

Lihtne funktsionaalkeel

Süntaks

```
type Var    = String
data Term = V Var
          | Term :@ Term
          | L Var Term
          | N Integer
          | TT | FF
          | If Term Term Term
          | Rec Term
```

Lihtne funktsionaalkeel

Semantilised kategooriad

```
data Val = I Integer  
         | B Bool  
         | F (Val → Val)  
type Env = [(Var, Val)]
```

Lihtne funktsionaalkeel

Keskkonnaga opereerimine

$empty :: [(a, b)]$

$empty = []$

$update :: a \rightarrow b \rightarrow [(a, b)] \rightarrow [(a, b)]$

$update\ a\ b\ abs = (a, b) : abs$

$unsafeLookup :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow b$

$unsafeLookup\ x\ ((a, b) : abs)$

| $x == a \quad = b$

| $otherwise = unsafeLookup\ x\ abs$

Lihtne interpretaator

Standardinterpretaator

```
ev :: Term → Env → Val
ev (V x)      env = unsafeLookup x env
ev (e0 :@ e1) env = case ev e0 env of
                        F f → f (ev e1 env)
ev (L x e)    env = F (λv → ev e (update x v env))
ev (N n)      env = I n
ev TT         env = B True
ev FF         env = B False
ev (If e e0 e1) env = case ev e env of
                        B b → if b then ev e0 env
                        else ev e1 env
ev (Rec e)    env = case ev e env of
                        F f → f (ev (Rec e) env)
```

Lihtne interpretaator

Algkeskkond

```
env0 = [( "+",    mkArithOp2 (+))
        ...
        , ("==",  mkCompOp2 (==))
        ...
        , ("||",  mkBoolOp2 (∨))
        , ("not", mkBoolOp1 (not))
      ]
```

```
where mkArithOp2 op = F $ λ(I i1) →
                                F $ λ(I i2) → I (i1 'op' i2)
      mkCompOp2 op = F $ λ(I i1) →
                                F $ λ(I i2) → B (i1 'op' i2)
      mkBoolOp2 op  = F $ λ(B b1) →
                                F $ λ(B b2) → B (b1 'op' b2)
      mkBoolOp1 op  = F $ λ(B b)  → B (op b)
```

```
eval t = ev t env0
```

Lihtne interpretaator

Näide

```
fact = Rec $ L "fact" $ L "x" $  
    If ( V "<=" :@ V "x" :@ N 1)  
        ( N 1)  
        ( V "*" :@ V "x"  
          :@ ( V "fact" :@ ( V "-" :@ V "x" :@ N 1)))
```

```
Main> eval $ fact :@ N 5  
120
```

Monaadilised interpretaatorid

Semantilised kategooriad

```
data Val t = I Integer  
           | B Bool  
           | F (Val t → t (Val t))  
type Env t = [(Var, Val t)]
```

Väärtustusmonaad

```
class Monad t ⇒ MonadEv t where  
    ev :: Term → Env t → t (Val t)  
eval t = ev t env0
```

Monaadilised interpretaatorid

Algkeskkond

```
env0 = [( "+",    mkArithOp2 (+))
        ...
        , ("not", mkBoolOp1 (not))
        ]

where mkArithOp2 op = F $ λ(I i1) → return $
                                F $ λ(I i2) → return $
                                    I (i1 'op' i2)
mkCompOp2 op = F $ λ(I i1) → return $
                                F $ λ(I i2) → return $
                                    B (i1 'op' i2)
mkBoolOp2 op  = F $ λ(B b1) → return $
                                F $ λ(B b2) → return $
                                    B (b1 'op' b2)
mkBoolOp1 op  = F $ λ(B b)  → return $
                                B (op b)
```

Monaadilised interpretaatorid

Baaskonstruksioonide interpreteerimine

```
_ev :: MonadEv t => Term -> Env t -> t (Val t)
_ev (V x)      env = return $ unsafeLookup x env
_ev (e0 :@ e1) env = do F f <- ev e0 env
                      v   <- ev e1 env
                      f v
_ev (L x e)     env = return $ F (\v -> ev e (update x v env))
_ev (N n)       env = return $ I n
_ev TT          env = return $ B True
_ev FF          env = return $ B False
_ev (If e e0 e1) env = do B b <- ev e env
                        if b then ev e0 env
                        else ev e1 env
_ev (Rec e)     env = do F f <- ev e env
                      v   <- _ev (Rec e) env
                      f v
```

