Continuation style passing

```
module Control.Monad.Cont

newtype Cont r a = Cont { runCont :: (a -> r) -> r }
```
CPS výpočet, který počítá dočasný výsledek typu `a` v CPS výpočtu, který nakonec spočítá výsledek typu `r`.

```
mapCont :: (r -> r) -> Cont r a -> Cont r a
mapCont f m = Cont $ f . runCont m

instance Functor (Cont r) where
    fmap f m = Cont $ \c -> runCont m (c . f)

instance Monad (Cont r) where
    return a = Cont ($ a)
    m >>= k  = Cont $ \c -> runCont m $ \a -> runCont (k a) c

square  x   =          x * x
square' x k = k      $ x * x
squareC x   = return $ x * x

runCont (squareC 5) print

class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a
```
`callCC` (call–with–current–continuation) zavolá funkci, které jako argument předá aktuální pokračování.
To umožňuje přerušit aktuální výpočet a okamžitě vrátit hodnotu.

```
instance MonadCont (Cont r) where
    callCC f = Cont $ \c -> runCont (f (\a -> Cont $ \_ -> c a)) c

squareC  x = return           $ x * x
squareC' x = callCC $ \k -> k $ x * x

-- Returns a string depending on the length of the name parameter.
-- If the provided string is empty, returns an error.
-- Otherwise, returns a welcome message.
whatsYourName :: String -> String
whatsYourName name =
  (`runCont` id) $ do
    response <- callCC $ \exit -> do
      validateName name exit
      return $ "Welcome, " ++ name ++ "!"
    return response

validateName name exit = do
  when (null name) (exit "You forgot to tell me your name!")
```

♣ Výjimky pomocí `callCC`

```
tryCont run handler =
  callCC $ \ok -> do
    err <- callCC $ \notOk -> do
      x <- run notOk
      ok x
    handler err

data SqrtException = LessThanZero deriving Show
sqrtExc n throw = do
  when (n < 0) $ throw LessThanZero
  return $ sqrt n

main = runCont (tryCont (sqrtExc (-3)) (error . show)) print
```

Víceparametrové typové třídy a typové rodiny

Řekněme, že chceme typovou třídu pro kolekci.

```
class Collection col where
  empty :: col
  member :: ??? -> col -> Bool
```

♣Kolekce může být kindu `* -> *`:

```
class Collection col where
  empty :: col elem
  member :: elem -> col elem -> Bool
```

Funguje dobře, dokud nebudeme chtít mít množinu čísel reprezentovanou v `Integer`u pomocí bitů, protože potom typ **`empty :: Integer Int`** nedává smysl. Obecně budeme mít problémy s kolekce, které fungují jenom pro nějaké typy.

♣Zkusíme víceparametrické třídy:

```
class Collection col elem where
  empty :: col                        Tohle způsobí kompilační chybu při prvním použití
  member :: elem -> col -> Bool
```

Jenomže když uvidíme **`empty :: col`**, jakou si máme vybrat? Máme mnoho voleb dle typy `elem`.
Řešením jsou funkční závislosti:

```
class Collection col elem | col -> elem where
  fromList :: [elem] -> col
  member :: Eq elem => elem -> col -> Bool
instance Eq a => Collection [a] a where
  fromList = id
  member = elem
instance Collection Integer Int where
  fromList [] = 0
  fromList (h : t) = (shift 1 h) + fromList t
  member elem col = (shift 1 elem .&. col) /= 0
```

♣Alternativa jsou typové třídy:

```
class Collection col where
  type ColElem col
  fromList :: [ColElem col] -> col
  member :: ColElem col -> col -> Bool
instance Eq a => Collection [a] where
  type ColElem [a] = a
  fromList = id
  member = elem
instance Collection Integer where
  type ColElem Integer = Int
  fromList [] = 0
  fromList (h : t) = (shift 1 h) + fromList t
  member elem col = (shift 1 elem .&. col) /= 0
```

Aritmetika pomocí funkčních závislostí a typových rodin

```
data Zero = Zero                zero = undefined :: One
data Succ a = Succ a
type One = Succ Zero            one = undefined :: One
type Two = Succ One             two = undefined :: Two
type Three = Succ Two           three = undefined :: Three
type Four = Succ Three          four = undefined :: Four
type Five = Succ Four           five = undefined :: Five

class IsNum n where
  showNum :: n->Int
instance IsNum Zero where
  showNum _ = 0
instance IsNum n => IsNum (Succ n) where
  showNum _ = 1 + showNum (undefined :: n)

class Plus a b c | a b -> c where
  plus :: a -> b -> c
  plus = undefined
instance Plus Zero a a where
instance Plus x a y => Plus (Succ x) a (Succ y) where

class Krat a b c | a b -> c where
  krat :: a -> b -> c
  krat=undefined
instance Krat Zero a Zero where
instance (Krat x a y, Plus y a z) => Krat (Succ x) a z where
```

Typ `plus two thre` je *Succ* (*Succ* (*Succ Two*))

`showNum $ krat four five` vrátí 20

♣Pomocí typových rodin to jde podobně:

```
class Plus a b where
  type PlusResult a b
  plus :: a -> b -> PlusResult a b
instance Plus Zero a where type PlusResult Zero a = a
instance Plus (Succ x) a where type PlusResult (Succ x) a = Succ (PT x a)

class Krat a b where
  type KratResult a b
  krat :: a -> b -> KratResult a b
instance Krat Zero a where type KratResult Zero a = Zero
instance Krat (Succ x) a where
  type KratResult (Succ x) a = PlusResult a (KratResult x a)
```