```
module PolyType (
        Type, Value, Term, Status,
        checktype,
        cnat, ctrue, cfalse,
        cmul, cadd,
        polyid, polycompose
) where


-- The basic types

data Type = NatType
          | BoolType
          | Fun Type Type
          | Poly Symbol Type    -- A polymorphic type
          | TypeVar Symbol      -- A bound type variable
          | Bound
            deriving (Read, Eq)

instance Show Type where
    show NatType = "Nat"
    show BoolType = "Bool"
    show (Fun a b) = "(" ++ show a ++ " -> " ++ show b ++ ")"
    show (Poly sym typ) = "(PI(" ++ [sym] ++ ")." ++ show typ ++ ")"
    show (TypeVar sym) = [sym]

-- We use just characters as variable symbols for simplicity
type Symbol = Char

-- a builtin function returns a _conditional_ value, because there may be
-- run-time type errors.. of course there cannot, since we have checked
-- types correctly.. :)

data Value = Bool Bool
           | Nat Integer
           | Func (Value -> Status Value)

instance Show Value where
    show (Func _) = "<func>"
    show (Bool x) = show x
    show (Nat n) = show n

-- The terms

data Term = Const Value Type
          | Var Symbol                  -- variable reference
          | App Term Term               -- function application
          | Lambda Symbol Type Term     -- lambda abstraction
          | TypeApp Term Type           -- type application on PI-types
          | TypeLambda Symbol Term      -- type abstraction

instance Show Term where
        show (Const val _) = show val
        show (Var sym) = [sym]
        show (App fun parm) = "(" ++ show fun ++ " " ++ show parm ++ ")"
        show (Lambda sym typ body) = "\\" ++ [sym] ++ ":" ++ show typ
                                     ++ "." ++ show body
        show (TypeApp term typ) = "(" ++ show term ++ " " ++ show typ ++ ")"
        show (TypeLambda sym body) = "\\" ++ [sym] ++ ":T." ++ show body
        show _ = "Foo"

-- A typing context is a list of symbol, type pairs. We make the
-- distinction between type variables and term variables

data CtxVal = CtxType Type      -- type variable
            | CtxVar Type       -- term variable
```

```
              deriving (Read,Eq)

type Ctx = [(Symbol, CtxVal)]




-- Looking up a symbol is a trivial association list search, except that
-- we recursively search type variables till we encounter one that is
-- bound.

lookupvar sym [] = fail $ "Term variable " ++ [sym] ++ " not found"
lookupvar sym ((s, t):xs) =
    if sym == s
       then case t of
                    CtxVar typ -> resolvetype typ xs
                    _ -> fail "Type var found where term expected"
       else lookupvar sym xs

lookuptype sym [] = fail $ "Type variable " ++ [sym] ++ " not found"

lookuptype sym ((s, t):xs) =
    if sym == s
       then case t of
                    CtxType Bound -> return $ TypeVar s
                    CtxType typ -> resolvetype typ xs
                    _ -> fail "Term var found where type expected"
       else lookuptype sym xs

resolvetype (TypeVar s) ctx = lookuptype s ctx
resolvetype (Fun parm ret) ctx =
    do resparm <- resolvetype parm ctx
       resret <- resolvetype ret ctx
       return $ Fun resparm resret

resolvetype (Poly sym typ) ctx =
    do restyp <- resolvetype typ ((sym, CtxType Bound):ctx)
       return $ Poly sym restyp

resolvetype typ _ = return typ

-- A status monad to pass error messages, since Maybe ignores them.

data Status a = Ok a
              | Error String
                deriving Show

instance Monad Status where
    Ok t >>= k = k t
    Error s >>= k = Error s
    return = Ok
    fail = Error


-- Typeof takes a term and a typing context and returns the type of the
-- term in that context if there is any, otherwise it fails.


