```
                           Control.Concurrent.STM
                           ----------------------
data STM a

data TVar a
newTVar  :: a -> STM (TVar a)                newTVarIO :: a -> IO (TVar a)
readTVar :: TVar a -> STM a                  readTVarIO :: TVar a -> IO a
writeTVar:: TVar a -> a -> STM ()

atomically:: STM a -> IO a
```

♣ Synchronizace

```
incS :: TVar Int -> QSemN -> IO ()
incS x s = do atomically $ do v <- readTVar x
                              writeTVar x (v + 1)
              signalQSemN s 1

doIncS :: Int -> TVar Int -> IO ()
doIncS n r = do s <- newQSemN 0
                replicateM_ n $ forkIO (incS r s)
                waitQSemN s n

w3 = do r <- atomically (newTVar 0)
        doIncS 100000 r
        x <- atomically (readTVar r)
        putStrLn (show x)
```

♣ Další operace s transakcemi

```
retry     :: STM a
orElse    :: STM a -> STM a -> STM a
throwSTM  :: Exception e => e -> STM a
catchSTM  :: Exception e => STM a -> (e -> STM a) -> STM a
```

♣ Finanční transakce

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically $ do withdraw from amount
                                          deposit to amount
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw from amount = do bal <- readTVar from
                          writeTVar from (bal - amount)

limitedWithdraw from amount = do bal <- readTVar from
                                 if amount > 0 && amount > bal
                                 then retry
                                 else writeTVar from (bal - amount)
```

♣ Čekání na událost

```
produce :: TVar [Int] -> Int -> IO ()
produce q n = atomically $ do s <- readTVar q
                              writeTVar q (n : s)

consume :: TVar Int -> TVar [Int] -> IO ()
consume e q = do s <- atomically $ do l <- readTVar q
                                      if length l < 100 then retry
                                      else do writeTVar q (drop 100 l)
                                              return (take 100 l)
                 putStrLn (show $ sum s)
                 atomically $ do k <- readTVar e
                                 writeTVar e (k-1)

w4 = do s <- atomically (newTVar [])
        e <- atomically (newTVar 10)
        replicateM_ 10 (forkIO $ consume e s)
        forM_ [1..1000] $ forkIO . produce s
        atomically $ do ne <- readTVar e            Existuje na to i funkce:
                        when (ne /= 0) retry        check (ne == 0)
```

♣ Další transakční struktury

Control.Concurrent.STM.TArray obsahuje **data** *TArray* i e

Control.Concurrent.STM.TChan obsahuje **data** *TChan* a s metodami

newTChan::*STM (TChan a)*, **readTChan**::*TChan a->STM a*, **writeTChan**::*TChan a->a->STM ()*

Template Haskell

♣ Funkce sčítající daný počet argumentů

```
{-# LANGUAGE TemplateHaskell #-}
import Language.Haskell.TH
sel i n = do
  a <- newName "a"
  lamE [if i’==i then varP a else wildP | i’<-[1..n]] (varE a)
```

Potom `$(sel 2 5)` je `\_ a_0 _ _ _ -> a_0 :: t->t1->t2->t3->t4->t1`

```
data Name
mkName :: String -> Name
newName :: String -> Q Name
nameBase :: Name -> String
nameModule :: Name -> Maybe String
```

`’funkce` vrátí *Name* od funkce ve scope

`’’typ` vrátí *Name* od typu ve scope

Pomocí `$(`něco typu *ExpQ*`)` se vloží výraz do kódu

```
type ExpQ = Q Exp; data Exp = VarE Name | ConE Name | LitE Lit | AppE Exp Exp |
  InfixE (Maybe Exp) Exp (Maybe Exp) | LamE [Pat] Exp | TupE [Exp] |
  CondE Exp Exp Exp | LetE [Dec] Exp | CaseE Exp [Match] | DoE [Stmt] |
  CompE [Stmt] | ArithSeqE Range | ListE [Exp] | SigE Exp Type |
  RecConE Name [FieldExp] | RecUpdE Exp [FieldExp]
```

Patterny jsou typu *PatQ*

```
type PatQ = Q Pat; data Pat = LitP Lit | VarP Name | TupP [Pat] |
  ConP Name [Pat] | InfixP Pat Name Pat | TildeP Pat | BangP Pat |
  AsP Name Pat | WildP | RecP Name [FieldPat] | ListP [Pat] |
  SigP Pat Type | ViewP Exp Pat
```

Typ `[| ... |]` je *ExpQ*

