Monadit ja niiden sovelluksia

Riku Saikkonen

23. maaliskuuta 2000

Historiaa

Tilan ja I/O:n kuvaaminen funktionaalisesti:

- ohjelma on funktio input-virralta output-virralle
- ohjelma on funktio vastausvirralta pyyntövirralle
- ”World”-muuttuja kaikkiin funktioihin
- *linear logic*: yksi World-muuttuja kerrallaan olemassa
- kontinuaatiot ja *continuation passing style*
- monadit

Monadit:

- käsite lähtöisin kategoriateoriasta (Kleisli, . . .)
- työkalu denotationaaliseen semantiikkaan (Moggi, 1989)
- poikkeusten käsittelyyn funktionaalisesti (Spivey, 1990)
- yleistyökalu funktionaaliseen ohjelmointiin (Wadler, 1990)

Esimerkki (Haskell)

```
data Maybe a = Just a | Nothing

doQuery :: Query -> DB -> Maybe Record

r :: Maybe [Record]
r = case doQuery initialQuery db of
      Nothing -> Nothing
      Just r1 -> case doQuery (nextQuery r1) db of
                   Nothing -> Nothing
                   Just r2 -> case doQuery (nextQuery2 r2) db of
                                Nothing -> Nothing
                                Just r3 -> Just [r1,r2,r3]
```

```
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
mB `thenMB` f = case mB of
                  Nothing -> Nothing
                  Just a -> f a

r = doQuery initialQuery db    `thenMB` \r1 ->
    doQuery (nextQuery r1) db  `thenMB` \r2 ->
    doQuery (nextQuery2 r2) db `thenMB` \r3 ->
    Just [r1,r2,r3]
```


Määritelmiä, monadi vs. monoidi

Monoidi: $(\mathbf{S}, \otimes, 1)$, esim. $(\mathbb{R}, \cdot)$

$$\begin{aligned} 1 \otimes m &= m \\ m \otimes 1 &= m \\ m \otimes (n \otimes o) &= (m \otimes n) \otimes o \end{aligned}$$

Monadi: $(\mathbf{M}, \texttt{>>=}, \texttt{return})$

$$\begin{aligned} \texttt{return a >>= f} &= \texttt{f a} \\ \texttt{p >>= return} &= \texttt{p} \\ \texttt{p >>= (\textbackslash a -> (f a >>= g))} &= \texttt{(p >>= (\textbackslash a -> f a)) >>= g} \end{aligned}$$

- monadi-käsite kategoriateoriasta
- `>>=` (bind, then) yhdistää monadin laskentaan, sitoen sen arvon ”kontinuaatiofunktion” argumentiksi
- `return` (unit) ottaa arvon ulos monadista

Monadi Haskellissa

```
class  Monad m  where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
    fail    :: String -> m a
    m >> k  =  m >>= \_ -> k
    fail s  =  error s
```

- tyyppiluokan `Monad` instanssien pitää toteuttaa edellä esitetyt lait (ohjelmoijan vastuulla)
- `>>` kuten `>>=` mutta ei sido arvoa (asettaa monadit ”peräkkäin”)
- monadin määrittely esimerkiksi:

```
instance Monad Maybe where
    (>>=) = thenMB
    return a = Just a
```


Monadi Haskellissa: do-syntaksi

- Maybe-monadin käyttäminen ilman do:ta:

```
r = doQuery initialQuery db    >>= \r1 ->
    doQuery (nextQuery r1) db  >>= \r2 ->
    doQuery (nextQuery2 r2) db >>= \r3 ->
    return [r1,r2,r3]
```

- sama do:n avulla:

```
r = do r1 <- doQuery initialQuery db
       r2 <- doQuery (nextQuery r1) db
       r3 <- doQuery (nextQuery2 r2) db
       return [r1,r2,r3]
```

- vrt. imperatiiviseen koodiin (pseudo-Javaa):

```
try { r1 = doQuery(initialQuery, db);
      r2 = doQuery(nextQuery(r1), db);
      r3 = doQuery(nextQuery2(r2), db);
      return list(r1, r2, r3);
} catch (QueryFailedException e) { return null; }
```


Monadi Haskellissa, huomioitavaa

- do sitoo kuten `let`, ts. `<-` ei ole sijoituslause
- monadit eivät vaadi tukea ohjelmointikieleltä (paitsi ensimmäisen luokan funktiot ja leksikaalisen sidonnan)
  - tyyppijärjestelmä auttaa (paljon)
  - hyödyllisimmillään puhtaasti funktionaalisissa kielissä
  - laiska evaluointi ei pakollista
- do-syntaksi muodostaa ”kielen”, joka näyttää imperatiiviselta, mutta ei ole sitä
  - monadit kuljettavat tilaa mukanaan ”piilossa”
  - myös kontrollirakenteita (esim. poikkeukset, epädeterminismi) voi toteuttaa monadeilla