Monaadilised interpretaatorid

Standardinterpretaator

```
newtype Id a = Id a  
instance Monad Id where  
    return a  = Id a  
    Id a >>= k = k a  
instance MonadEnv Id where  
    ev e env  = _ev e env
```

Monaadilised interpretaatorid

Süntaks: erindid

```
data Term = ...  
          | Raise  
          | Term 'Handle' Term
```

Erinditöötusega interpretaator

```
instance MonadEv Maybe where  
  ev Raise env          = mzero  
  ev (e0 'Handle' e1) env = ev e0 env 'mplus' ev e1 env  
  ev e env              = _ev e env
```

Monaadilised interpretaatorid

Süntaks: mittedeterminism

```
data Term = ...  
          | Deadlock  
          | Term 'Choice' Term
```

Mittedetermineeritud interpretaator

```
instance MonadEv [] where  
  ev Deadlock env      = []  
  ev (e0 'Choice' e1) env = ev e0 env ++ ev e1 env  
  ev e env              = _ev e env
```

Monaadilised interpretaatorid

Näited

```
Main> eval $ V "+" :@ N 2 :@ N 3 :: Id (Val Id)  
Id 5
```

```
Main> eval $ V "+" :@ N 2 :@ N 3 :: Maybe (Val Maybe)  
Just 5
```

```
Main> eval $ N 2 'Handle' N 3 :: Maybe (Val Maybe)  
Just 2
```

```
Main> eval $ Raise 'Handle' N 3 :: Maybe (Val Maybe)  
Just 3
```

```
Main> eval $ V "+" :@ N 2 :@ (N 3 'Choice' N 4) :: [Val []]  
[5, 6]
```

Monaadilised interpretaatorid

Näited

```
Main> eval $ fact :@ N 5 :: Id (Val Id)  
Id 120
```

NB!

Rekursiooni väärtustamine toimub liiga agaralt!

Monaadilised interpretaatorid

Näited

```
Main> eval $ fact :@ N 5 :: Id (Val Id)  
Id 120
```

```
Main> eval $ fact :@ N 5 :: Maybe (Val Maybe)  
ERROR – Control stack overflow
```

NB!

Rekursiooni väärtustamine toimub liiga agaralt!

Monaadilised interpretaatorid

Näited

```
Main> eval $ fact :@ N 5 :: Id (Val Id)  
Id 120
```

```
Main> eval $ fact :@ N 5 :: Maybe (Val Maybe)  
ERROR – Control stack overflow
```

NB!

Rekursiooni väärtustamine toimub liiga agaralt!

Monaadilised interpretaatorid

Monaadiline püsipunktioperaator

```
class Monad t  $\Rightarrow$  MonadFix t where  
    mfix :: (a  $\rightarrow$  t a)  $\rightarrow$  t a  
  
    fix :: (a  $\rightarrow$  a)  $\rightarrow$  a  
    fix f = f (fix f)  
  
instance MonadFix Id where  
    mfix f = fix (f  $\circ$  unId)  
    where unId (Id x) = x  
  
instance MonadFix Maybe where  
    mfix f = fix (f  $\circ$  unJust)  
    where unJust (Just x) = x  
  
instance MonadFix [] where  
    mfix f = fix (f  $\circ$  unSglt)  
    where unSglt [x] = x
```

Monaadilised interpretaatorid

Monaadiline püsipunktioperaator

```
instance MonadFix (( $\rightarrow$ ) e) where
```

```
  mfix f =  $\lambda e \rightarrow$  let a = f a e
```

```
    in a
```

```
instance MonadFix (StM s) where
```

```
  mfix f = StM $  $\lambda s \rightarrow$  let (a, s') = runStM (f a) s
```

```
    in (a, s')
```

Monaadilised interpretaatorid

Rekursiooni väärtustamine

```
class MonadFix t  $\Rightarrow$  MonadEv t where  
    ev :: Term  $\rightarrow$  Env t  $\rightarrow$  t ( Val t )  
    _ev :: MonadEv t  $\Rightarrow$  Term  $\rightarrow$  Env t  $\rightarrow$  t ( Val t )  
    ...  
    _ev ( Rec e ) env = do F f  $\leftarrow$  ev e env  
                           mfix f
```

