o valoare în interiorul valorii monadice (ca intr-un conatainer). Declarația operatorului `>>= :: ma >>= \v -> mb` arată ca el combină o valoare monadică `ma` ce conține valori de tipul `a` și o funcție ce operează pe o valoare `v` de tipul `a`, returnând o valoare monadică `mb`. Rezultatul constă în combinarea ma și f::: a -> mb într-o valoare monadică ce conține b. Operatorul `>>` este folosit când funcția nu are nevoie de valoarea produsă de primul operator monadic, fiind invariabilă în unicul său argument.

Semnificația/ Implementarea concretă a operatorului `>>=` depinde, bineînțeles, de monada propriu- zisă. Ca exemplu în monada `IO`, `x >>= y` va executa două operații de IO succesive, trecând rezultatul din prima în cea de a doua. Pentru celelalte monade predefinite, listele și monada `Maybe`, operatorii monadici pot fi înțelesi ca operând cu zero sau mai multe valori de la o etapa de calcul la următoarea. Vom clarifica puțin mai încolo prin câteva exemple.

`do` – notația permite o abreviere a lanțului de operatori monadici. Exprimarea sa este dată de următoarele (doua deocamdată) reguli:

```
do e1 ; e2        =     e1 >> e2
do p <- e1; e2    =     e1 >>= \p -> e2
```

În cazul în care pentru a doua formă a do-notației șablonul este incorect, se apelează operația `fail`. Acest lucru poate genera o eroare (în cazul monadei `IO`) sau să returneze valoarea “nulă” (în cazul monadei listă) . Astfel o exprimare completă ar fi:

```
do p <- e1; e2    =     e1 >>= (\v -> case v of p -> e2 ; _ -> fail "s")
```

unde `"s"` este un string care indică locația declarației `do` pentru un eventual mesaj de eroare. Ca exemplu, în cazul monadei IO o declarație de tipul `'a' <- getChar` va apela `fail` atunci cand caracterul introdus **nu este** `'a'`. Acest lucru va duce la oprirea programului deoarece la monada de `I/O` `fail` va apela `error`.

Regulile pentru `>>=` și `return` sunt:

```
return a >>= k                  = k a
m >>= return                    = m
xs >>= return . f               = fmap f xs
m >>= (\x -> k x >>= h)         = (m >>= k) >>= h
```

Clasa MonadPlus este folosită pentru implementarea monadelor care au element `zero` si operatorul `plus,` avand deja definite ca atare în declaratia de clasa:

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

La aceste monade elementul zero se supune următoarelor reguli:

```
m >>= (\x -> mzero)       = mzero
mzero >>= m               = mzero
```

În cazul listelor, elementul zero este `[]`, lista vidă. Monada `I/O` nu are element zero si deci nu va apartine acestei clase de tipuri.

Regulile care descriu comportarea lui `mzero` in raport cu adunarea din monada (operatorul `` `mplus` ``) sunt: `m `mplus` mzero = m`

```
mzero `mplus` m = m
```

În cazul monadei listelor, operatorul `` `mplus` `` este (echivalent cu) operatia de concatenare.

9.2 Monade predefinite

Având la dispoziție operatorii monadici și regulile acestora, la ce se pot folosi: Am particularizat deja cazul monadei `I/O` așa că vom porni cu alte două monade predefinite.

În cazul listelor, legarea monadică (adica operatorul bind) presupune gruparea unui set de calcule asupra fiecarei valori din lista. Când este implementat pentru liste, operatorul `>>=` are tipul:

```
(>>=)                 :: [a] -> (a -> [b]) -> [b]
```

Altfel spus, dată fiind o lista de `a` şi o funcție care duce un `a` într-o lista de `b`, operatorul de legare aplică aceasta funcție fiecărui `a` din intrare şi produce o listă cu toți `b` obținuți. Funcția `return` va crea o lista `singleton`. Aceste operații ar trebui să fie deja familiare: multimile ordonate definite implicit (eng. list comprehensions) pot fi exprimate uşor cu ajutorul operatorilor monadici pentru liste. Astfel, pentru liste, urmatoarele trei expresii sunt echivalente:

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
(\r -> case r of True -> return (x,y)
                 _ -> fail "")))
```

Această ultimă exprimare depinde de definirea lui `fail` în aceasta monadă ca fiind lista vidă. Concret, fiecare `<-` produce un set de valori care este transmis mai departe în calculul monadic.

Astfel `x <- [1,2,3]` apelează calculul monadic de trei ori, câte odată pentru fiecare element din listă. Expresia returnată `(x,y)` va fi evaluată pentru toate combinațiile posibile de legări care o incadrează. Din acest punct de vedere, monada listă poate fi văzută ca un grup de funcții cu argumente multi-valoare. Ca exemplu această funcție (n.n. Funcția proiectează operații binere în universul monadei.):

```
mvLift2       :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y  = do x' <- x
                    y' <- y
                    return (f x' y')
```

transformă o funcție uzuală de doua argumente (f) într-o funcție ce lucrează cu valori (n.n. monadice) multiple (liste de argumente), returnând o valoare pentru fiecare combinație posibilă a celor două argumente. Ca exemplu:

```
mvLift2 (+) [1,3] [10,20,30]      ➔   [11,21,31,13,23,33]
mvLift2 (\a b->[a,b]) "ab" "cd"   ➔   ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []            ➔   []
```

Această funcție `mvLift2` este o versiune specializată a funcției `LiftM2` din biblioteca de funcții monadice. Ea se poate considera ca transportând o funcție externă monadei listelor, `f`, în monada listelor - caz în care ea va servi la a prelucra valori multiple.

Monada definită peste tipul `Maybe` este similară cu monada listelor (mai exact a listelor de lungime 1): valoarea `Nothing` ne servește cum ne servea `[]` la liste iar `Just x` este pe postul lui `[x]`.