

Seisundi monaad

`State s a` -- s on seisundi tüüp; a väärtuse tüüp

- definitsioon

```
get :: State s s
put :: s -> State s ()
runState :: State a b -> a -> (a, b)
```

- Tavaliselt tehakse tüübisünonüüm iga konkreetse alamprogrammi jaoks.

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a
```

...

Seisundi monaad II

- Näide:

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a

applyIsik :: String -> (Double -> Double) -> PangaArvutus ()
applyIsik nimi f = do
  s <- get
  put (map g s)
  where g (n,s) | n==nimi    = (n, f s)
              | otherwise = (n, s)

rahaVälja :: String -> Float -> PangaArvutus Bool
rahaVälja nimi summa = do
  s <- get
  case lookup nimi s of
    Nothing -> return False
    Just r   -> do
      applyIsik nimi (\ x -> x-summa)
      return True
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)

get :: State s s
get s = (s, s)

put :: s -> State s a
put s _ = (s, ())

runState :: State s a -> s -> (s, a)
runState f = f

return :: a -> State s a
return x s = (s, x)

(>>=) :: State s a -> (a -> State s b) -> State s b
(f >>= g) s = g x s'
  where (s',x) = f s
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)
```

```
get :: s -> (s, s)
```

```
get x = (x, x)
```

```
put :: s -> s -> (s, ())
```

```
put s _ = (s, ())
```

```
runState :: (s -> (s, a)) -> s -> (s, a)
```

```
runState f = f
```

```
return :: a -> s -> (s, a)
```


```
return x s = (s, x)
```

```
(>>=) :: (s -> (s, a)) -> (a -> s -> (s, b)) -> s -> (s, b)
```

```
(f >>= g) s = g x s'
```

```
  where (s', x) = f s
```

IO modeleerimine seisundimonaadiga

type IO a = State  a

ehk

type IO a =  -> (, a)

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]

Main> sequence_ [print 1, print 'a']
1
'a'

Main> it
()

Main> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]

Main> sequence [Just 1, Nothing, Just 3]
Nothing
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline map

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Näited:

```
Main>mapM print [1..5]
1
2
3
4
5
[(), (), (), (), ()]
Main>mapM print (Just 6)
6
Just ()
```

Monaadilised funktsioonid standardprelüüdis

- "for"-tsükkel

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM x y = mapM x y
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ x y = mapM_ x y
```

Näited:

```
main = do
  forM_ [1..10] $ \ i -> do
    let n = fact i
        putStrLn $ show i ++ "! = " ++ show n
```


Monaadilised funktsioonid standardprelüüdis

- Tingimuslik täitmine

```
when :: Monad m => Bool -> m () -> m ()  
when p s = if p then s else return ()  
unless :: Monad m => Bool -> m () -> m ()  
unless p s = when (not p) s
```

Näited:

```
import System.Environment (getArgs)  
import System.Directory (doesDirectoryExist)  
  
main = do names <- getArgs  
          forM_ names $ \ dir -> do  
            b <- doesDirectoryExist dir  
            when b $ putStrLn dir
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline filter

```
filterM :: Monad m -> (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x : xs) = do
  b <- p x
  ys <- filterM p xs
  return (if b then x : ys else ys)
```

Näited:

```
main = do names <- getArgs
          dirs  <- filterM doesDirectoryExist names
          mapM_ putStrLn dirs
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (1)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do x <- m
            return (f x)
```

Näited:

```
countLines :: FilePath -> IO Int
countLines = liftM (length . lines) . readFile
```

Enamasti kasutatakse funktsiooni:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (2)

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do x1 <- m1
                  x2 <- m2
                  return (f x1 x2)
```

Näited:

```
Main> liftM2 (+) (Just 1) (Just 2)
Just 3
Main> liftM2 (+) (Just 1) Nothing
Nothing
Main> liftM2 (+) [0, 3] [5, 6, 7]
[5,6,7,8,9,10]
```

Analoogiliselt on defineeritud **liftM3**, **liftM4** ja **liftM5** .

Monaadilised funktsioonid standardprelүүdis

- Monaadiline aplikatsioon

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Näited:

```
Main> [(+2)] `ap` [1, 2, 3]
[3, 4, 5]
Main> [(+2), (*2)] `ap` [1, 2, 3]
[3,4,5,2,4,6]
```

```
liftMn f x1 x2 ... xn = return f `ap` x1 `ap` x2 `ap` ... `ap` xn
```

Aplikatsioon on võimalik ka osade mitte-monaadide puhul! (<*>)