Monaadilised interpretaatorid

Näited

```
Main> eval $ fact :@ N 5 :: Id (Val Id)
```

```
Id 120
```

```
Main> eval $ fact :@ N 10 :: Maybe (Val Maybe)
```

```
Just 3628800
```

```
Main> eval $ fact :@ N 20 :: [Val []]
```

```
[2432902008176640000]
```

Monaadilised interpretaatorid

Väljundmonaad

```
newtype Output a = O (a, String)

instance Monad Output where
    m >>= f  = let O (x1, o1) = m
                O (x2, o2) = f x1
                in O (x2, o1 ++ o2)
    return x = O (x, "")

output :: String → Output ()
output s = O ((), s)

instance MonadFix Output where
    mfix f = fix (f ∘ unO)
    where unO (O (x, o)) = x
```

Monaadilised interpretaatorid

Süntaks: väljund

```
data Term = ...  
          | Out Term
```

Varjatud parameetriga interpretaator

```
instance MonadEv Output where  
  ev (Out e) env = do v ← ev e env  
                    output (show v)  
                    return v  
  ev e env       = _ev e env
```

Monaadilised interpretaatorid

Süntaks: varjatud parameeter

```
data Term = ...  
          | UseX
```

Varjatud parameetriga interpretaator

```
instance MonadEv (( $\rightarrow$ ) Integer) where  
  ev UseX env = do x  $\leftarrow$  getEnv  
                  return (I x)  
  ev e env = _ev e env
```

NB!

Kasutab ülekattuvaid isendeid!

Monaadilised interpretaatorid

Süntaks: globaalne olek

```
data Term = ...  
          | GetS  
          | SetS Term
```

Varjatud parameetriga interpretaator

```
instance MonadEv (StM Integer) where  
  ev GetS env    = do x ← getStM  
                    return (I x)  
  ev (SetS e) env = do I x ← ev e env  
                    setStM x  
                    return (I x)  
  ev e env       = _ev e env
```

Komonaadid

Definitsioon

Komonaad kategoorial $\mathcal{C}$ koosneb:

- endofunktorist $D : \mathcal{C} \rightarrow \mathcal{C}$
- loomulikust teisendusest $\varepsilon_A : DA \rightarrow A$
 - nn. komonaadi **ühik**
- loomulikust teisendusest $\delta_A : DA \rightarrow DDA$
 - nn. komonaadi **komultiplikatsioon**

selliselt, et järgnevad diagrammid kommuteeruvad:

$$\begin{array}{ccccc} D^2A & \xleftarrow{\delta_A} & DA & \xrightarrow{\delta_A} & D^2A \\ & \searrow D\varepsilon_A & \parallel & \swarrow \varepsilon_{DA} & \\ & & DA & & \end{array}$$

$$\begin{array}{ccc} DA & \xrightarrow{\delta_A} & D^2A \\ \delta_A \downarrow & & \downarrow \delta_{DA} \\ D^2A & \xrightarrow{D\delta_A} & D^3A \end{array}$$

Komonaadid

Definitsioon

Ko-Kleisli kolmik kategoorial $\mathcal{C}$ koosneb:

- kujutisest $D : \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$
- $\text{Obj}(\mathcal{C})$ -indekseeritud kujutiste perest $\varepsilon_A : DA \rightarrow A$
- operatsioonist $-^\dagger$, mis igale morfismile $f : DA \rightarrow B$ seab vastavusse morfismi $f^\dagger : DA \rightarrow DB$
 - nn. komonaadi **ekstensioon**

selliselt, et iga $f : DA \rightarrow B$ ja $g : DB \rightarrow C$ korral kehtivad järgmised võrdused:

$$\begin{aligned}\varepsilon_A^\dagger &= id_{DA} \\ \varepsilon_B \cdot f^\dagger &= f \\ (g \cdot f^\dagger)^\dagger &= g^\dagger \cdot f^\dagger\end{aligned}$$

Komonaadid

Lemma

Komonaadid ja ko-Kleisli kolmikud on ekvivalentssed mõisted.

