Tüübituletus

- Avaldised

```
type Var  = String
data Term = Var Var | App Term Term | Lam Var Term | Let Var Term Term
          | Con Int | Pair Term Term | If Term Term Term
```

- Tüübid

```
type TVar = String
type TCon = String

data Type = TVar TVar | TCon TCon [Type]

tArrow t1 t2 = TCon "->" [t1,t2]
tPair  t1 t2 = TCon "*"  [t1,t2]
tBool        = TCon "Bool" []
tInt         = TCon "Int"  []
```

Tüübituletus

- Substitutsioonid

```
type TSubst = [(TVar,Type)]

nullTS :: TSubst
nullTS = []

(+->) :: TVar -> Type -> TSubst
x +-> t = [(x,t)]

(@@) :: TSubst -> TSubst -> TSubst
s1 @@ s2 = [(x, applyTS s1 t) | (x,t) <- s2] ++ s1
```

- Substitutsiooni rakendamine

```
class Types t where
   applyTS :: TSubst -> t -> t
   tVars   :: t -> [TVar]
```

Tüübituletus

- Substitutsiooni rakendamine

```
instance Types Type where
    applyTS s (TVar x)    = case lookup x s of
                              Just t  -> t
                              Nothing -> TVar x
    applyTS s (TCon c ts) = TCon c (map (applyTS s) ts)
    tVars (TVar x)    = [x]
    tVars (TCon c ts) = foldr union [] . map tVars $ ts

instance Types a => Types [a] where
    applyTS s = map (applyTS s)
    tVars     = nub . concat . map tVars
```

Tüübituletus

- Unifitseerimine

```
mgu :: Monad m => Type -> Type -> m TSubst
mgu (TVar x) t    = varBind x t
mgu t (TVar x)    = varBind x t
mgu (TCon c1 ts1) (TCon c2 ts2)
     | c1 == c2   = mguList nullTS ts1 ts2
     | otherwise = fail "Types do not unify"

mguList :: Monad m => TSubst -> [Type] -> [Type] -> m TSubst
mguList s0 (t1:ts1) (t2:ts2)
                  = do s1 <- mgu (applyTS s0 t1) (applyTS s0 t2)
                       mguList (s1 @@ s0) ts1 ts2
mguList s0 [] [] = return s0
```

Tüübituletus

- Unifitseerimine

```
varBind :: Monad m => TVar -> Type -> m TSubst
varBind x (TVar y) | x == y    = return nullTS
varBind x t | x `elem` tVars t = fail "Occurs check fails"
            | otherwise        = return (x +-> t)
```

Tüübituletus

- Tüübiskeemid

```
data TScheme = ForAll [TVar] Type

instance Types TScheme where
    applyTS s (ForAll xs t) = ForAll xs (applyTS s t)
    tVars (ForAll xs t)     = tVars t \\ xs

toScheme :: Type -> TScheme
toScheme t = ForAll [] t
```

- NB! applyTS definitsioon eeldab, et skemaatilised muutujad oleksid kõigist teistest muutujatest erinevate nimedega!?

Tüübituletus

- Eeldused

```
data Assumption = Var :>: TScheme

instance Types Assumption where
    applyTS s (x :>: sc) = x :>: applyTS s sc
    tVars (x :>: sc)     = tVars sc
```

- Keskonnad

```
type TEnv = [Assumption]

find :: Monad m => Var -> TEnv -> m TScheme
find x [] = fail ("Unbound identifier: " ++ x)
find x ((y:>:sc):env) | x == y    = return sc
                      | otherwise = find x env

extEnv :: TEnv -> (Var,TScheme) -> TEnv
extEnv env (x,sc) = (x:>:sc):env
```

Tüübituletus

- Tüübituletusmonaad

```
newtype TI a = TI (TSubst -> Int -> (a,TSubst,Int))

instance Monad TI where
   return x     = TI (\s n -> (x,s,n))
   (TI f) >>= g = TI (\s n -> case f s n of
                                (x,s',n') -> let TI gx = g x
                                             in gx s' n')

runTI :: TI a -> a
runTI (TI f) = x
   where (x,_,_) = f nullTS 0
```

Tüübituletus

- Tüübituletusmonaad

```
newTVar :: TI TVar
newTVar = TI (\s n -> ("a" ++ show n, s, n+1))

getSubst :: TI TSubst
getSubst = TI (\s n -> (s,s,n))

extSubst :: TSubst -> TI ()
extSubst u = TI (\s n -> ((), u @@ s, n))
```

- Unifitseerimine

```
unify :: Type -> Type -> TI ()
unify t1 t2 = do s <- getSubst
                 u <- mgu (applyTS s t1) (applyTS s t2)
                 extSubst u
```

Tüübituletus

- Instantsieerimine

```
freshInst :: TScheme -> TI Type
freshInst (ForAll xs t) = do s <- mapM (\x -> newTVar >>= \y ->
                                          return (x, TVar y)) xs
                             return (applyTS s t)
```

- Kvantifitseerimine

```
mkScheme :: Type -> TEnv -> TI TScheme
mkScheme t env = do s <- getSubst
                    let t' = applyTS s t
                    let scvs = tVars t' \\ tVars env
                    xs <- mapM (\x -> newTVar) scvs
                    let s' = zipWith (\v x -> (v, TVar x)) scvs xs
                    return (ForAll xs (applyTS s' t'))
```

Tüübituletus

- Tuletusreeglid

```
inferT :: Term -> TEnv -> TI Type
inferT (Con n) env        = return tInt
inferT (Pair e1 e2) env   = do t1 <- inferT e1 env
                               t2 <- inferT e2 env
                               return (tPair t1 t2)
inferT (If e0 e1 e2) env = do t0 <- inferT e0 env
                               unify t0 tBool
                               t1 <- inferT e1 env
                               t2 <- inferT e2 env
                               unify t1 t2
                               return t1
inferT (Var x) env        = do sc <- find x env
                               freshInst sc
```

Tüübituletus

- Tuletusreeglid

```
inferT (App e1 e2) env
          = do t1 <- inferT e1 env
               t2 <- inferT e2 env
               t  <- newTVar  >>= \x -> return (TVar x)
               unify (t2 `tArrow` t) t1
               return t
inferT (Lam x e1) env
          = do t0 <- newTVar >>= \x -> return (TVar x)
               t1 <- inferT e1 (extEnv env (x, toScheme t0))
               return (t0 `tArrow` t1)
inferT (Let x e1 e0) env
          = do t1 <- inferT e1 env
               sc <- mkScheme t1 env
               inferT e0 (extEnv env (x,sc))
```

Tüübituletus

- Algkeskond ja kogu avaldise tüübi leidmine

```
env0 = [ "true"  :>: toScheme tBool
       , "false" :>: toScheme tBool
       , "eq"    :>: toScheme (tArrow tInt (tArrow tInt tBool))
       , "add"   :>: toScheme (tArrow tInt (tArrow tInt tInt))
       ]

typeInf0 :: Term -> TI Type
typeInf0 e = do t <- inferT e env0
                s <- getSubst
                return (applyTS s t)
```