```
return f `ap` x1 `ap` x2 `ap` ... `ap` xn = f <$> x1 <*> x2 <*> ... <*> xn
```

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega...
- ... aga kunagi tuleb aeg neid ühendada terviklikuks programmiks.
- Monaadide kombineerimiseks on monaaditeisendajad. Selle asemel, et kasutada monaadi `Maybe` kasutame mõnedel juhtudel monaadi `MaybeT m`, kus `m` on mingi teine monaad.
- Haskellis nõrkus — raske on ette näha, mis informatsiooni on vaja läbi programmi kaasas kanda.

QuickCheck on programmide automaatse testimise raamistik:

- Kontrollib, kas programm vastab etteantud **omadustele** kasutades **jhuslikult genereeritud** sisendeid.
- Algselt kirjutatud Haskellis, kuid nüüd ka Scalas, Javas jne.
- Teegis on:
 - omaduste kombineerimise kombinaatorid;
 - juhuslike väärtuste generaatorid standardtüüpide jaoks;
 - kombinaatorid oma andmetüüpide juhuslikuks genereerimiseks.

Liides

```
import Test.QuickCheck  
quickCheck :: Testable prop => prop -> IO ()
```

- Funktsioon `quickCheck` võtab argumendiks **omaduse** mida juhuslikel argumentidel kontrollitakse.
- Vaikimisi 100-l juhuslikul väärtusel.
- Kui test ebaõnnestub, siis trükitakse vastunäide.

Näide

reverse omadused

Katsetame ...

prop
prop

```
Main> quickCheck prop_RevRev  
+++ OK, passed 100 tests.
```

```
Main> quickCheck prop_RevApp  
+++ OK, passed 100 tests.
```

prop

```
prop_RevApp xs ys = reverse (xs ++ ys)  
                  == reverse ys ++ reverse xs
```

Näide (järg)

reverse omadused

$$\forall a. \forall b. \text{reverse } (a ++ b) = (\text{reverse } b) ++ (\text{reverse } a)$$

Katsetame ...

```

Main> quickCheck prop_RevWrong
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0]
[1]

```

```

prop_RevWrong      :: [Int] -> [Int] -> Bool
prop_RevWrong xs ys = reverse (xs ++ ys)
                    == reverse xs ++ reverse ys

```

Omadused

```
class Testable prop where
  property :: prop -> Property

instance Testable Bool

instance (Arbitrary a, Show a, Testable prop) =>
  Testable (a -> prop)
```

- Omadused on avaldised, mille tüüb on klassist Testable.
- Argumendid peavad olema monomorfsed tüüpi.
 - Vaja argumentide genereerimise jaoks.
- Nimed tava järgi prefiksiga prop_

Omadused

Näide: sorteerimine pistemeetodil

```
isort :: Ord a => [a] -> [a]
```

```
isort = foldr insert []
```

```
insert :: Ord a => a -> [a] -> [a]
```

```
insert x [] = [x]
```

```
insert x (y:ys) | x <= y = x : y : ys  
                | otherwise = y : insert x ys
```

Omadused

Omadus 1: tulemus peab olema järjestatud

```
prop_sortOrder :: [Int] -> Bool  
prop_sortOrder xs = ordered (isort xs)
```

```
ordered :: Ord a => [a] -> Bool  
ordered (x:y:ys) = x <= y && ordered (y:ys)  
ordered ys      = True
```

Omadused

Omadus 2: operatsioon ei lisa ega kustuta elemente

```
prop_sortElems :: [Int] -> Bool
```

```
prop_sortElems xs = sameElems xs (isort xs)
```

```
sameElems :: Eq a => [a] -> [a] -> Bool
```

```
sameElems xs ys = null (xs \\  
ys) && null (ys \\  
xs)
```

Omadused

Kui pikad on listid?

Mis täpselt olid argumendid?

```
Main> quickCheck (\ xs -> collect xs (p xs))  
+++ OK, passed 100 tests:  
8% []  
1% [97723,95805,-104521,45943,-73844,6249,64936]  
...
```

```
4% 4
```

Omadused

piste omadused: säilitab sorteerituse (ver. 1)

Problem!

```
Main> let p = prop_insertOrder1
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
+++ OK, passed 100 tests:
87% False
13% True
```


Omadused

Implikatsioon

```
(==>) :: Testable prop => Bool -> prop -> Property
```

```
instance Testable Property
```

- Kombinaator (`==>`) ignoreerib sisendit, kui eeldus pole täidetud, ning genereerib uue väärtused.
- Vaikimisi 500 korda.

Omadused

juba parem ...