Tõestus

Olgu $f : DA \rightarrow B$ ja $g : A \rightarrow B$

- $(D, \varepsilon, \delta) \implies (D, \varepsilon, -^\dagger)$

$$f^\dagger = Df \cdot \delta_A$$

- $(D, \varepsilon, -^\dagger) \implies (D, \varepsilon, \delta)$

$$\begin{aligned} Dg &= (g \cdot \varepsilon_A)^\dagger \\ \delta_A &= id_{DA}^\dagger \end{aligned}$$

Komonaadid

Intuitsioon

- Komonaadid modellerivad kontekstist sõltuvaid arvutusi.
- Kui A on väärtusi esitav tüüp, siis DA on kontekstis sõltuv väärtus.
- Kontekstist sõltuv funktsioon tüübist A tüüpi B on funktsioon tüüpi $DA \rightarrow B$.
- Komonaadi ühik $\varepsilon_A : DA \rightarrow A$ ignoreerib konteksti ning väljastab puhta väärtuse.
- Komonaadi ekstensioonioon $f^\dagger : DA \rightarrow DB$ arvutab etteantud kontekstist sõltuva funktsiooni $f : DA \rightarrow B$ põhjal uue väärtuse säilitades konteksti.
- Konteksti muutvad arvutused on komonaadi spetsiifilised.

Komonaadid Haskellis

Komonaadide klass

```
class Comonad d where
  counit  :: d a → a
  cobind  :: (d a → b) → d a → d b
  cmap    :: Comonad d ⇒ (a → b) → d a → d b
  cmap f x = cobind (f ∘ counit) x
  cdup    :: Comonad d ⇒ d a → d (d a)
  cdup     = cobind id
```

NB!

Komonaadi defineerimisel peab kontrollima, et ta rahuldaks komonaadi seadusi.

Komonaadid Haskellis

Identsuskomonaad

```
instance Comonad Id where  
  counit (Id a) = a  
  cobind k d    = Id (k d)
```

NB!

Identsuskomonaad esitab "puhtaid" (so. kontekstist sõltumatuid) arvutusi.

Komonaadid Haskellis

Korrutiskomonaad

instance *Comonad* ((,) *e*) **where**

counit (_, *x*) = *x*

cobind *k d*@(*e*, _) = (*e*, *k d*)

askP :: (*e*, *a*) → *e*

askP (*e*, _) = *e*

localP :: (*e* → *e*) → (*e*, *a*) → (*e*, *a*)

localP *f* (*e*, *x*) = (*f e*, *x*)

NB!

Korrutiskomonaad esitab "tavakeskkonnast" sõltuvaid arvutusi.

Komonaadid Haskellis

Striimide komonaad

```
data Stream a = a :< Stream a
instance Comonad Stream where
    counit (x :< xs)      = x
    cobind k d@(x :< xs) = k d :< cobind k xs
fbyS :: a → Stream a → Stream a
fbyS x xs = x :< xs
nextS :: Stream a → Stream a
nextS (x :< xs) = xs
```

NB!

Mis liiki arvutusi esitab striimide komonaad?

Andmevooarvutused

Striimid ja signaalid

- Striimid esitavad diskreetse ajaga signaale.
- Striimid (lõpmatud jadad) A^ω on isomorfsed funktsioonidega $\mathbb{N} \rightarrow A$.

Isomorfismi realistsioon Haskellis

```
str2fun :: Stream a → Int → a
str2fun (x :< xs) 0 = x
str2fun (x :< xs) n = str2fun xs (n - 1)

fun2str :: (Int → a) → Stream a
fun2str f = fun2str' f 0
  where fun2str' f n = f n :< fun2str' f (n + 1)
```

Andmevooarvutused

Striimifunktsioonid ja andmevooarvutused

- Andmevooarvutused on (diskreetse ajaga) signaalide teisendajad.
- Ehk siis striimifunktsioonid $A^\omega \rightarrow B^\omega$.