Monadinen I/O

- Haskellin `IO`-monadissa imperatiivisia I/O-primitiivejä: `getChar`, `putChar`, `openFile`, `hGetChar`, `getContents`, ...
- `IO`-monadi näkyy primitiivien tyypissä, esim. `getChar :: IO Char`, `putChar :: Char -> IO ()`, ja leviää primiitiivejä käyttäviin funktioihin
- Haskell-toteutus osaa suorittaa `IO ()`-tyypin ”funktion” (tapahtuman, *action*), ohjelma alkaa tapahtumasta `main`
- esim. yhden merkin `cat`:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

Monadinen I/O (jatkoa)

- `main` on siis `IO`-monadin "jäsen", ja kuvaa toteutettavat operaatiot (`>>=` sitoo ne yhteen)
- `main` evaluoituu tapahtumaksi (tyyppi `IO ()` = tapahtuma, joka palauttaa arvon `()` eli "ei mitään"), jonka Haskell-tulkki suorittaa varsinaisen evaluoinnin ulkopuolella
- edellinen esimerkki aukikirjoitettuna:
  `main = getChar >>= \c -> putChar c`
- `getChar :: IO Char` on tapahtuma, joka suoritettuaan palauttaa `Char`in; `>>=` antaa tämän `Char`in funktiolle, joka evaluoituu seuraavaksi suoritettavaksi tapahtumaksi, jonka paluuarvo `()` on `main`-tapahtuman paluuarvo
- `>>` unohtaa paluuarvon eli asettaa tapahtumia peräkkäin (vrt. C:n puolipiste); esim. `putChar 'x' >> putChar 'y' >> putChar 'z'` tulostaa `xyz`

Lisää monadien sidonnasta

- `main = do putChar getChar` ei toimi, koska `putChar` haluaa `Char`-tyypin argumentin, mutta `getChar` on tyyppiä `IO Char` (tapahtuma eikä merkki)
- `main = getChar >>= putChar` toimii
- monadin palauttama arvo pitää siis sitoa suoraan johonkin (`>>=`:n oikean puolen argumentiksi), jos sitä haluaa käyttää
- käytännössä tämä tarkoittaa sitä, että tapahtumien täydellinen evaluaatiojärjestys pitää määrätä
  - `[getChar, getChar, getChar]` ei siis toimi, vaan:
    ```
    getChar >>= \a -> getChar >>= \b -> getChar >>= \c ->
      return [a, b, c]
    ```
  - vrt. C: `fgets(s, 100, fp);`, jossa luettu rivi pitää heti kirjoittaa jonnekin
- edellä `return` teki listasta tapahtuman, joka sisältää sen

I/O-esimerkki

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)

sumInts :: IO Int
sumInts = do n <- getInt
             if n == 0
                then return 0
                else (do m <- sumInts
                         return (n+m))

sumInteract :: IO ()
sumInteract = do putStrLn "Anna lukuja, 0 lopettaa"
                 sum <- sumInts
                 putStr "Summa on "
                 print sum
                 sumInteract
```


Tilan talletus monadiin

```
type S = Integer
data SM a = SM (S -> (a,S))
instance Monad SM where
  SM c1 >>= fc2 =  SM (\s0 -> let (r,s1) = c1 s0
                                  SM c2 = fc2 r in
                                 c2 s1)
  return k      =  SM (\s -> (k,s))

readSM          :: SM S
readSM          =  SM (\s -> (s,s))
updateSM        :: (S -> S) -> SM ()
updateSM f      =  SM (\s -> ((), f s))
runSM           :: S -> SM a -> (a,S)
runSM s0 (SM c) =  c s0

test init = runSM init      -- test 0 => ([0,1],1)
                  (do a <- readSM
                      updateSM (+1)
                      b <- readSM
                      return [a,b])
```

MonadPlus

Monadeilla voi olla myös nolla-alkio mzero ja operaatio mplus:

```
class  (Monad m) => MonadPlus m  where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Seuraavat lait tule toteuttaa (monadilakien lisäksi):

```
m >>= \x -> mzero  =  mzero
      mzero >>= m  =  mzero
  m `mplus` mzero  =  m
  mzero `mplus` m  =  m
```

- tämä siis lisää monadiin toisen operaation; vrt. kunta ryhmäteoriassa

Lista on monadi

```
instance  Monad []  where   -- melkein Haskellia
    m >>= k  = concat (map k m)
    return x = [x]
    fail s   = []
instance  MonadPlus []  where
    mzero    = []
    mplus    = (++)
```

- listamonadi kuvaa epädeterminististä (monta arvoa palauttavaa) laskentaa: lista tulosvaihtoehdoista
- `>>=` kuljettaa arvolistaa eteenpäin
- `return` palauttaa yhden arvon, `fail` ei yhtään
- `mplus` on *tai*-operaatio: lasketaan molemmat operandit ja laitetaan tuloslistat peräkkäin
- joskus määritellään myös operaatio `(+++)`, joka on kuten `mplus` mutta valitsee tuloslistoista ensimmäisen tuloksen

