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)
```

♣ Automatické generování instancí

```
class Json a where
  toJson :: a -> String
  fromJson :: ReadS a
instance Json Int where ...
instance Json String where ...

getRecFields (TyConI (DataD _ _ _ [RecC _ fields] _)) =
  [name | (name, _, _) <- fields]

toJsonT typ = do
  typdesc <- reify typ
  let fields = getRecFields typdesc
  [| \a -> intercalate "," $(listE [ [|$(stringE $ nameBase f) ++ ":" ++ toJson
                                        ($(varE f) a)|] | f <- fields]) |]

data A = A { a :: Int, b :: String, c :: String } deriving (Typeable, Data)
instance Json A where toJson = $(toJsonT ''A)
```

Jiný přístup ke generickému programování

```
data Company = C [Dept]
data Dept = D Name Manager [SubUnit]
data SubUnit = PU Employee | DU Dept
data Employee = E Person Salary
data Person = P Name Address

data Salary = S Float
type Manager = Employee
type Name = String
type Address = String

increase :: Float -> Company -> Company
increase k (C ds) = C (map (incD k) ds)

incD :: Float -> Dept -> Dept
incD k (D nm mgr us) =
  D nm (incE k mgr) (map (incU k) us)

incU :: Float -> SubUnit -> SubUnit
incU k (PU e) = PU (incE k e)
incU k (DU d) = DU (incD k d)

incE :: Float -> Employee -> Employee
incE k (E p s) = E p (incS k s)
```

```
incS :: Float -> Salary -> Salary          --"Tohle je jediný zajímavý kód
incS k (S s) = S (s * (1+k))

increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))

mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
mkT f = case cast f of Just g -> g
                       Nothing -> id

class Typeable a => Data a where
  gmapT :: (forall b. Data b => b -> b) -> a -> a
  gmapQ :: (forall b. Data b => b -> r) -> a -> [r]
instance Data Company where
  gmapT f (C depts) = C (f depts)
  gmapQ f (C depts) = [f depts]
instance Data Dept where
  gmapT f (D name man subs) = D name (f man) (f subs)
  gmapQ f (D name man subs) = [f man, f subs]
instance Data SubUnit where
  gmapT f (PU emp) = PU (f emp)
  gmapT f (DU dept) = DU (f dept)
  gmapQ f (PU emp) = [f emp]
  gmapQ f (DU dept) = [f dept]
instance Data Employee where
  gmapT f (E per sal) = E (f per) (f sal)
  gmapQ f (E per sal) = [f per, f sal]
instance Data Person where
  gmapT f p = p
  gmapQ f p = []
instance Data Salary where
  gmapT f s = s
  gmapQ f s = []

instance Data a => Data [a] where
  gmapT f [] = []
  gmapT f (x:xs) = f x : f xs
  gmapQ f [] = []
  gmapQ f (x:xs) = [f x, f xs]
```

Aplikuj transformaci všude, odspoda nahoru

```
everywhere :: Data a => (forall b. Data b => b -> b) -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

Aplikuj transformaci všude, odshora dolu

```
everywhereR :: Data a => (forall b. Data b => b -> b) -> a -> a
everywhereR f x = gmapT (everywhereR f) (f x)
```

♣ Jiný příklad

```
flatten :: Name -> Company -> Company
flatten d = everywhere (mkT (flatD d))

flatD :: Name->Dept->Dept
flatD d (D n m us) = D n m (concatMap unwrap us)
  where
    unwrap (DU (D d' m us)) | d == d' = PU m : us
    unwrap u = [u]
```

♣ Dotazy: kolik utratíme za platy?

```
totalSalary :: Company -> Float
totalSalary = everything (+) (0 `mkQ` getSalary)

getSalary :: Salary -> Float
getSalary (S s) = s

mkQ :: (Typeable a, Typeable b) => r -> (b->r) -> a -> r
mkQ r q a = case cast a of Just b -> q b
                           Nothing -> r
```

```
everything :: Data a => (r -> r -> r) -> (forall a. Data a => a -> r) -> a -> r
everything k f x = foldl k (f x) (gmapQ (everything k f) x)

type GenericT = forall a. Term a => a -> a
everywhere :: GenericT -> GenericT

type GenericQ r = forall a. Term a => a -> r
everything :: (r -> r -> r) -> GerericQ r -> GerericQ r

gmapM :: Monad m => (forall b. Term b => b -> m b) -> a -> m a

gfoldl :: (forall a b. Term a => w (a -> b) -> a -> w b)->
          (forall g. g -> w g) ->
          a -> w a

gmapT f (E p s) = E (f p) (f s)
gmapQ f (E p s) = f p : (f s : [])
gmapM f (E p s) = do { p' <- f p; s' <- f s; return (E p' s') }
gfoldl k z (E p s) = (z E `k` p) `k` s

gmapT f = gfoldl k id where k c x = c (f x)
newtype ID x = ID {unID :: x}
gmapT f x = unID (gfoldl k ID x) where k (ID c) x = ID (c (f x))

gmapM f = gfoldl k return where k c x = do c' <- c
                                           x' <- f x
                                           return (c' x')

gmapQ f = gfoldl k (const id) [] where k c x rs = c (f x : rs)
newtype Q r a = Q (unQ :: [r]->[r])
gmapQ f x = unQ (gfoldl k (const (Q id)) x) [] where k (Q c) x = Q (\rs -> c (f
x : rs))
```

♣ Inspekce typů

```
class Typeable a => Data a where
  ...
  dataTypeOf :: a -> DataType
  toConstr :: a -> Constr
  gunfold :: (forall b r. Data b => c (b -> r) -> c r) ->
             (forall r. r -> c r) ->
             Constr -> c a

dataTypeName :: DataType -> String
dataTypeConstrs :: DataType -> [Constr]
maxConstrIndex :: DataType -> ConIndex
indexConstr :: DataType -> ConIndex -> Constr
type ConIndex = Int -- Starts at 1

constrType :: Constr -> DataType
showConstr :: Constr -> String
constrIndex :: Constr -> ConIndex
constrFixity :: Constr -> Fixity
constrFields :: Constr -> [String]
data Fixity = ... -- Details omitted

gshow :: Data a => a -> String
gshow t = "(" ++ showConstr (toConstr t)
              ++ intercalate " " (gmapQ gshow t) ++ ")"

toJsonG :: Data a => a -> String
toJsonG a | Just i <- cast a :: Maybe Int = show i
toJsonG a | Just s <- cast a :: Maybe String = show s
toJsonG a | maxConstrIndex (dataTypeOf a) == 1 =
  intercalate "," [field ++ ":" ++ value | field <- constrFields $ toConstr a
                                         | value <- gmapQ toJsonG a ]
```