Haskell Live

[09] IO in Haskell und Aufgabenblatt 6

Bong Min Kim
e0327177@student.tuwien.ac.at

Christoph Spörk
christoph.spoerk@inode.at

Florian Hassanen
florian.hassanen@googlemail.com

Bernhard Urban
lewurm@gmail.com

3. Dezember 2010

```
import Data.List
import Data.Char
import System
```

IO in Haskell – Idee

```
  -- ACHTUNG: Folgendes ist (teilweise) kein gültiger Haskell Code
getchar :: Char   -- wir gehen von der offensichtlichen Funktionalität aus
get2chars :: [Char]
get2chars = [getchar, getchar]
  -- Referenzielle Transparenz ⇒ getChar muss immer
  -- den selben Wert liefern!
getchar2 :: Int → Char
get2chars2 :: [Char]
get2chars2 = [getchar2 1, getchar2 2]
  -- Wir erzwingen nun, dass getchar2 zweimal ausgewertet werden muss.
  -- Die gleiche Taktik wollen wir für get2chars2 ebenfalls verfolgen:
getchar3 :: Int → Char
get2chars3 :: Int → [Char]
```

```
get2chars3 _ = [getchar3 1, getchar3 2]
  -- Eine weitere Frage: In welcher Reihenfolge werden die Ausdrücke
  -- ausgewertet? Bei I/O von wesentlicher Bedeutung.
  -- Wie können wir eine Reihenfolge erzwingen? Datenabhängigkeit!
getchar4 :: Int → (Char, Int)
get2chars4 :: Int → [Char]
get2chars4 _ = [a, b]
  where
  (a, i) = getchar4 1
  (b, _) = getchar4 i
  -- Schaut schon ziemlich gut aus! Probleme könnten noch bei mehrere Aufrufen
  -- von get2chars4 entstehen, da der Compiler/Interpreter ja sieht, dass
  -- wir in Wirklichkeit den Int gar nicht verwenden, deshalb verwenden
  -- wir diesen einfach:
getchar5 :: Int → (Char, Int)
get2chars5 :: Int → ([Char], Int)
get2chars5 i0 = ([a, b], i2)
  where
  (a, i1) = getchar5 i0
  (b, i2) = getchar5 i1
  -- Das Muster entspricht schon fast der tatsächlichen Definition von
  -- IO a in GHC (hugs verwendet hierfür ein Built-In):
data RealWorld
type IO a = RealWorld → (a, RealWorld)
  -- auch nicht ganz wahr... ;-) Interessierte können sich im GHC Source erkundigen
```

IO Monad – Idee

```
main1 :: IO ()
main1 world0 = (b, world2)
  where
  (a, world1) = putStrLn "hallo" world0
  (b, world2) = putStrLn "welt" world1
  -- Nicht besonders komfortabel, deshalb machen wir eine Funktion daraus:
(≫) :: IO a → IO b → IO b
(action1 ≫ action2) world0 = (b, world2)
  where
  (a, world1) = action1 world0
  (b, world2) = action2 world1
  -- Erlaubt uns main1 einfacher zu notieren:
main2 :: IO ()
```

```
main2 = putStrLn "hallo" >> putStrLn "welt"
  -- Lässt sich auch verschachteln:
main3 :: IO ()
main3 = putStrLn "hallo" >> (putStrLn "welt" >> putStrLn "juhu")
  -- Ein anderes Beispiel: Wir wollen das Ergebnis einer vorigen Operation in der
  -- nächsten verwenden:
main4 :: IO ()
main4 world0 = (b, world2)
  where
  (a, world1) = getLine world0
  (b, world2) = putStrLn a world1
  -- lässt sich auch wieder in eine Funktion kapseln:
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 = (b, world2)
  where
  (a, world1) = action1 world0
  (b, world2) = action2 a world1
  -- Erlaubt uns main4 einfacher zu notieren:
main5 :: IO ()
main5 world0 = getLine >>= putStrLn
```

Tatsächlich haben wir hier schon die Funktionen für den sogenannten IO Monaden implementiert und sind so auch (ähnlich) im Prelude definiert.