Monadinen jäsennys

- tapa tehdä *recursive descent* -jäsennystä monadien avulla
  - ei siis tehokkain eikä ilmaisuvoimaisin mahdollinen jäsennin, mutta kätevä kokeiluissa
  - monadiratkaisu tukee moniselitteisiä kielioppeja hyvin
- jäsennin on monadi:
  `newtype Parser a = Parser (String -> [(a,String)])`
- funktion argumenttina jäsennettävä merkkijono
- palauttaa listan tulosvaihtoehtoja; tuloksessa on jäsennetty alkio (tyyppiä a) ja jäljellä oleva merkkijono
- tyhjä lista: kieliopin mukaista jäsennystä ei ole
- monta alkiota listassa: monta jäsennysvaihtoehtoa

Jäsentimen määrittely (Haskell)

```
import Monad    -- Haskellin MonadPlus-määrittely
newtype Parser a = Parser (String -> [(a,String)])

parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p

instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f  = Parser (\cs -> concat [parse (f a) cs' |
                              (a,cs') <- parse p cs])
instance MonadPlus Parser where
  mzero       = Parser (\cs -> [])
  p `mplus` q = Parser (\cs -> parse p cs ++ parse q cs)

(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p `mplus` q) cs of
                           []     -> []
                           (x:xs) -> [x])
```

Jäsentimen osia

```
item   :: Parser Char               -- jäsennä jokin merkki
item  = Parser (\cs -> case cs of
                         ""     -> []
                         (c:cs) -> [(c,cs)])
sat    :: (Char -> Bool) -> Parser Char  -- ehdollinen jäsennys
sat p   = do {c <- item; if p c then return c else mzero}
char   :: Char -> Parser Char       -- jäsennä tietty merkki
char c  = sat (c ==)
string :: String -> Parser String   -- jäsennä tietty merkkijono
string ""     = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}

many   :: Parser a -> Parser [a]    -- toista jäsennystä (0..n)
many p  = many1 p +++ return []     -- +++ tekee ahneeksi
many1  :: Parser a -> Parser [a]    -- toista jäsennystä (1..n)
many1 p = do {a <- p; as <- many p; return (a:as)}

space   = many (sat isSpace)
token p = do {a <- p; space; return a}
symb cs = token (string cs)         -- tietty symbolinen token
```


Jäsenninesimerkki

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

-- expr ::= expr addop term | term
-- chainl1 tekee tämän, poistaen vasemman rekursion
expr   = term   ‘chainl1‘ addop
term   = factor ‘chainl1‘ mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDigit); return (ord x - ord ’0’)}

addop  = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop  = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```


Monadien sovelluksia

- funktionaalinen I/O, graafiset käyttöliittymät
- muu ulkoinen tila, esim. *Foreign Function Interface*
- poikkeukset, virheilmoitukset, `Maybe`
- sisäisen tilan kuljetus, esim. ympäristömuuttujat, laskurit
- epädeterministinen laskenta
- jäsennys
- rinnakkaisuus
- muokattavat (*mutable*) tietorakenteet

Tulevaisuus

Wadler, *The essence of functional programming*, 1992:

- monadien rajat: mitä kielen ominaisuuksia monadeilla voi kuvata?
- tarvitaanko syntaktista tukea? (esim. do)
- tehokkuus? (käyttävät paljon korkeamman kertaluvun funktioita)
- erilaisten monadien yhdistäminen? (ks. viite)
- kaivataanko primitiivimonadeja? (muita kuin I/O)
- muitakin uusia tilanhallintamenetelmiä on
- voiko tyyppipäättelyä laajentaa tunnistamaan ”epäpuhtaudet”?

Muita:

- onko käsite intuitiivinen? (*melkein* imperatiivista)
- kaikki tai ei mitään -ratkaisu: vaikea lisätä monadisia ominaisuuksia ohjelman sisälle

Lisätietoja

- Hudak, Peterson, Fasel: *A Gentle Introduction to Haskell*, 1999. `http://haskell.org/tutorial`
- Hutton, Meijer: *Functional Pearls: Monadic Parsing in Haskell*, Journal of Functional Programming, **1**(1), 1993.
- Hutton, Meijer: *Monadic Parser Combinators*, NOTTCS-TR-96-3, University of Nottingham, 1996.
- King, Wadler: *Combining Monads*, Functional Programming, Glasgow, 1992.
- Wadler: *The Essence of Functional Programming*
- Wadler: *How to Declare an Imperative*
- Wadler: *Monads for functional programming*
- Wadler: *Monads* (lista artikkeleista), `http://www.cs.bell-labs.com/who/wadler/topics/monads.html`
- Winstanley: *Introduction [to monads]*, `http://www.dcs.gla.ac.uk/~nww/Monad.html`