Näiteid andmevooprogrammidest

```
pos    = 0 fby (pos + 1)  
sum x = x + (0 fby (sum x))  
fact   = 1 fby (fact * (pos + 1))  
fibo   = 0 fby (fibo + (1 fby fibo))
```

<i>pos</i>	0	1	2	3	4	5	6	...
<i>sum pos</i>	0	1	3	6	10	15	21	...
<i>fact</i>	1	1	2	6	24	120	720	...
<i>fib</i> o	0	1	1	2	3	5	8	...

Andmevooarvutused

Striimifunktsioonid

Striimifunktsioonid on loomulikult isomorfsed funktsioonidega:

$$\begin{aligned} A^\omega \rightarrow B^\omega &\cong A^\omega \rightarrow (\mathbf{N} \Rightarrow B) \\ &\cong A^\omega \times \mathbf{N} \rightarrow B \end{aligned}$$

Positsiooniga striimide komonaad

Striimid koos positsiooniga moodustavad komonaadi:

$$\begin{aligned} \mathbf{D}A &= A^\omega \times \mathbf{N} \\ \varepsilon_A(\alpha, n) &= \alpha_n \\ f^\dagger(\alpha, n) &= (\beta, n) \\ &\text{kus } \forall i. \beta_i = f(\alpha, i) \end{aligned}$$

NB!

Striimifunktsioonid vs. olekuteisendajad.

Andmevooarvutused

Positsiooniga striimid

- Positsiooniga striimis (α, n) vastab striim α striimifunktsiooni sisendile ning positsioon n on väljundstriimi meid hetkel huvitava elemendi indeks.
- Sisendstriimi saame jaotada positsiooni suhtes kolmeks:

$$a_0, a_1, \dots, a_{n-1}, \boxed{a_n}, a_{n+1}, a_{n+2}, \dots$$

- ajalugu — antud positsioonist väiksemate indeksitega elemendid (neid on lõplik hulk);
- olevik — antud positsioonis olev element (täpselt üks);
- tulevik — antud positsioonist suuremate indeksitega elemendid (neid on lõpmatu hulk).

Andmevooarvutused

Positsiooniga striimide alternatiivseid esitusi

$$\begin{aligned}DA &= A^\omega \times \mathbf{N} \\ &\cong (\mathbf{N} \Rightarrow A) \times \mathbf{N} \\ &\cong A^* \times A \times A^\omega \\ &\cong A^+ \times A^\omega \\ &\cong A^* \times A^\omega\end{aligned}$$

Andmevooarvutused

Kausaalsed striimifunktsioonid

Sünkroonsed andmevooarvutused esitavad mitte üldisi vaid ainult kausaalseid striimifunktsioone

- s.o. hetke väljundväärtus võib sõltuda ainult hetke sisendist ning varasematest sisenditest, aga mitte tulevikust;
- seega, kausaalsed striimifunktsioonid on funktsioonid kujul $A^+ \rightarrow B$.

Mittetühjade listide komonaad

$$\begin{aligned} \text{DA} &= A^+ \\ \varepsilon_A[\alpha_1, \dots, \alpha_n] &= \alpha_n \\ f^\dagger[\alpha_1, \dots, \alpha_n] &= [f[\alpha_1], f[\alpha_1, \alpha_2], \dots, f[\alpha_1, \dots, \alpha_n]] \end{aligned}$$

Andmevooarvutused

Antikausaalsed striimifunktsioonid

- Antikausaalsed on striimifunktsioonid, millede hetke väljundväärtus võib sõltuda ainult hetke sisendist ning tulevikus saabuvatest sisenditest, aga mitte minevikust.
- Pole just väga praktiline, aga ...
- antikausaalsed striimifunktsioonid on funktsioonid kujul $A^\omega \rightarrow B$.

NB!

Striimide komonaad esitab antikausaalseid striimifunktsioone!