IO Monaden

```
  -- ab hier ausführbare Beispiele!
io1 :: IO ()
io1 = putStr "hallo" >> putStrLn " welt :) (again)"
io2 :: IO ()
io2 = putStr "Name? "
   >> getLine
   >>= \a -> putStr "Alter? "
   >> getLine
   >>= \b -> putStr (a ++ " " ++ b ++ "\n")
  -- Warum geht das? >> und >>= sind linksassoziativ,
  -- Lambda aber rechtsassoziativ. Das ganze wird recht schnell unübersichtlich,
  -- daher verwendet man die sogenannte do-Notation:
io3 :: IO ()
io3 = do    -- gleiche Funktionalität wie in io2
  putStr "Name? "
  a <- getLine
```

```
  putStr "Alter? "
  b ← getLine
  putStr (a ++ "  " ++ b ++ "\n")
  -- Man kann auch eigene Funktionen im IO-Kontext definieren und verwenden:
myGetLine :: String → IO String
myGetLine prefix = do
  x ← getLine
  return (prefix ++ x)
io4 :: IO ()
io4 = do
  putStr "-> "
  a ← myGetLine "foo"
  putStr a
  -- Manchmal will man eine Liste von IO-Objekten verarbeiten:
main :: IO ()
main = do
  args ← getArgs
  let arg1 = if length args > 0 then args !! 0 else "/proc/cpuinfo"
  content ← readFile arg1
    -- Funktionen (hier: lines) ohne IO auch verwendbar
    -- müessen aber in einem let-Ausdruck stehen
  let alllines = lines content
  sequence_ [putStrLn i | i ← (take 5 alllines)]
```

Tipp: Mit `ghc -o 09hl 09hl.lhs` kann man aus diesem Haskellskript ein ausführbares Programm generieren lassen. Ausgeführt werden beim Aufruf vom Programm `09hl` die Anweisungen in `main`.

Aufgabenblatt 6

```
type Vertex = Integer
type Origin = Vertex
type Destination = Vertex
type Key    = Integer
type Name   = Integer


data BTree a = BLeaf Key a |
  BNode Key a (BTree a) (BTree a) deriving Show
data LTree a = LNode Key a [(LTree a)] deriving Show
data ALgraph = ALg [(Origin, [Destination])] deriving (Eq, Show)
```

```
class Structure s where
  noOfSources :: s → Integer
  noOfSinks :: s → Integer
  notSourceConnected :: s → [Name]
  notSinkConnected :: s → [Name]
```

instance eq btree

```
instance Eq a ⇒ Eq (BTree a) where
  (BLeaf _ a) ≡ (BLeaf _ a') = (a ≡ a')
  (BNode _ a t1 t2) ≡ (BNode _ a' t1' t2') = (a ≡ a' ∧ t1 ≡ t1' ∧ t2 ≡ t2')
  _                 ≡ _                    = False
```

instance eq ltree

```
instance Eq a ⇒ Eq (LTree a) where
  (LNode _ a trees) ≡ (LNode _ a' trees') = a ≡ a' ∧ (trees ≡ trees')
```

instance of structure

btree

```
b2l (BLeaf k v) = (LNode k v [])
b2l (BNode k v t1 t2) = (LNode k v [b2l t1, b2l t2])
instance Structure (BTree a) where
  noOfSources t       = noOfSources       (b2l t)
  noOfSinks t         = noOfSinks         (b2l t)
  notSourceConnected t = notSourceConnected (b2l t)
  notSinkConnected t  = notSinkConnected  (b2l t)
```

ltree

```
l2al tree = ALg (l2al' tree)
l2al' (LNode k v list) = [(k, children)] ++ (concat [l2al' tree | tree ← list])
                         where children = [k | (LNode k _ _) ← list]
instance Structure (LTree a) where
  noOfSources t       = noOfSources       (l2al t)
  noOfSinks t         = noOfSinks         (l2al t)
  notSourceConnected t = notSourceConnected (l2al t)
  notSinkConnected t  = notSinkConnected  (l2al t)
```

