

Vstup a výstup, zpracování chyb a výjimek

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2021

Vstup a výstup: opakování

- běžné funkce jsou referenčně transparentní
→ nemohou pracovat se vstupem/výstupem
- **IO** funkce (akce), typový konstruktore **IO**
- `getLine :: IO String`
`putStrLn :: String -> IO ()`
- z **IO** není úniku, funkce volající **IO** akci nemůže vrátit typ bez konstruktore **IO**
- hodnoty zabaleny, extrakce v **do** bloku nebo pomocí monadického operátore **bind**
`(>>=) :: Monad m => m a -> (a -> m b) -> m b`
- **IO** zajišťuje, že operace budou provedeny sekvenčně a budou mít odpovídající efekty na svět

Modul `System.IO`: multiplatformní práce se soubory

- `type FilePath = String`
- `data IOMode = ReadMode | WriteMode
 | AppendMode | ReadWriteMode`
- 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, vizte Hoogle/dokumentaci

Utility pro práci s cestami a jmény souborů

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

- platformně nezávislé
- `"some_dir" </> "some_file" <.> "ext"`
- `isAbsolute "/foo" ~>* True,`
`isRelative "bar" ~>* True`
- `takeExtension "a.out" ~>* ".out"`
- a spousta dalších
- importujeme `System.FilePath`, ten podle systému importuje buď POSIX nebo Windows variantu
- 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.

- `getCurrentDirectory`, `listDirectory`
- `doesFileExist "/etc/passwd" ~> * True`
- `doesDirectoryExist "/etc" ~> * True`
- `getTemporaryDirectory ~> * "/tmp"`
- `getPermissions`, `setPermissions` (jen základní práva, POSIX práva v POSIX-specifických modulech)
- `executable <$> getPermissions "/bin/ls" ~> * True`
- `getFileSize`
- `findFile`
- ...

Kombinování akcí IO, 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 akce IO

- `forM`: jako `mapM`, převrácené argumenty
- varianty `mapM_`, `forM_` ignorují výsledek

- `sequence :: (Traversable t, Monad m) =>`
`t (m a) -> m (t a)`

spuštění seznamu akcí IO, obdobně `sequence_`

- `void :: Functor f => f a -> f ()`
- a další, vizte dokumentaci
- v podstatě umožňuje v IO-akcích psát skoro jako v imperativním jazyce, včetně cyklů

Užitečné funkce pro interakci a argumenty příkazové řádky

Modul `System.Environment` poskytuje funkce pro práci s argumenty příkazové řádky a proměnnými prostředí.

- `getArgs :: IO [String]`
- `getProgName :: IO String`
- `getEnvironment :: IO [(String, String)]` a další

Při interakci (nejen) s uživatelem

- `when :: Applicative f => Bool -> f () -> f ()`
- `forever :: Applicative f => f a -> f b`
- `interact :: (String -> String) -> IO ()`

Pozor na buffering! Více v [dokumentaci](#).

Výjimky

- **Maybe, Either**
- využití toho, že obojí jsou funktory/monády
 - `fmap (+ 2) (lookup "a" [...])`
 - `lookup "a" [...] >>= foo`
- `maybe1 :: b -> (a -> b) -> Maybe a -> b`
`fromMaybe :: a -> Maybe a -> a`
(z `Data.Maybe`)
- `either1 :: (a -> c) -> (b -> c) -> Either a b -> c`

Věci, kde je selhání běžné: **Maybe**, **Either**

- `findExecutable :: String -> IO (Maybe FilePath)`
- zabalení v **IO** komplikuje práci

Věci, kde je selhání neočekávané/závažná chyba: **výjimky**

- podobně jako v imperativních jazycích (C++, C#, Java) používáme pro výjimečné případy
- pokud nechceme chybu ošetřovat hned, jak nastane
- o něco komplikovanější než v imperativních jazycích
- typicky neočekávaný stav souborového systému, nedostatečná práva,...

Výjimky v Haskellu I

- 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**
 - každý typ výjimky je instancí
- při zachytávání nutné, aby byl jednoznačně určen typ výjimky
 - aby se odvodilo, zda má být chycena
 - často vyžaduje explicitní otypování
 - **SomeException** pro libovolnou
- `catch :: Exception e => IO a -> (e->IO a) -> IO a`
 - `expr `catch` \ex -> print (ex :: IOException)`
`<-> IOException`
`>> handleIOExc`
- `try :: Exception e => IO a -> IO (Either e a)`

- správa zdrojů – `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
```

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

Interakce mezi výjimkami a leností je problém

```
>λ= pure (4 `div` 0) `catch`  
    \ex -> print (ex :: SomeException) >> pure 0
```

*** **Exception:** divide by zero

- výjimka nechycena!
- `pure` vrací nevyhodnocenou věc, až při vyhodnocení se vyhodí výjimka
- `catch` dříve než vyhodnocení
- pokud výjimku vyhazuje IO-funkce, problém typicky není

Výjimky v Haskellu IV

Při špatné interakci s leností je třeba vynutit vyhodnocení

- `evaluate :: a -> IO a` – vyhodnotí po vnější datový konstruktor
- vrací `IO`-akci, která vyhodí výjimku, pokud vyhodnocení vyhodilo výjimku

```
>λ= evaluate (4 `div` 0) `catch`  
  \ex -> print (ex :: SomeException) >> pure 0  
divide by zero
```

POZOR!

```
>λ= evaluate [4 `div` 0] `catch`  
  \ex -> print (ex :: SomeException) >> pure []  
[*** Exception: divide by zero
```

- úplné vyhodnocení pomocí `Control.DeepSeq.force`
(`evaluate (force x)`)

Výjimky v Haskellu: příklad

```
import Control.Exception
import System.Environment
import System.IO

main = handle ioExceptionHandler $ do
  [from, to] <- getArgs
  withFile from ReadMode $ \hFrom ->
    withFile to WriteMode $ \hTo ->
      until (hIsEOF hFrom) $ do
        line <- hGetLine hFrom
        hPutStrLn hTo line

where
  ioExceptionHandler :: IOException -> IO ()
  ioExceptionHandler 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  -> pure ()
```