

Vstup a výstup, práce se souborovým systémem, výjimky, POSIX

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2017

Vstup a výstup: opakování

- běžné funkce referenčně transparentní
→ nemohou pracovat se vstupem/výstupem
- IO funkce (akce), typový konstruktor IO zabudovaný v jazyce
- `getLine :: IO String`
`putStrLn :: String -> IO ()`
- z IO není úniku, funkce volající IO akci nemůže vrátit typ bez konstruktoru IO
- hodnoty zabaleny, extrakce v `do` bloku nebo pomocí monadického bind operátoru
`(>>=) :: Monad m => m a -> (a -> m b) -> m b`

Modul System.IO: multiplatformní práce se soubory

- `type FilePath = String`
- základní funkce `readFile`, `writeFile`, `appendFile`
- datový typ `Handle` pro práci s proudy
 - `stdin`, `stdout`, `stderr` :: `Handle`
 - `openFile` :: `FilePath -> IOMode -> IO Handle`
 - `hClose` :: `Handle -> IO ()`
 - `hGetContents` :: `Handle -> IO String`
 - `hPutStr` :: `Handle -> String -> IO ()`
 - `withFile` ::
`FilePath -> IOMode -> (Handle -> IO r) -> IO r`
pro automatické uzavření handle po dokončení IO operace
 - a další, viz Hayoo/dokumentace

Modul `System.FilePath`: utility pro práci s cestami a jmény souborů

- platformě nezávislé
- automaticky používá správné separátory, rozpoznávání absolutních/relativních cest...
- importujeme `System.FilePath`, ten podle systému importuje buď `System.FilePath.Posix` nebo `System.FilePath.Windows`
- výrazně vhodnější než pracovat s cestami ručně

Práce s adresářovou strukturou

Modul `System.Directory`: multiplatformní manipulace s adresářovou strukturou, kopírování a odstraňování souborů, základní práce s právy.

```
import System.Directory
import System.FilePath
import Control.Monad ( forM_ )
main = do
  pwd <- getCurrentDirectory
  -- have to get rid of ., .., dotfiles
  entries <- filter ((/= '.') . head) <$>
              getDirectoryContents pwd
  forM_ entries $ \e -> do
    let abspath = pwd </> e
        isfile <- doesFileExist abspath
        let typ = if isfile then " is file"
                    else " is not file"
    putStrLn $ concat [ abspath, ": ", typ ]
```

Kombinování IO akcí, utility

Modul `Control.Monad`, funguje nejen pro IO ale pro všechny monády.

- `mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)`
`filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]`
obdoba běžných funkcí, volané funkce jsou IO akce
 - `forM`: jako `mapM`, převrácené argumenty
 - `mapM_`, `forM_` varianty ignorují výsledek
- `sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)`
spuštění seznamu IO akcí, obdobně `sequence_`
- `void :: Functor f => f a -> f ()`
- a další, viz dokumentace

- v podstatě umožňuje v IO akcích psát skoro jako v imperativním jazyce, včetně cyklů

Výjimky v Haskellu I

Výjimky lze chytat jen v IO, do funkcionálního paradigmatu se nehodí. Modul `Control.Exception`.

- výjimky je možné zachytávat v závislosti na typu
- typová třída `Exception`, vyžaduje `Show`
- při zachytávání třeba specifikovat typ výjimky
 - `SomeException` pro libovolnou
- `catch :: Exception e => IO a -> (e->IO a) -> IO a`
 - `expr `catch` \ex -> print (ex :: IOException) >> handleIOExc`
- často lépe používat `bracket`:

```
bracket :: IO a           -- ^ získání zdroje
        -> (a -> IO b)   -- ^ uvolnění zdroje
        -> (a -> IO c)   -- ^ operace se zdrojem
        -> IO c
```

```
withFile name mode =
    bracket (openFile name mode) hClose
```

Výjimky v Haskellu II

- vyhazování výjimek v čistém kódu možné, ale silně nevhodné
- `throwIO :: Exception e => e -> IO a`

```
import Control.Exception
import System.Environment; import System.IO
main = handle ioExpHandler $ do
    [from, to] <- getArgs
    withFile from ReadMode $ \hFrom ->
        withFile to WriteMode $ \hTo ->
            until (hIsEOF hFrom) $ do
                line <- hGetLine hFrom
                hPutStrLn hTo line

where
    ioExpHandler :: IOException -> IO ()
    ioExpHandler e = putStrLn $ "fatal: " ++ show e
    until :: IO Bool -> IO a -> IO ()
    until bool act = bool >>= \x -> case x of
        False -> act >> until bool act
        True  -> return ()
```


Moduly `System.Posix.*`: rozhraní k POSIX funkcím, funguje jen na Linux/Unix.

- plná podpora práv
- linky
- nízkoúrovňové rozhraní (file deskriptory, stat, . . .)
- signály

Vesměš lépe nepoužívat.

- 1 Zjistěte, jak v Haskellu lze o zadaném souboru rozhodnout, jestli je spustitelný.

- 1 Zjistěte, jak v Haskellu lze o zadaném souboru rozhodnout, jestli je spustitelný.
- 2 Implementujte funkci `isExecutable :: FilePath -> IO Bool`, která toto realizuje.

- 1 Zjistěte, jak v Haskellu lze o zadaném souboru rozhodnout, jestli je spustitelný.
- 2 Implementujte funkci `isExecutable :: FilePath -> IO Bool`, která toto realizuje.
- 3 Napište program, který za pomoci funkce `isExecutable` vypíše spustitelné soubory v aktuálním adresáři.

- 1 Zjistěte, jak v Haskellu lze o zadaném souboru rozhodnout, jestli je spustitelný.
- 2 Implementujte funkci `isExecutable :: FilePath -> IO Bool`, která toto realizuje.
- 3 Napište program, který který za pomoci funkce `isExecutable` vypíše spustitelné soubory v aktuálním adresáři.
- 4 Použili jste v tomto programu `filterM`? Jestli ne, napravte to.

Práce se souborovým systémem: úkol

Vaším úkolem je naprogramovat zjednodušenou verzi unixového příkazu `which`. Vytvořte tedy binárku, která pro každý argument z příkazové řádky zjistí, jestli se jedná o spustitelný soubor nacházející se někde v automaticky prohledávaných cestách (proměnná `PATH`). Jestli ano, vypíše celou cestu k této binárce. Několik poznámek:

- Kompilovaný program začíná běh ve funkci `main :: IO ()`.
- Použijte funkce z modulů `System.Directory`, `System.Environment` a `System.FilePath`.
- Doporučení: Raději než odchyťávat výjimky kontrolujte existenci souboru předem (i když to vytváří *race condition*).
- Pokud nebude zadán žádný argument, vypíše nápovědu.
- Vypisujte cestu pouze pro první nalezený spustitelný soubor daného jména.