Komonaadid Haskellis

FunArg komonaad

```
data FunArg a = (Int → a) :# Int
instance Comonad FunArg where
    counit (f :# i)    = f i
    cobind k (f :# i) = (λ i' → k (f :# i')) :# i
fbyFA :: a → FunArg a → a
fbyFA a (f :# 0)      = a
fbyFA _ (f :# (i + 1)) = f i
nextFA :: FunArg a → a
nextFA (f :# i) = f (i + 1)
runFA :: (FunArg a → b) → Stream a → Stream b
runFA k as = runFA' k (str2fun as :# 0)
    where runFA' k d@(f :# i) = k d <: runFA' k (f :# (i + 1))
```

Komonaadid Haskellis

LVS komonaad

```
data List a = Nil | List a :> a
data LV a   = List a := a
data LVS a = LV a :| Stream a

instance Comonad LVS where
  counit (az := a :| as) = a
  cobind k d = cobindL d := k d :| cobindS d
    where cobindL (Nil      := a :| as) = Nil
           cobindL (az' :> a' := a :| as) = cobindL d' :> k d'
           where d' = az' := a' :| (a :< as)
           cobindS (az := a :| (a' :< as')) = k d' :< cobindS d'
           where d' = az :> a := a' :| as'
```

Komonaadid Haskellis

LVS komonaadi operatsioonid

$fbvLVS :: a \rightarrow LVS\ a \rightarrow a$

$fbvLVS\ a0\ (Nil := _ : | _) = a0$

$fbvLVS\ _ ((_ :> a') := _ : | _) = a'$

$nextLVS :: LVS\ a \rightarrow a$

$nextLVS\ (_ := _ : | (a :< _)) = a$

$runLVS :: (LVS\ a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b$

$runLVS\ k\ (a' :< as') = runLVS'\ k\ (Nil := a' : | as')$

where $runLVS'\ k\ d@(az := a : | (a' :< as'))$
 $\quad = k\ d :< runLVS'\ k\ (az :> a := a' : | as')$

Komonaadid Haskellis

LV komonaad

instance *Comonad LV* **where**

counit ($_ := a$) = a

cobind k $d@(az := _)$ = *cobindL* k $az := k$ d

where *cobindL* k *Nil* = *Nil*

cobindL k ($az := a$) = *cobindL* k $az := k$ ($az := a$)

fbyLV :: $a \rightarrow LV\ a \rightarrow a$

fbyLV $a0$ (*Nil* := $_$) = $a0$

fbyLV $_$ ($(_ := a')$:= $_$) = a'

runLV :: $(LV\ a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b$

runLV k ($a' := as'$) = *runLV'* k (*Nil* := a') as'

where *runLV'* k $d@(az := a)$ ($a' := as'$)

= k $d := runLV'$ k ($az := a := a'$) as'

Komonaadilised interpretaatorid

Semantilised kategooriad

```
data Val d = I Integer  
           | B Bool  
           | F (d (Val d) → Val d)  
type Env d = [(Var, Val d)]
```

Väärtustukomonaad

```
class Comonad d ⇒ ComonadEv d where  
  ev :: Term → d (Env d) → Val d
```

Komonaadilised interpretaatorid

Baaskonstruktsioonide interpreteerimine

```
_ev :: ComonadEv d => Term -> d (Env d) -> Val d
_ev (V x)          denv = unsafeLookup x (counit denv)
_ev (e0 :@ e1)     denv = case ev e0 denv of
                           F f -> f (cobind (ev e1) denv)
_ev (L x e)        denv = ...
_ev (N n)          denv = I n
_ev TT             denv = B True
_ev FF             denv = B False
_ev (If e e0 e1)   denv = case ev e denv of
                           B b -> if b then ev e0 denv
                           else ev e1 denv
_ev (Rec e)        denv = ...
```

Komonaadilised interpretaatorid

Rekursiooni väärtustamine

$$_ev (Rec\ e)\ denv = \text{case } ev\ e\ denv\ \text{of} \\ F\ f \rightarrow f\ (cobind\ (_ev\ (Rec\ e))\ denv)$$

NB!

Komonaadide korral "naiivne" rekursiooni realisatsioon tööab;
sh. rekursioonioperaator on uniformne kõikidele komonaadidele.

Komonaadiline rekursioonioperaator

$$cfix \quad ::\ Comonad\ d \Rightarrow d\ (d\ a \rightarrow a) \rightarrow a \\ cfix\ f = counit\ f\ (cobind\ cfix\ f)$$

Komonaadilised interpretaatorid

Lambda-abstraktsiooni väärtustamine

$$_ev (L\ x\ e)\ denv = F\ (\lambda d \rightarrow ev\ e\ (extend\ x\ d\ denv))$$

NB!