```
Main> let p = prop_insertOrder2
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
*** Gave up! Passed only 82 tests (100% True).
```

Omadused

Univresaalne kvantifitseerimine

```
forall :: (Show a, Testable prop) =>  
  Gen a -> (a -> prop) -> property
```

- Kombinaator forall saab argumendiks generaatori, millega väärtusi luuakse.
- Saame luua väärtusi, mis vastavad valitud omadustele.

Omadused

pis
pro
pro

Töötab!!

```
Main> quickCheck (forAll orderedList ordered)
+++ OK, passed 100 tests.
Main> quickCheck prop_insertOrder3
+++ OK, passed 100 tests.
```

Generaatorid

Generaatorid

```
newtype Gen a = ...
```

```
instance Monad Gen
```

```
instance Functor Gen
```

```
instance (Testable prop) => Testable (Gen prop)
```

- Generaatorid on abstraktse tüübiga Gen.
- Gen on monaad, millel on juurdepääs juhuarvudele.

Generaatorid

Trükitab mõned genereeritud väärtused

```
sample :: Show a => Gen a -> IO ()
```

Generaatorite kombineerimine

```
choose    :: Random a => (a, a) -> Gen a  
elements :: [a] -> Gen a  
oneof    :: [Gen a] -> Gen a  
frequency :: [(Int, Gen a)] -> Gen a  
sized    :: (Int -> Gen a) -> Gen a  
vectorOf :: Int -> Gen a -> Gen [a]
```

Generaatorid

Vaike-generaatorid

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

shrink _ = []
```

- Vaike-generaatoriga tüübid kuuluvad klassi Arbitrary.
- Lisaks on klassis meetod shrink, mis vähendab väärtust.
 - shrink tagastab struktuurselt väiksemate väärtuste listi;
 - kui test ebaõnnestub, proovitakse shrink abil argumente vähendada.

Generaatorid

Lihtsad generaatorid

```
instance Arbitrary Bool where
  arbitrary = choose (False, True)
```

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
  arbitrary = do
    x <- arbitrary
    y <- arbitrary
    return (x,y)
```

```
data Color = Red | Blue | Green
```

```
instance Arbitrary Color where
  arbitrary = elements [Red, Blue, Green]
```


Generaatorid

Lihtsad generaatorid

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary ]
```

Probleem!

Parem **Pool väärtustest on Nothing!!**

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = frequency [ (1, return Nothing)
                        , (3, liftM Just arbitrary) ]
```

Generaatorid

Täisarvude genereerimine (ver. 1)

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)
```

Täisarvude genereerimine (ver. 2)

```
instance Arbitrary Int where
  arbitrary = sized (\ n -> choose (-n,n))
```

Generaatorid

Rekursiivsete andmetüüpide genereerimine (ver. 1)

```
data Tree a = Leaf a
```

Probleem!

```
instance Arbitrary a => Arbitrary (Tree a) where  
  arbitrary = frequency [(1, liftM Leaf arbitrary),  
                        (2, liftM2 Node arbitrary arbitrary)]
```

Termineerimine pole kindlustatud!

Generaatorid

Rekursiivsete andmetüüpide genereerimine (ver. 2)

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbitraryTree

arbitraryTree :: Arbitrary a => Int -> Gen (Tree a)
arbitraryTree 0 = liftM Leaf arbitrary
arbitraryTree n = frequency [ (1, liftM Leaf arbitrary)
                             , (4, liftM2 Node t t) ]
  where t = arbitraryTree (n `div` 2)
```

NB!

- Teine võrdus võib ka genereerida Leaf-e.
- Muidu genereeriks ainult tasakaalus puid.

Generaatorid

Eeldefineeritud erilised generaatorid

```
newtype OrderedList a = Ordered [a]  
instance (Ord a, Arbitrary a) => Arbitrary (OrderedList a)
```

```
newtype NonEmptyList a = NonEmpty [a]  
instance Arbitrary a => Arbitrary (NonEmptyList a)
```

```
newtype Positive a = Positive a  
instance (Num a, Ord a, Arbitrary a) => Arbitrary (Positive a)
```

```
newtype NonZero a = NonZero a  
newtype NonNegative a = NonNegative a
```

Funktsioonide genereerimine

```
class CoArbitrary a where
  coarbitrary :: a -> Gen b -> Gen b

instance (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

Example

```
variant :: Integral n => n -> Gen a -> Gen a

instance CoArbitrary a => CoArbitrary [a] where
  coarbitrary []      = variant 0
  coarbitrary (x:xs) = variant 1 . coarbitrary (x,xs)
```

- Soovitav kasutada variant et segada juhuslikku genereerimist.
- Erineva esimesed argumendi puhul püütakse tagastatakse erinev väärtus.