```
sum 1 = [| id |]
sum n = [| \x -> $(sum (n-1)) . (+ x) |]
```

V `ghci` je `:t $(sum 3)` typu `(Num a) => a -> a -> a -> a`. `:t sum` je `(Num t) => t -> ExpQ`.

V `ghci$ vrátí-} $(stringE . pprint =<< sum 3)`

```
\x_0->(\x_1->GHC.Base.id GHC.Base.. (GHC.Num.+ x_1)) GHC.Base.. (GHC.Num.+ x_0)
```

♣ Druhý pokus

```
sum n = do
  xs <- replicateM n (newName "x")
  lamE (map varP xs) $ foldr (\x sum -> [| $(varE x) + $sum |]) [|0|] xs
```

V `ghci$ vrátí-} $(stringE . pprint =<< sum 3)`

```
\x_0 x_1 x_2 -> x_0 GHC.Num.+ (x_1 GHC.Num.+ (x_3 GHC.Num.+ 0))
```

♣ Funkce `map` na i-tou položku n-tice

```
tmap i n = do
    as <- replicateM n (newName "a")
    [| \f -> $(lamE [tupP (map varP as)] $
                tupE [ if i==i’ then [| f $a |]
                                else a
                     | (a,i’) <- map varE as ‘zip‘ [1..] ])|]
```

♣ Typovaný `printf`

```
printf str = printf’ str [| [] |]
 where
  printf’ [] a = a
  printf’ (’%’:’s’:ss) a = [| \s -> $(printf’ ss [| $a ++       s  |]) |]
  printf’ (’%’:’d’:ss) a = [| \d -> $(printf’ ss [| $a ++ (show d) |]) |]
  printf’ (c:ss) a = printf’ ss  [| $a ++ [c] |]
```

Kód `printf "Ahoj %d %s"` se expanduje na

```
\d_0 -> \s_1 ->
  (((((((GHC.Types.[] GHC.Base.++ [’A’]) GHC.Base.++ [’h’]) GHC.Base.++ [’o’])
    GHC.Base.++ [’j’]) GHC.Base.++ [’ ’]) GHC.Base.++ GHC.Show.show d_0)
    GHC.Base.++ [’ ’]) GHC.Base.++ s_1

printf str = do (vars, code) <- printf’ str
                lamE vars code
 where printf’ [] = return ([], [| [] |])
       printf’ (’%’:’s’:ss) = do var <- newName "s"
                                 (vars, code) <- printf’ ss
                                 return (varP var:vars,[|$(varE var) ++ $code|])
       printf’ (’%’:’d’:ss) = do v <- newName "d"
                                 (vs, code) <- printf’ ss
                                 return(varP v:vs,[|show $(varE v) ++ $code|])
              -- return(sigP (varP v) [t|Int|]:vs,[|show $(varE v) ++ $code|])
       printf’ (c:ss) = do (vars, code) <- printf’ ss
                           return (vars, [| c : $code |])
```

Kód `printf "Ahoj %d %s"` se expanduje na

```
\d_0 s_1 -> ’a’ GHC.Types.: (’h’ GHC.Types.: (’o’ GHC.Types.: (’j’ GHC.Types.:
  (’ ’ GHC.Types.: (GHC.Show.show d_0 GHC.Base.++ (’ ’ GHC.Types.:
  (s_1 GHC.Base.++ GHC.Types.[])))))))
```

`[t| ... |]` vytváří *TypeQ*
`[d| ... |]` vytváří *DecQ*

♣ Zkoumání datových typů pomocí `reify`

```
reify :: Name -> Q Info
data Info = ClassI Dec [ClassInstance] | ClassOpI Name Type Name Fixity |
  TyConI Dec | PrimTyConI Name Int Bool | DataConI Name Type Name Fixity |
  VarI Name Type (Maybe Dec) Fixity | TyVarI Name Type
reify ’’Maybe = TyConI (DataD [] Data.Maybe.Maybe [PlainTV a]
                              [NormalC Data.Maybe.Nothing [],
                               NormalC Data.Maybe.Just [(NotStrict,VarT a)]] [])

reify ’foldr = VarI GHC.Base.foldr (ForallT [PlainTV a,PlainTV b] [] (AppT (AppT
 ArrowT (AppT (AppT ArrowT (VarT a)) (AppT (AppT ArrowT (VarT b)) (VarT b)))) (A
ppT (AppT ArrowT (VarT b)) (AppT (AppT ArrowT (AppT ListT (VarT a))) (VarT b))))
) Nothing (Fixity 9 InfixL)
```