Programy interaktywne

Typ reprezentujący operacje IO

1. funkcja zmieniająca „stan świata”

```
type IO = World -> World
```

2. funkcja zmieniająca „stan świata” i zwracająca wynik

```
type IO a = World -> (a, World)
```

Akcje

Akcja to wyrażenie typu `IO a`

```
IO Char   <-- typ akcji zwracającej znak
IO ()     <-- typ akcji zwracającej pustą krotkę
```

Typ jednostkowy

```
data () = ()
```

Podstawowe akcje

- akcja *getChar* wczytuje znak z klawiatury, wyświetla go na ekranie i zwraca jako rezultat

```
getChar :: IO Char
```

- akcja *putChar c* wyświetla znak *c* na ekranie i zwraca pustą krotkę

```
putChar :: Char -> IO ()
```

```
ghci> putChar 'a'
'a'
```

- akcja *return v* zwraca wartość *v* bez jakichkolwiek interakcji

```
return :: a -> IO a
return v = \world -> (v, world)
```

Operator sekwencji

```
(>>=) :: IO a -> (a -> IO b) -> IO b
f >>= g = \world -> case f world of
                      (v, world') -> g v world'
```

Uwaga

Jak w przypadku parserów zamiast operatora >>= można korzystać z notacji do

Przykład

```
a :: IO (Char, Char)
a = do x <- getChar
       getChar
       y <- getChar
       return (x, y)
```

Dalsze prymitywy

- *getLine*

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then return []
                else
                   do xs <- getLine
                      return (x:xs)
```

- *putStr*

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- *putStrLn*

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Przykład

```
strlen :: IO ()
strlen = do putStr "Enter a string: "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

```
ghci> strlen
Enter a string: Ala ma kota
The string has 11 characters
```

Przykład

```
ghci> hangman
Think of a word:
-------
Try to guess it:
> basic
-as----
> pascal
-as--ll
> haskell
You got it!
```

```
hangman :: IO ()
hangman = do putStrLn "Think of a word:"
             word <- sgetLine
             putStrLn "Try to guess it:"
             guess word
```

```
sgetLine :: IO String
sgetLine = do x <- getCh
              if x == '\n' then
                  do putChar x
                     return []
                else
                  do putChar '-'
                     xs <- sgetLine
                     return (x:xs)

getCh :: IO Char
getCh = do hSetEcho stdin False
           c <- getChar
           hSetEcho stdin True
           return c
```

```
guess :: String -> IO ()
guess word = do putStr "> "
                xs <- getLine
                if xs == word then
                   putStrLn "You got it!"
                  else
                    do putStrLn (diff word xs)
                       guess word

diff :: String -> String -> String
diff xs ys = [if elem x ys then x else '-' | x <- xs]

ghci> diff "haskell" "pascal"
"-as--ll"
```

Definicje lokalne

```
power = do putStr "Podaj liczbe: "
           n <- getLine
           let x = read n
               y = x^2
           putStrLn (n ++ " do kwadratu: " ++ show y)
```

```
ghci> power
Podaj liczbe: 12
12 do kwadratu: 144
```

Biblioteka IO (`import System.IO`)

```
data IOMode = ReadMode | WriteMode | AppendMode |
              ReadWriteMode

type FilePath = String

openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
hIsEOF   :: Handle -> IO Bool

hGetChar :: Handle -> IO Char
hGetLine :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar :: IO Char  <-- hGetChar stdin
getLine :: IO String
```

```
hPutChar  :: Handle -> Char -> IO ()
hPutStr   :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()

putChar  :: Char -> IO ()  <-- hPutChar stdout
putStr   :: String -> IO ()
putStrLn :: String -> IO ()

readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Przykład

```
doLoop = do
    putStrLn "Wpisz cNazwaPliku, zNazwaPliku lub w:"
    cmd <- getLine
    case cmd of
      'c':fname -> do handle <- openFile fname ReadMode
                      contents <- hGetContents handle
                      putStrLn "Pierwsze 100 znakow:"
                      putStrLn (take 100 contents)
                      hClose handle
                      doLoop
      'z':fname -> do putStrLn "Wpisz tekst:"
                      contents <- getLine
                      handle <- openFile fname WriteMode
                      hPutStrLn handle contents
                      hClose handle
                      doLoop
      'w':_     -> return ()
      _         -> doLoop
```

Rodzaje błędów IO (`import System.IO.Error`)

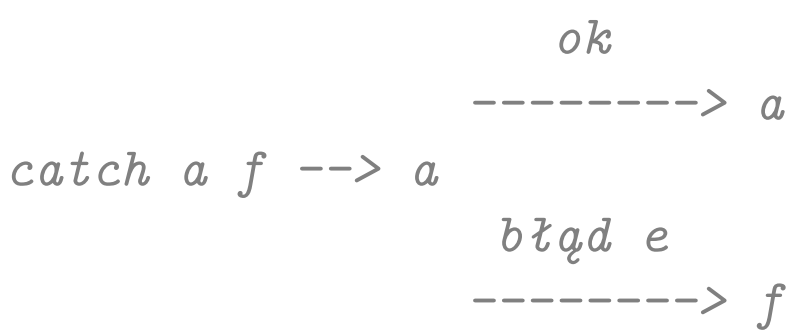
```
data IOError = ...   <-- typ błędu

isDoesNotExistError :: IOError -> Bool
isAlreadyInUseError :: IOError -> Bool
isPermissionError   :: IOError -> Bool
isEOFError          :: IOError -> Bool
```

Obsługa błędów

Akcja *catch* definiuje obsługę błędów, które może zgłosić akcja:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Idea działania:



Przykład