- Funktsioon *extend* tüüp peaks olema:

$$\begin{aligned} extend :: Comonad\ d \Rightarrow Var \rightarrow d\ (Val\ d) \\ \rightarrow d\ (Env\ d) \rightarrow d\ (Env\ d) \end{aligned}$$

- Ta peab laiendama keskkonda selliselt, et muutuja *x* oleks seotud muutujas *d* "oleva" väärtusega.
- Seejuures peab väljundkontekst ühendama mõlemad sisendkontekstid.

Komonaadilised interpretaatorid

Klass *ComonadZip*

```
class Comonad d ⇒ ComonadZip d where  
  czip :: d a → d b → d (a, b)
```

NB!

Funktsioon *czip* peab rahuldama järgmist võrdust:

$$czip \circ (\lambda d \rightarrow (d, d)) = cmap (\lambda x \rightarrow (x, x))$$

Komonaadilised interpretaatorid

Kontekstide ühendamise identsuskomonaadis

```
instance ComonadZip Id where  
  czip (Id a) (Id b) = Id (a, b)
```

Kontekstide ühendamise striimide komonaadis

```
instance ComonadZip Stream where  
  czip (x :< xs) (y :< ys) = (x, y) :< czip xs ys
```

Kontekstide ühendamise *FunArg* komonaadis

```
instance ComonadZip FunArg where  
  czip (f :# i) (g :# j) | i == j = ( $\lambda n \rightarrow (f\ n, g\ n)$ ) :# i
```

Komonaadilised interpretaatorid

Kontekstide ühendamine LV komonaadis

```
zipL :: List a → List b → List (a, b)
zipL (xz :=> x) (yz :=> y) = zipL xz yz :=> (x, y)
zipL _           _         = Nil

instance ComonadZip LV where
  czip (xz := x) (yz := y) = zipL xz yz := (x, y)
```

Kontekstide ühendamine LVS komonaadis

```
instance ComonadZip LVS where
  czip (xz := x :| xs) (yz := y :| ys)
    = zipL xz yz := (x, y) :| czip xs ys
```

Komonaadilised interpretaatorid

Lambda-abstraktsiooni väärtustamine

```
class ComonadZip d  $\Rightarrow$  ComonadEnv d where  
    ev :: Term  $\rightarrow$  d (Env d)  $\rightarrow$  Val d  
    _ev :: ComonadEnv d  $\Rightarrow$  Term  $\rightarrow$  d (Env d)  $\rightarrow$  Val d  
    ...  
    _ev (L x e) denv = F ( $\lambda$ d  $\rightarrow$  ev e (cmap repair (czip d denv)))  
        where repair (a, env) = update x a env  
    ...
```

Komonaadilised interpretaatorid

Algkeskkond

```
env0 = [ ("+", mkArithOp2 (+))  
        , ...  
        , ("not", mkBoolOp1 not)  
        ]
```

```
where mkArithOp2 op = F $ λdi1 → let I i1 = counit di1  
                                in F $ λdi2 → let I i2 = counit di2  
                                in I (i1 'op' i2)
```

```
mkCompOp2 op = F $ λdi1 → let I i1 = counit di1  
                        in F $ λdi2 → let I i2 = counit di2  
                        in B (i1 'op' i2)
```

```
mkBoolOp2 op = F $ λdb1 → let B b1 = counit db1  
                        in F $ λdb2 → let B b2 = counit db2  
                        in B (b1 'op' b2)
```

```
mkBoolOp1 op = F $ λdb → let B b = counit db  
                        in B (op b)
```

Komonaadilised interpretaatorid

Standardinterpretaator

```
instance MonadEv Id where  
    ev e env = _ev e env  
evalI    :: Term → Val Id  
evalI e = ev e (Id env0)
```

Näide

```
Main> evalI $ fact :@ N 5  
120
```

Komonaadilised interpretaatorid

Süntaks: sünkroonsed andmevookeeled

```
data Term = ...  
          | Term 'Fby' Term
```

Kausaalsete striimifunktsioonidega interpretaator

```
instance ComonadEv LV where  
  ev (e0 'Fby' e1) denv = ev e0 denv  
                                'fbyLV' cobind (ev e1) denv  
  ev e                        denv = _ev e denv
```

