1.4 Monads

Mittwoch, 4. Mai 2016 10:00

Monads = concept from category theory, can be used in functional programs as a prog. technique and to handle input/output.

Functional programs should not have side effects

⇒ Referential transparency:

Applying a function to the same arguments should always yield the same result (i.e., it should not depend on the memory, the user, the contents of files, ...)

⇒ We can't write function whose result is the input that was typed in by the user.

In-/Output are desired side effects.

How can we perform in-/output without violating referential transparency?

Solution:
- There is a data type `IO ()` whose values are actions.
- Evaluation of an expression of type `IO ()` performs the corresponding action.
- `IO ()` is an abstract data type with

certain pre-defined functions. The user can implement new function building upon these pre-defined functions.

- Pre-defined function

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

The value of the expression `putChar '!'` is the action that prints a ! on the screen.

- To combine actions, there is also a pre-defined function `>>` ("then"):

```
(>>) :: IO () -> IO () -> IO ()
```

Ex: `putChar 'a' >> putChar 'b'`

action of type IO() which prints ab on the screen

- pre-defined function:

```
return () :: IO ()
```

empty action

```
putChar '!' >> return ()
```

putChar ! -- return ()

action which prints ! on the screen

One can implement (recursive) algorithms on this data type:

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

Up to now, we only considered output actions.
Now we also regard input.

⇒ Consider a type `IO a`

This is the type of IO-actions which compute a value of type a. (IO is a pre-defined type constructor.)

pre-defined function `getChar :: IO Char`

Its value is the action which reads a character from the keyboard. Its value is not the character that the user types in.

We depict actions of type `IO a` as:

← The value of type a is encapsulated inside the IO action.

IO () is a special case: () is a type with just a single value (). So output actions are treated as if one would determine the fixed value ().

In general, return can be used to create empty actions of any type IO a:

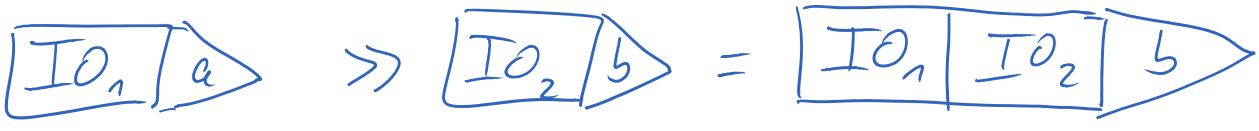
```
return :: a -> IO a
```

So return '!' is the empty action of type IO Char where the value '!' of type Char is encapsulated.

Similarly, >> can be used to combine two arbitrary IO-actions:

```
(>>) :: IO a -> IO b -> IO b
```

Graphically: (Slide 27)



↑ The value of type a that is determined in the first action is ignored in the resulting combined action.

```
getChar >> return ()
```

reads a character from the keyboard

IO Char IO () and ignores it

The operator $\gg$ is only useful for actions $p \gg q$ where the action q does not depend on the value that was determined during the action p.
⇒ We need another operator $\gg=$ ("bind") where the second action can access the value that is encapsulated in the first action.

$$(\gg=) :: IO\ a \to (a \to IO\ b) \to IO\ b$$

To evaluate $p \gg= f$, one first evaluates the action p (of type $IO\ a$). During this evaluation, a value x of type a is determined. Now f can access this encapsulated value x and the action $f\ x$ of type $IO\ b$ is performed.

type $a \to IO\ b$ ↗ ↖ type a

Ex: One could implement $\gg$ using $\gg=$:

$$p \gg q = p \gg= (\backslash \_ \to q)$$

function that ignores its argument and returns the action q

```
echo :: IO ()
echo = getChar >>= putChar
```

← echo reads a character from the keyboard and prints it on the screen

getChar: type IO Char

putChar: type Char → IO ()

Reconsider referential transparency: What is the difference between

`echo >> echo` and `let x = echo in x >> x`

There is no difference! The value of echo is always the same (it's the action "read a character and print it").
In both cases, the echo-action is performed twice (the user enters 2 possibly different characters which are printed on the screen.)
Value of echo is always the same, it does not depend on the user. ⇒ Referential transparency is not destroyed.
To this end, it is important that Haskell does not allow the programmer to access the value that is encapsulated in an IO action and to return it as the result of a function:

```
result :: IO a -> a
```

Such a function "result" does not exist.

Values that depend on inputs are always encapsulated

in IO actions ⇒ clear separation between program parts with and without side-effects.

To perform IO actions with files, there exist similar pre-defined functions:

```
readFile :: String -> IO String
```

readFile "myFile" — action which reads myFile and encapsulates the content of the file

```
writeFile :: String -> String -> IO ()
```

writeFile "myFile" "hello" — action which overwrites the current content of myFile by the string "hello".

Ex: gets n should be a function which reads a string of length n from the keyboard (Slide 28)

```
gets :: Int -> IO String
gets n = if n <= 0 then return []
         else getChar >>= \x ->
              gets (n-1) >>= \xs ->
              return (x:xs)
```

return [] — empty action which encapsulates the empty string

x — first character that is read from the keyboard

xs — string of length n-1 read by gets (n-1)

Very often, one has to combine actions in this way:

```
p >>= \x -> q
```

p: type IO a; x: type a; q: type IO b

This can instead be written with a more readable syntax:

```
do x <- p
   q
```

type IO a

q

type IO b

| do x ← p
  q

This means: first perform action p, store the value determined during this action in x, then perform action q.

p >>= \x ->
  q >>= \y ->
    r

is equivalent to

do x ← p
   y ← q
   r

In general, the do-construct is defined by the following translation:

```
do {x<-p; S}   = p >>= \x -> do {S}
do { p ; S }   = p >>  do {S}
do {p}         = p
```

With this notation, the algorithm for gets becomes much more readable:

```
gets :: Int -> IO String
gets n = if n <= 0 then return []
         else do x  <- getChar
                 xs <- gets (n-1)
                 return (x:xs)
```

→ looks like an imperative program

But: it is still a functional program with

referential transparency. There is no possibility to extract x or xs outside an IO action.

Type constructor

IO is a special case of a monad.

Haskell has several more monads (not for IO, but for prog. style). Every monad has the functions return, >>, >>= and supports the "do"-notation.

One can also implement user-defined monads.

The idea is always to separate computations of encapsulated values from other computations.

Only the IO monad forbids the extraction of encapsulated values to ensure referential transparency.