
MP 4 – Unification Algorithm

CS 421 – Spring 2007

Revision

Assigned February 21, 2007

Due February 28, 2006, at 23:59pm

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

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. In the next MP, you will use the solution to this assignment in implementing a type inferencing algorithm.

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

3 Given Code

You are given some initial code in a file `mp4common.cmo`. You need to have open `Mp4common` on top of your `mp4.ml` file.

You may use any of the functions in the `List` module.

3.1 Types

You are given the following meta-types:

```
type expType = TyVar of int | TyConst of (string * expType list)
type substitution = int-> expType
```

In order to express the type of an expression we need a type definition to represent types. Again, these should all be familiar. The `TyVar` constructor takes an integer and represents the ‘a style polymorphic type variables. The constructor `TyConst` represents a type constructor, *name*, applied to a (possibly empty) list of arguments. For example, ‘a list would become `TyConst ("list", [TyVar 0])` and `int -> 'b` would become `TyConst ("->", [TyConst ("int", []); TyVar 1])`.

There is a companion function `showExpType` to display the types in an OCaml sort of way. Thus product types and function types are printed with an infix notation, while other type constructors use a postfix notation. Quoted alphabets are used for type variables, so that `TyVar 0` prints as ‘a.

Because the types `int` and `bool`, and the type constructors `->`, `*`, and `list` are used so frequently, we have included the following functions to ease their construction:

```

val int_ty : expType = TyConst ("int", [])
val bool_ty : expType = TyConst ("bool", [])
val mk_fun_ty : expType * expType -> expType = <fun>
val mk_list_ty : expType -> expType = <fun>
val mk_prod_ty : expType * expType -> expType = <fun>

```

Finally we provide a pair of functions `newTyVar : unit -> expType` and `resetCounter : unit -> unit`. The function `newTyVar` generates fresh type variables based on a counter, and `resetCounter` resets this counter.

3.2 Unification

The first thing you need to do to write a unification-based type inferencer is to write a unifier. A unifier takes a list of pairs of types that are supposed to be equal. It will return a substitution

Functions, integers, lists, etc. will be the terms in this system, and `TyVars` will represent variables.

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 unification problem C , consisting of a head (s, t) and tail C' , there are five cases to consider.

1. **Delete rule:** If s and t are equal, discard the pair, and unify C' .
2. **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'' . Output ϕ updated with $s \mapsto \phi(t)$
3. **Orient rule:** If t is a variable, and s is not, then discard (s, t) , add (t, s) to C' , and unify the result.
4. **Decompose rule:** If $s = \text{TyConst}(\text{name}, [s_1; \dots; s_n])$ and $t = \text{TyConst}(\text{name}, [t_1; \dots; t_n])$, then discard (s, t) , and add (s_i, t_i) to C' , for all $i = 1, \dots, n$, and unify the result.
5. If C is empty, then we return the identity substitution which maps each type variable the type expression that is that type variable.
6. If none of the above cases apply, it is a unification error (your `unify` function should return the `None` option in this case).

4 Problems

1. Write a function `contains : int -> Mp4common.expType -> 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. This function is used in case 2, and prevents recursive types.

```

| # contains 0 (TyConst ("->", [TyVar 0; TyVar 0]));
| - : bool = true
| # contains 0 (TyConst ("->", [TyVar 1; TyVar 2]));
| - : bool = false

```

2. Write a function `substitute : (int * Mp4common.expType) -> Mp4common.expType -> Mp4common.expType`. The first component in the input pair is the integer component of a `TyVar`, the second is the replacement value. The second input argument is the expression in which to perform the substitution.

```

# substitute (0, TyConst ("int", [])) (TyConst ("->", [TyVar 0; TyVar 0]));
- : Mp4common.expType = TyConst ("->", [TyConst ("int", []); TyConst ("int", [])])
# substitute (0, TyConst ("int", [])) (TyConst ("->", [TyVar 1; TyVar 2]));
- : Mp4common.expType = TyConst ("->", [TyVar 1; TyVar 2])

```

3. Write a function `lift_subst : (int -> Mp4common.expType) -> Mp4common.expType -> Mp4common.expType`. The first component in the input is a substitution and the second is a type expression, to which the substitution is to be applied.

```

# lift_subst (fun n -> if n = 0 then TyConst ("bool", []) else TyVar n)
  (TyConst ("->", [TyVar 1; TyVar 0]));
- : Mp4common.expType = TyConst ("->", [TyVar 1; TyConst ("bool", [])])

```

4. Now you are ready to write the unification function. Here's a sample run, based on the example given during the unification lecture.

```

# unify;;
- : (Mp4common.expType * Mp4common.expType) list -> substitution option = <fun>
# let f_opt =
  unify [(TyVar 0, TyConst ("list", [TyConst ("int", [])]));
        (TyConst ("->", [TyVar 0; TyVar 0]),
         TyConst ("->", [TyVar 0; TyVar 1]))];;
val f_opt : (int -> Mp4common.expType) option = Some <fun>
# match f_opt with Some f -> [f 0; f 1] | None -> [];;
- : Mp4common.expType list =
[TyConst ("list", [TyConst ("int", [])]);
 TyConst ("list", [TyConst ("int", [])])]

```