```
readLine :: Handle -> IO String
readLine h = catch (hGetLine h) (\e -> return "")
```

ioError

Akcja *ioError* przekazuje nieobsłużony błąd dalej

```
ioError :: IOError -> IO a
```

Przykład

```
readLine :: Handle -> IO String
readLine h = catch (hGetLine h) errorHandler
                where
                  errorHandler e = if isEOFError e
                                      then return ""
                                         else ioError e
```

Przykład

Argumentem *catch* może być sekwencja akcji

```
doRead :: String -> IO ()
doRead fname =
  catch (do handle <- openFile fname ReadMode
            contents <- hGetContents handle
            putStrLn "Pierwsze 100 znakow:"
            putStrLn (take 100 contents)
            hClose handle
        ) errorHandler
        where
          errorHandler e =
            if isDoesNotExistError e
              then putStrLn ("Nie istnieje " ++ fname)
                else return ()
```

Typy monadyczne

Klasa *Monad*

W przypadku parserów i programów interaktywnych definiowaliśmy:

1. ```
   return :: a -> Parser a
   (>>=)  :: Parser a -> (a -> Parser b) -> Parser b
   ```

2. ```
   return :: a -> IO a
   (>>=)  :: IO a -> (a -> IO b) -> IO b
   ```

Ogólnie:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

Uwaga

Notacja do może być użyta z dowolnym typem monadycznym

Moduły

Przykład

plik: Tree.hs

```
module Tree where
data Tree a = Empty | Node a (Tree a) (Tree a)

depth :: Tree a -> Int
depth Empty          = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

plik: Main.hs

```
module Main where
import Tree
main = putStrLn (show (depth (Node 1 Empty Empty)))
```

Uwaga

Każdy moduł zaczyna się domyślnie od `import Prelude`

Eksportowanie wybranych nazw

```
module Tree (Tree(Empty, Node), depth) where
lub
module Tree (Tree(...), depth) where
```

Importowanie nazw

```
import Tree
a = depth ...   lub   Tree.depth ...

import qualified Tree
a = Tree.depth ...   <-- jedyna możliwość

import qualified Tree as T
a = Tree.depth ... lub T.depth ...
```

Importowanie wybranych nazw

```
import Tree (depth)  <-- tylko depth
import Tree hiding (depth)  <-- wszystko oprócz depth
```

Leniwe wartościowanie

Wartościowanie/redukcja wyrażeń polega na aplikowaniu stosownych definicji do momentu, gdy kolejna aplikacja nie będzie możliwa

Przykład

```
inc n = n + 1
```

```
  inc (2 * 3)
=
  inc 6
=
  6 + 1
=
  7
```

lub

```
  inc (2 * 3)
=
  (2 * 3) + 1
=
  6 + 1
=
  7
```

Uwaga

W Haskellu dowolne dwa sposoby wartościowania jednego wyrażenia dają zawsze tę samą wartość (pod warunkiem, że oba się zakończą)

Strategie wartościowania

W pierwszej kolejności zawsze redukuj:

1. najbardziej wewnętrzne podwyrażenie, które można zredukować (*innermost reduction*)
2. najbardziej zewnętrzne podwyrażenie, które można zredukować (*outermost reduction*)

Problem stopu

```
inf = inf + 1
```

1. *innermost reduction*

```
  fst (0, inf)
=
  fst (0, 1 + inf)
=
  fst (0, 1 + (1 + inf))
=
  itd.
```

2. *outermost reduction*

```
  fst (0, inf)
=
  0
```

Uwagi

- Strategia *outermost reduction* może zwrócić wynik w sytuacji, gdy *innermost reduction* prowadzi do nieskończonych obliczeń
- Jeżeli dla danego wyrażenia istnieje kończąca się sekwencja redukcji, to *outermost reduction* również się kończy (z tym samym wynikiem)

Liczba redukcji

```
square n = n * n
```

1. *innermost reduction*

```
  square (1 + 2)
=
  square 3
=
  3 * 3
=
  9
```

2. *outermost reduction*

```
  square (1 + 2)
=
  (1 + 2) * (1 + 2)
=
  3 * (1 + 2)
=
  3 * 3
=
  9
```

Uwagi

- Strategia *outermost reduction* może wymagać większej liczby kroków niż *innermost reduction*
- Problem można rozwiązać współdzieląc wyniki wartościowania argumentów

```
  square (1 + 2)
=  ----------
  |    |       |
  _ * _    1 + 2
=  ----------
  |    |       |
  _ * _       3
=
  9
```

Uwagi

- Leniwe wartościowanie (*lazy evaluation*) = *outermost reduction* + współdzielenie wyników wartościowania argumentów
- Leniwe wartościowanie nigdy nie wymaga więcej kroków niż *innermost reduction*
- Haskell stosuje leniwe wartościowanie

Listy nieskończone

```
ones :: [Int]
ones = 1 : ones
```

```
ghci> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,... <Ctrl-C>
ghci> head ones
1
```

```
  head ones
=
  head (1 : ones)
=
  1
```

Uwaga

Korzystając z leniwego wartościowania, wyrażenia są wartościowane tylko w takim stopniu w jakim jest to potrzebne do obliczenia ich rezultatu

Przykład

```
take 0 _        = []
take _ []       = []
take n (x:xs) =
         x : take (n - 1) xs
```

```
ghci> take 3 ones
[1,1,1]
```

```
  take 3 ones
= ones
  take 3 (1 : ones)
= take
  1 : take 2 ones
= ones
  1 : take 2 (1 : ones)
= take
  1 : 1 : take 1 ones
= ones
  1 : 1 : take 1 (1 : ones)
= take
  1 : 1 : 1 : take 0 ones
= take
  1 : 1 : 1 : []
=
  [1,1,1]
```