```
        let val t1 = tc e1
            val t2 = tc e2
            val new = Ty.freshTyvar ()
            val newt= Ty.mkTypeTyvar new
            val t2' = Ty.mkTypeList newt
            val S1 = Unify.unify(t2, t2')
                     handle Unify.Unify=>
                     raise TypeError(e2,"expected list type")

            val S2 = Unify.unify(S1 on newt,S1 on t1)
                     handle Unify.Unify=>
                     raise TypeError(exp,
                      "element and list have different types")
         in S2 on (S1 on t2)
        end
    | ...

   )handle Unify.NotImplemented msg => raise NotImplemented msg

end; (*TypeChecker*)
```

We also have to extend the Type functor to meet the enriched TYPE signature. The easiest way of doing this is

```
functor Type():TYPE =
struct
  type tyvar = int
  val freshTyvar =
      let val r= ref 0 in fn()=>(r:= !r +1; !r) end
  datatype Type = INT
              | BOOL
              | LIST of Type
              | TYVAR of tyvar
  ...

  fun mkTypeTyvar tv = TYVAR tv
  and unTypeTyvar(TYVAR tv) = tv
    | unTypeTyvar _ = raise Type

  fun mkTypeList(t)=LIST t
  and unTypeList(LIST t)= t
    | unTypeList(_)= raise Type
  type subst = Type -> Type

  fun Id x = x

  fun mkSubst(tv,ty)=
     let fun su(TYVAR tv')= if tv=tv' then ty else TYVAR tv'
         |   su(INT) = INT
         |   su(BOOL)= BOOL
         |   su(LIST ty') = LIST (su ty')
      in su
     end

  fun on(S,t)= S(t)

  fun prType ...
  |   prType (LIST ty) = "(" ^ prType ty ^ ")list"
  |   prType (TYVAR tv) = "a" ^ makestring tv
end;
```

Exercise 4 Extend Version 2 to handle equality. All you have to do is to fill in the relevant case in the definition of the function `tc`. (See appendix B about how you get the source of Version 2).

4.3 VERSION 3: A different implementation of types

Version 3 arises from Version 2 by replacing the `Type` functor by a different implementation of types. The idea is that istead of having substitutions as functions, we can implement type variables by references (pointers) and then do substitutions directly by assignments.

In case you have not seen the reserved word `withtype` before, `withtype` is used to declare a type abbreviaton locally within a `datatype` declaration.

```
functor ImpType():TYPE =
struct
  datatype 'a option = NONE | SOME of 'a

  datatype Type = INT
                | BOOL
                | LIST of Type
                | TYVAR of tyvar

  withtype tyvar =  Type option ref
  type tyvar = Type option ref

  fun freshTyvar() = ref (NONE)

  exception Type

  fun mkTypeInt() = INT
  and unTypeInt(INT)=()
    | ...
    | unTypeInt(TYVAR(ref (SOME t)))= unTypeInt t
    | unTypeInt _ = raise Type



  ...
  type subst = unit

  val Id = ();

  exception MkSubst;

  fun mkSubst(tv,ty)=
     case tv of
       ref(NONE) => tv:= (SOME ty)
     | ref(SOME t) => raise MkSubst

  fun on(S,t)= t

  fun prType ...
  |   prType (TYVAR (ref NONE)) = "a?"
  |   prType (TYVAR (ref (SOME t))) = prType t
end;
```

We can now build two systems at the same time and compare the efficiency of the two implementations. The nice thing is that we do not have to modify the typechecker functor at all, nor do we even have to modify the unification functor; we can just extend the final sequence of structure declarations to use both implementations of types.

Exercise 5 When I did this, I found (to my surprise), that the functional version in some cases was twice as fast, and never slower than the imperative variant. The relative performance of the two vary greatly from expression to expression. Can you find an expression for which the imperative version really is faster? (See Appendix B for how to get hold of the source of Version 3). Be careful with generating very demanding tasks for the ML system; you can make it crash!

ML implementors normally opt for the imperative version. In all fairness, the above comparison ignores that composing substitutions is much easier in the imperative version than it is in the applicative version; in the fragment of Mini ML considered so far, we have not had to compose substitutions.

