Session 7

Io and Monads

Universiteit Antwerpen

Concept

- Monad = class of types which capture a concept.
  "A separate world"
- "Inside" a monad, types passed as
  template parameters are "wrapped"

Important: instantiating a monadic type with a
new template parameter creates a new type
"inside" the monad, but incompatible with another
instantiation (e.g. IO String >< IO Char)

More precisely, ...

- Monadic types support 2 essential operations

  return and **>>=** (“bind”)

  to respectively get "inside" the monad,
  and propagate further inside the monad

- Instance of Monad must be parameterized by 1 other type
  => e.g. instance Monad Maybe ...
  data Maybe a ...

Note for IO instance of Monad:
IO () == NULL: 1 type with 1 instance

Monad definition

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

- a type constructor m to represent the monad
- one operation to “enter” monad
- one operation to “propagate inside” monad

Universiteit Antwerpen

“do” Syntax

Syntactic sugar for chains of >>=

```
(e1 >>= \p -> e2) >>= \q -> e3
==
do p <- e1        -- mind indentation
   q <- e2
   e3
==
do { p <- e1;
   q <- e2; e3 }
```

Universiteit Antwerpen

The Maybe Monad

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just

data Maybe a = Nothing | Just a

myF :: Int -> Maybe Int
myF 3 = Nothing
myF x = Just (x+1)

myF 0 >>= myF >>= myF          -> Just 3
myF 0 >>= myF >>= myF >>= myF >>= myF ...      -> Nothing
```

Universiteit Antwerpen

Example

```
data Cow = Ncow String | Ccow String deriving Show

father :: Cow -> Maybe Cow
father (Ncow x) = return (Ncow ("Father of " ++ x))
father (Ccow x) = Nothing

mother :: Cow -> Maybe Cow
mother (Ncow x) = return (Ncow ("Mother of " ++ x))
mother (Ccow x) = Nothing
```

Example (cont.)

```
maternalGrandfather :: Cow -> Maybe Cow
maternalGrandfather s = case (mother s) of
    Nothing -> Nothing
    Just m  -> father m

mothersPaternalGrandfather :: ??
```

Find a more elegant way using >>= and return
Find a more elegant way using “do”

Example functions written on the IO instance

```
writeln::String->IO ()
writeln cs = do write cs
                putChar '\n'
```

```
writeln cs = do {write cs; putChar '\n'}
```

```
readn::Int->IO String
readn 0 = return []
readn (n+1) = do {c <- getChar;
                  readn n >>= (\cs -> return (c:cs))}
```

```
readn (n+1) = do {c <- getChar;
                  cs <- readn n;
                  return (c:cs)}
```

Universiteit Antwerpen

Exercises

- Palindrome
- GuessWord
  - ? guessword // User starts the 'game'
  - Think of a word: // Prompt to user
  - ---- // do not echo the word that is entered
  - Now try to guess it! // Prompt to user
  - guess: last // Prompt user who enters a word
  - --al // Indication of right letters
  - guess: dial // Prompt user who enters a word
  - --al // Indication of who letters
  - guess: opal // Prompt user who enters a word
  - -oal // Indication of right letters
  - guess: foal // Prompt user who enters a word
  - -oal // Indication of right letters
  - guess: goal // Prompt user who enters a word
  - You got it! // Indication of SUCCESS, GAME OVER

```
getChar= do {
            c<-getCh;
            putChar c;
            return c
        }
```

OR (new version of hugs)

```
hSetEcho stdin False
```

in System.IO

```
compare::String->String->String
compare word cs= map check word
    where check w= if member cd w
                   then w
                   else '-'
```

Universiteit Antwerpen