a number `n` and a type `t` and checks whether the type variable `TpVar (n)` occurs in `t`.

```
# let rec isInType n t = ...
val isInType : int -> tp -> bool = <fun>
# isInType 24 Bool ;;
- : bool = false
# isInType 13 (TpVar 13) ;;
- : bool = true
# isInType 0 (Fnc(TpVar 2, Prod(List (TpVar 0), Bool))) ;;
- : bool = true
# isInType 0
(Fnc(List (Fnc ((TpVar 34), TpVar 2)), Prod(List (TpVar 12), Bool))) ;;
- : bool = false
```

4. Write a function `isSol` that takes a list of type equations and checks whether it is in *solution form*. By definition, a list of type equations is in solution form iff, for any equation $\text{Eq}(t, t')$ in `eqs`, `t` is a type variable that does not appear in `t'` and does not appear in any other equation in `eqs`.

```
# let isSol eqs = ...
val isSol : tpEquation list -> bool = <fun>
# isSol [] ;;
- : bool = true
# isSol [Eq(TpVar 1, Bool)] ;;
- : bool = true
# isSol [Eq(TpVar 1, Bool) ; Eq(TpVar 2, Nat)] ;;
- : bool = true
# isSol [Eq(TpVar 1, Bool) ; Eq(TpVar 1, Nat)] ;;
- : bool = false
# isSol [Eq(TpVar 0, List (Fnc(TpVar 1, String))); Eq(TpVar 8, Bool);
Eq(TpVar 1, TpNetVar 5)] ;;
- : bool = false
```

5. Write a function `eqSol2subst` that converts a list of equations in solution form into a corresponding substitution. More precisely, `eqSol2subst` takes a list of equations `eqs` and:

- If `eqs` is not in solution form, returns `None`
- If `eqs` is in solution form, returns `Some s`, where `s` is the function mapping each number `n` to:
 - `t`, if $\text{Eq}(\text{TpVar}(n), t)$ is in `eqs`,
 - `TpNetVar(n)`, otherwise.

```
let eqSol2subst eqs = ...
val eqSol2subst : tpEquation list -> (int -> tp) option = <fun>
# eqSol2subst [Eq(TpNetVar 1, Bool) ; Eq(TpNetVar 2, Nat) ;
Eq(TpNetVar 3, Bool) ; Eq(TpNetVar 1, Bool)] ;;
- : (int -> tp) option = None
# getProper(eqSol2subst []) 5 ;;
- : tp = TpNetVar 5
# getProper(eqSol2subst [Eq(TpNetVar 1, Bool)]) 1 ;;
- : tp = Bool
# getProper(eqSol2subst [Eq(TpNetVar 1, Bool)]) 0 ;;
- : tp = TpNetVar 0
```

6. This problem requires you to implement, for the object-language types, the unification algorithm described in the lectures.

Recall that a substitution is a function $s : \text{int} \rightarrow \text{tp}$. Any substitution yields a function $\text{extend}(s) : \text{tp} \rightarrow \text{tp}$ in a natural way: $\text{extend}(s)(t)$ is the type `t` with each of its type variables `TpNetVar(n)` substituted with `s(n)`.

Given two substitutions `s` and `s'`, `s` is said to be *more general* than `s'` if there exists a substitution `s''` such that $\text{extend}(s') = \text{extend}(s'') \circ \text{extend}(s)$. A substitution `s` is said to be a *unifier* for a type equation $\text{Eq}(t, t')$ if $\text{extend}(s)(t) = \text{extend}(s)(t')$. A *most general unifier* for a type equation $\text{Eq}(t, t')$ is a substitution `s` such that:

- `s` is a unifier for $\text{Eq}(t, t')$;

- s is more general than any unifier for $\text{Eq}(t, t')$.

Write a function `unif` that takes a type equation `eq` and returns:

- `None`, if `eq` does not have any unifier;
- `Some s`, if `s` is a most general unifier for `eq`.

(Note the indefinite article from “a most general unifier” due to the fact that most general unifiers are not unique, but only unique up to renaming of variables; for instance, if `eq` is `Eq(TpVar 0, TpVar 1)`, then there are at least two correct answers: the function mapping 0 to `TpVar 1` and any other number `n` to `TpVar n`, and the function mapping 1 to `TpVar 0` and any other number `n` to `TpVar n`. Any most general unifier that you provide will be accepted.)

You may want to define some helper functions inside the body of `unif`; also, you may want to use one or more functions previously defined in this MP.

```
# let unif eq = ...
val unif : tpEquation -> (int -> tp) option = <fun>
# unif (Eq(Nat, Bool)) ;;
- : (int -> tp) option = None
# getProper (unif (Eq(Fnc(TpVar 1, Prod(TpVar 3, TpVar 3)),
Fnc(Fnc(TpVar 2, String), TpVar 2)))) 3 ;;
- : tp = TpVar 3
# getProper(unif (Eq(Fnc(TpVar 0, Fnc (TpVar 0, TpVar 1)),
Fnc(Fnc (TpVar 1, TpVar 2), Fnc(Fnc(TpVar 1, TpVar 2),
Fnc(TpVar 3, TpVar 4)))))) 0 ;;
- : tp = Fnc (Fnc (TpVar 3, TpVar 4), TpVar 2)
```

7. Given two types t and t' , t is said to be *more general* than t' if there exists a substitution s such that $\text{extend}(s)(t) = t'$. Given a type equation $\text{Eq}(t, t')$ and a type t'' , t'' is said to be a *solution* for $\text{Eq}(t, t')$ if there exists a substitution s such that $s(t) = s(t') = t''$. A *most general solution* for an equation $\text{Eq}(t, t')$ is a solution for $\text{Eq}(t, t')$ which is more general than any other solution for $\text{Eq}(t, t')$.

You are asked to write a function `getSolution` that takes two types t and t' and returns:

- `None`, if the equation $\text{Eq}(t, t')$ does not have any solution;
- `Some t''`, if t'' is a most general solution for $\text{Eq}(t, t')$.

Note: Any version of most general solution that you provide will be considered to be a correct answer.

For solving this problem, it would help if you understood the relationship between (most general) solutions and (most general) unifiers.

```
# getSolution (TpVar 23) Bool ;;
- : tp option = Some Bool
# getSolution Bool Bool ;;
- : tp option = Some Bool
# getSolution (Prod(TpVar 1, TpVar 2)) (TpVar 2) ;;
- : tp option = None
```

```

# getSolution (Fnc(TpVar 0, Fnc (TpVar 0, TpVar 1)))
(Fnc(Fnc (TpVar 1, TpVar 2),Fnc(Fnc(TpVar 1,TpVar 2),Fnc(TpVar 3, TpVar 4)))) ;;
- : tp option =
Some
  (Fnc
    (Fnc (Fnc (TpVar 3, TpVar 4), TpVar 2),
      Fnc (Fnc (Fnc (TpVar 3, TpVar 4), TpVar 2), Fnc (TpVar 3, TpVar 4))))

```

Extra Credit Problem

- Write a function `isSol2` that has the same behavior as `isSol` from problem 4, but satisfies the following condition: does *not* use recursion explicitly (that is, does not use `let rec` in its body), but uses `List.fold_left` instead.

```

# let isSol2 eqs = ...
val isSol2 : tpEquation list -> bool = <fun>

```