One should not be too concerned with performance issues at too early a stage. It can be surprisingly difficult to predict where efficiency is most needed, and it is much more important, at first, to get the overall structure of the system right. It was important, for example, that we did NOT make the constructors of the datatype `Type` visible in the signature `TYPE`, and that we wrote the unification algorithm in a way which does not use the internal structure of `Type`. Had we not done this, we would not have been able to switch from one implementation to another that easily, and therefore chances are that we would chosen the imperative one, assuming that it was the more efficient one, without ever trying the "obvious" applicative implentation.

4.4 VERSION 4: Introducing variables and `let`

We now extend Version 3 by implementing the type checking of `let` expressions and of identifiers.

The typechecker function `tc` now has to take TWO arguments,

```
tc(TE, e)
```

where `e` is an expression and `TE` is a TYPE ENVIRONMENT, which maps variables occurring free in `e` to TYPE SHEMES. The definition of what a type scheme is will be given below; for now it suffices to know that every type can be regarded as a type scheme.

To take an example, if `TE` maps `x` to `int` and `y` to `int`, then `tc` will deduce the type `int` for the expression `x+y`. (However, if `TE` mapped `y` to `bool`, there would be a type error.)

The fact that we can bind variables to expressions whose types have been inferred to contain type variables means that we get type variables in the type environment. For instance, to type check

```
let x = [] in 4 :: x end
```

we first check `[]` yielding the type `'a1 list`, say. Then we bind `x` to the type scheme ∀`'a1.'a1 list`. Here the binding ∀`'a1` of `'a1` indicates that when we look up the the type of `x` in the type environment, we return a type obtained from the type scheme ∀`'a1.'a1 list` by instantiating the bound variables (here just `'a1`) by fresh type variables. In our example, when we look up `x` in the type environment during the checking of `4 :: x`, we instantiate `'a1` to a fresh type variable `'a2`, say, yielding the type `'a2 list` for `x`. Thus we get to unify `int list` against `'a2 list`, yielding the substitution of `int` for `'a2`.

Throughout the body of the `let`, `x` will be bound to ∀`'a1.'a1 list` in the type environment. Since we take a fresh instance of this type scheme each time we look up `x`, we can use `x` both as an `int list` and as an `int list list`, say:

```
    let x = [] in   (4::x)::x end
```

Exercise 6 Assuming that you instantiate the bound `'a1` to `'a3` when you meet the last occurrence of `x`, what two types should be unified, and what is the resulting substitution on `'a3` ?

The variable `x` is an example of POLYMORPHISM: after `x` has been declared, an occurrence of `x` can potentially be given infinitely many types: `int list`, `bool list`, `int list list`, and so on, all captured by the type scheme ∀ `'a1.'a1 list`. In ML, a TYPE SCHEME always takes the form $\forall\,\alpha_1 \cdots \alpha_n.\tau$, $(n \geq 0)$, where $\alpha_1, \ldots, \alpha_n$ are type variables and τ is a type. In the fragment of Mini ML considered so far, it will always be the case that any type variable occurring in τ is amongst the $\alpha_1, \ldots, \alpha_n$, but when one introduces functions and application, this no longer is the case.

Here is how we implement variables and `let`. We first extend the `TYPE` signature:

```
signature TYPE =
   sig
      ...
      type TypeScheme

      val instance: TypeScheme -> Type
      val close: Type -> TypeScheme

   end
```

Version 1 (Appendix A) already contains a signature for environments (find it). It was actually intended for the practical where you need it to extend the evaluator, but we can make use of it to implement type environments. The signature of the typechecker can be left unchanged, but we need to change the functor that builds the typechecker by including the environment management among the formal parameters:

```
functor TypeChecker
  (structure Ex: EXPRESSION
   structure Ty: TYPE
   structure Unify: UNIFY
      sharing Unify.Type = Ty
   structure TE: ENVIRONMENT
  )=
struct
  infix on
  val (op on) = Ty.on
  structure Exp = Ex
  structure Type = Ty
```