-- The type of a constant is the type of the constant, regardless of the
-- context
typeof (Const _ typ) _ = return typ

-- The type of a variable needs just be looked up in the context
typeof (Var sym) ctx =
    do typ <- lookupvar sym ctx
       return typ
```

```
-- To get the type of an application term, we take the type of the
-- function and the type of the parameter, and if they are compatible, the
-- application returns the return type of the function
typeof (App fun parm) ctx =
    do Fun fparmtype rettype <- typeof fun ctx
       parmtype <- typeof parm ctx
       if fparmtype == parmtype
          then return rettype
          else fail $ "Function type incompatible with parameter"

-- We check that the restrictions of predicative polymorphism are obeyed.
typeof (Lambda _ (Poly _ _) _) _ =
    fail "Cannot lambda-abstract polymorphic types"

-- A lambda abstraction returns a function type, and adds a new variable to
-- the context where the body of the lambda is checked
typeof (Lambda sym typ term) ctx =
    do rettype <- typeof term $ (sym, CtxVar typ):ctx
       realtyp <- resolvetype typ ctx
       return $ Fun realtyp rettype

-- A lambda abstraction over types returns a PI-type. We simply add a new
-- type variable to the context where the body of the lambda is checked.
typeof (TypeLambda sym term) ctx =
    do rettype <- typeof term $ (sym, (CtxType Bound)):ctx
       return $ Poly sym rettype

-- Type application is fairly analogous to function application. The
-- term's type must be polymorphic, and we obtain the resulting type by
-- replacing the type variable with the parameter type.
typeof (TypeApp term typ) ctx =
    do (Poly sym rettyp) <- typeof term ctx
       realtyp <- resolvetype typ ctx
       resolvetype rettyp ((sym, CtxType realtyp):ctx)

-- checktype tests the type of a term in an empty context.

checktype term = disptype term (typeof term [])

disptype _ (Error s) = s
disptype term (Ok typ) = show term ++ " :: " ++ show typ


-- Now some stuff to actually construct terms to be tested.
-- Some sensible constants..

ctrue = Const (Bool True) BoolType
cfalse = Const (Bool False) BoolType
cnat n = Const (Nat n) NatType

-- wrapper to create binary integer operations

cnatbinop f = Const (natbinop f) $ Fun NatType $ Fun NatType NatType

natunop f = Func $ \x ->
            do Nat a <- return x
               return $ Nat $ f a

natbinop f = Func $ \x ->
             do Nat a <- return x
                return $ natunop $ f a

cmul = cnatbinop (*)
cadd = cnatbinop (+)
```

```
-- a builtin polymorphic function
-- cif : PI(t).Bool->t->t->t

cif = Const (Func $ \x ->
             do Bool b <- return x
                return $ Func $ \c ->
                 return $ Func $ \a ->
                  return $ if b then c else a)


      (Poly 't' $ Fun BoolType $ Fun (TypeVar 't') $ Fun (TypeVar 't') (TypeVar
't'))


-- Example terms, try "typecheck polycompose" and the like to see what the
-- typechecker thinks of the terms

-- polyid === \t:T.\x:t.x : PI(t).t->t

polyid = TypeLambda 't' $
         Lambda 'x' (TypeVar 't') $
         Var 'x'

-- compose === \r:T.\s:T.\t:T.\f:s->r.\g:t->s.\x:t.f(g x)
-- : PI(r).PI(s).PI(t).(s->r)->(t->s)->t->r

polycompose = TypeLambda 'r' $
              TypeLambda 's' $
              TypeLambda 't' $
              Lambda 'f' (Fun (TypeVar 's') (TypeVar 'r')) $
              Lambda 'g' (Fun (TypeVar 't') (TypeVar 's')) $
              Lambda 'x' (TypeVar 't') $
              App (Var 'f') $
              App (Var 'g') (Var 'x')

-- natcompose === polycompose Nat Nat Nat : (Nat->Nat)->(Nat->Nat)->Nat->Nat

natcompose = TypeApp (TypeApp (TypeApp polycompose NatType) NatType) NatType

-- square === \x:Nat.x*x : Nat->Nat

square = Lambda 'x' NatType $
         App (App cmul (Var 'x')) (Var 'x')

-- fourthpow === natcompose square square : Nat->Nat

fourthpow = App (App natcompose square) square

-- twice == \t:T.\x:t->t.polycompose t t t x x :: PI(t).(t->t)->t->t

twice = TypeLambda 't' $
        Lambda 'x' (Fun (TypeVar 't') (TypeVar 't')) $
        App (App (TypeApp
                  (TypeApp
                   (TypeApp polycompose (TypeVar 't'))
                   (TypeVar 't'))
                  (TypeVar 't'))
             (Var 'x'))
        (Var 'x')
```