algraph

```
instance Structure (ALgraph) where
  noOfSources g          = fromIntegral $ length (getSources g)
  noOfSinks g            = fromIntegral $ length (getSinks g)
  notSourceConnected g = (getNodes g) \\ (sourceConnected g)
  notSinkConnected g = (getNodes g) \\ (sinkConnected g)


getNodes (ALg al) = (sort ∘ nub ∘ concat) [from : tos | (from, tos) ← al]


getSources g@(ALg al) =
  [n | n ← nodes, ¬ (n ∈ all_tos)]
  where nodes = getNodes g
    all_tos = concat [tos | (_, tos) ← al]


getSinks g@(ALg al) =
  [n | n ← nodes, ¬ (n ∈ all_froms)]
  where nodes = getNodes g
    all_froms = [from | (from, tos) ← al, tos ≢ []]


type VertexMapping = (Vertex → [Vertex])
forwardMapping :: ALgraph → VertexMapping
forwardMapping (ALg []) _ = []
forwardMapping (ALg ((from, tos) : rs)) x
| x ≡ from = tos
| otherwise = forwardMapping (ALg rs) x


reverseMapping :: ALgraph → VertexMapping
reverseMapping (ALg []) _ = []
reverseMapping (ALg ((from, tos) : rs)) x
| x ∈ tos = from : (reverseMapping (ALg rs) x)
| otherwise = reverseMapping (ALg rs) x


mappingMinus :: VertexMapping → Vertex → VertexMapping
mappingMinus mapping remove = (filter (remove ≢)) ∘ mapping


(\\\) :: VertexMapping → Vertex → VertexMapping
(\\\) = mappingMinus
```

```
(\\\\) :: VertexMapping → [Vertex] → VertexMapping
(\\\\) = foldl' (\\\)

sourceConnected g@(ALg al) = reachable (mapping \\\\ sources) sources
  where sources = getSources g
        mapping = forwardMapping g

sinkConnected g@(ALg al) = reachable (mapping \\\\ sinks) sinks
  where sinks = getSinks g
        mapping = reverseMapping g

reachable :: VertexMapping → [Vertex] → [Vertex]
reachable mapping set
  | neighbours ≢ [] = reachable (mapping \\\\ neighbours) (set ++ neighbours)
  | otherwise = set
  where neighbours = nub [n | x ← set, n ← mapping x]
```

accept

```
type State       = Integer
type StartState = State
type AcceptingStates = [State]
type Word a     = [a]
type Row a      = [[a]]

data AMgraph a = AMg [(Row a)] deriving (Eq, Show)
type Automaton a = AMgraph a

type AutomataMapping a = State → a → [State]
amapping :: Eq a ⇒ (AMgraph a) → (AutomataMapping a)
amapping (AMg matrix) from weight = neighbours
  where row = matrix !! (fromIntegral from)
        size       = (length row) − 1
        neighbours = [fromIntegral n | n ← [0 .. size], weight ∈ (row !! n)]

accept :: Eq a ⇒ (Automaton a) → StartState → AcceptingStates → (Word a) → Bool
accept auto start ends word =
  accept' (amapping auto) start ends word

accept' :: Eq a ⇒ (AutomataMapping a) → StartState → AcceptingStates → (Word a) → Bool
accept' _ current ends [] = current ∈ ends
accept' mapping current ends (char : chars) =
  any (λnext → accept' mapping next ends chars) (mapping current char)
```

alternatives accept

```
accept2 :: Eq a => (Automaton a) -> StartState -> AcceptingStates -> (Word a) -> Bool
accept2 ma src sinks [] = src ∈ sinks
accept2 ma@(AMg matrix) src sinks (kante : xs) =
  or [accept2 ma (fromIntegral y) sinks xs
     | y ← [0 .. ((length row) − 1)]
     , kante ∈ (row !! y)
     ]
  where row = matrix !! (fromIntegral src)
```