Komonaadilised interpretaatorid

Kausaalsete striimifunktsioonidega interpretaator

$denv0L \quad :: \text{ComonadEv } d \Rightarrow \text{Int} \rightarrow \text{List} (\text{Env } d)$
 $denv0L \ 0 \quad = \text{Nil}$
 $denv0L \ n \quad = denv0L \ (n - 1) :> env0$
 $evalLV \quad :: \text{Term} \rightarrow \text{Int} \rightarrow \text{Val } LV$
 $evalLV \ e \ i \quad = ev \ e \ (denv0L \ i := env0)$
 $denv0S \quad :: \text{ComonadEv } d \Rightarrow \text{Stream} (\text{Env } d)$
 $denv0S \quad = env0 :< denv0S$
 $evRunLV \quad :: \text{Term} \rightarrow \text{Stream} (\text{Val } LV)$
 $evRunLV \ e \quad = runLV \ (ev \ e) \ denv0S$

Komonaadilised interpretaatorid

Näited

```
– pos = 0 'fby' (pos + 1)
pos    = Rec (L "pos" (N 0 'Fby' (V "pos" #+ N 1)))
– summ x = x + (0 'fby' summ x)
summ   = L "x" (Rec (L "sumx"
                    (V "x" #+ (N 0 'Fby' V "sumx"))))
– diff x = x – (0 'fby' diff x)
diff    = L "x" (V "x" #- (N 0 'Fby' V "x"))
– ini x = x 'fby' ini x
ini     = L "x" (Rec (L "inix" (V "x" 'Fby' V "inix")))
```


Komonaadilised interpretaatorid

Näitetestid

```
Main> evalLV pos 10
```

```
10
```

```
Main> evRunLV pos
```

```
0 :< (1 :< (2 :< (3 :< (4 :< (5 :< (6 :< (7 :< (8 :< ({ Interrupted! }
```

```
Main> evRunLV $ summ :@ pos
```

```
0 :< (1 :< (3 :< (6 :< (10 :< (15 :< (21 :< (28 :< ({ Interrupted! }
```

```
Main> evRunLV $ fakt
```

```
1 :< (1 :< (2 :< (6 :< (24 :< (120 :< (720 :< (5040 :< ({ Interrupted! }
```

```
Main> evRunLV $ fibo
```

```
0 :< (1 :< (1 :< (2 :< (3 :< (5 :< (8 :< (13 :< (21 :< ({ Interrupted! }
```

Komonaadilised interpretaatorid

Süntaks: üldised andmevookeeled

```
data Term = ...  
          | Term 'Fby' Term  
          | Next Term
```

Üldiste striimifunktsioonidega interpretaator

```
instance ComonadEv FunArg where  
  ev (e0 'Fby' e1) denv = ev e0 denv  
                        'fbyFA' cobind (ev e1) denv  
  ev (Next e)          denv = nextFA (cobind (ev e) denv)  
  ev e                 denv = _ev e denv
```

Komonaadilised interpretaatorid

Näide: filtreerimine

```
—  $x \text{ 'wvr' } y = \text{if } y \text{ then } x \text{ 'fby' } (\text{next } x \text{ 'wvr' next } y)$   
—  $\text{else } (\text{next } x \text{ 'wvr' next } y)$   
 $wvr = \text{Rec } (L \text{ "wvr" } (L \text{ "x" } (L \text{ "y" } ($   
     $\text{If } (V \text{ "y"})$   
         $(V \text{ "x" 'Fby' } (V \text{ "wvr" } :@ (\text{Next } (V \text{ "x"}))$   
             $:@ (\text{Next } (V \text{ "y"}))))$   
         $(V \text{ "wvr" } :@ (\text{Next } (V \text{ "x"}))$   
             $:@ (\text{Next } (V \text{ "y"}))))))$ 
```

Komonaadilised interpretaatorid

Näide: algarvud

```
— sieve x = x 'fby' sieve (x 'wvr' (x 'mod' ini x  $\neq$  0))  
sieve = Rec (L "sieve" (L "x" (  
    (V "x") 'Fby'  
    (V "sieve" :@  
        (wvr :@ V "x" :@  
            (V "x" #% (ini :@ (V "x"))  $\neq$  N 0))))))  
eratosthenes = sieve :@ (pos #+ N 2)
```