Uute tüüpide defineerimine

- Tüübisünonüümid:

$$\texttt{type}\ tcon\ tv_1 \ldots tv_n \quad = \quad texpr \qquad\qquad n \geq 0$$

- Algebralised andmetüübid:

$$\begin{array}{lll} \texttt{data}\ tcon\ tv_1 \ldots tv_n & = & dcon_1\ texpr_{11} \ldots texpr_{1m_1} \\ & | & \ldots \qquad\qquad n,m_1,\ldots,m_n \geq 0 \\ & | & dcon_n\ texpr_{n1} \ldots texpr_{nm_n} \end{array}$$

- "Uustüübid":

$$\texttt{newtype}\ tcon\ tv_1 \ldots tv_n \quad = \quad dcon\ texpr \qquad n \geq 0$$

Tüübisünonüümid

- Annab tüübiavaldisele nime, mis on esialgese tüübiavaldisega täiesti ekvivalentne
- Tüüp võib olla polümorfne, kuid mitte rekursiivne ja tüübikonstruktorit ei saa osaliselt rakendada
- Näited:

```
type String  = [Char]    -- eeldefineeritud
type AssocList a b = [(a,b)]
type Predicate a   = a -> Bool
```

Algebralised andmetüübid

- Loenditüübid

```
data Day = Mon | Tue | Wed | Thu
         | Fri | Sat | Sun
    deriving Show
```

- Nimed `Mon` – `Sun` on konstruktor-konstandid (kuna neil ei ole argumente) ja on tüübi `Day` ainsad elemendid.

- Näide:

```
valday :: Int -> Day
valday n = days !! (n-1)
   where days = [Mon,Tue,Wed,Thu,Fri,Sat,Sun]
```

Algebralised andmetüübid

- Funktsioonid defineeritakse näidiste sobitamise abil

```
workday :: Day -> Bool
workday Mon  = True
workday Tue  = True
workday Wed  = True
workday Thu  = True
workday Fri  = True
workday Sat  = False
workday Sun  = False
```

Algebralised andmetüübid

- Variant-kirjed

```
data Tagger = Tagn Integer | Tagb Bool
```

- Konstruktorid `Tagn` ja `Tagb` on funktsioonid

```
Tagn :: Integer -> Tagger
Tagb :: Bool    -> Tagger
```

- Erinevalt (tavalistest) funktsioonidest
  - on konstruktor-aplikatsioonid (näit. `Tagn 7`) kanoonilisel kujul (so. neid ei saa edasi lihtsustada);
  - saab konstruktoreid kasutada näidiste sobitamisel

Algebralised andmetüübid

- Näiteid:

```
number  (Tagn i) = i
boolean (Tagb b) = b

isNum (Tagn _) = True
isNum (Tagb _) = False
isBool x       = not (isNum x)

Hugs> :t number
number :: Tagger -> Integer
Hugs> number (Tagn 3)
3
```

Algebralised andmetüübid

- Polümorfsed andmetüübid

```
data Maybe a = Just a | Nothing
             deriving Show
```

- Konstruktorid `Just` ja `Nothing` on polümorfset tüüpi

```
Just    :: a -> Maybe a
Nothing :: Maybe a
```

- Polümorfsete konstantide “näitamine” nõuab tüübi ilmutatud deklareerimist (analoogselt tühja listiga)!

Ebaõnnestumise modelleerimine

- Otsida tabelist võtmele vastav väärtus

```
type Table = [(String,Int)]

lookup :: String -> Table -> Int
lookup key ((k,v):xs) | key == k  = v
                      | otherwise = lookup key xs
```

- Kui võtmele vastavat elementi pole, siis on viga!

```
lookup :: String -> Table -> Maybe Int
lookup key []                     = Nothing
lookup key ((k,v):xs) | key == k  = Just v
                      | otherwise = lookup key xs
```

Algebralised andmetüübid

- Rekursiivsed andemtüübid

```
data Tree a = Empty
            | Branch a (Tree a) (Tree a)
```

- Näide — puu "lamendamine"

```
flatten :: Tree a -> [a]
flatten Empty = []
flatten (Branch x t1 t2)
              = flatten t1 ++ [x] ++ flatten t2
```

- Konstruktori argumendid võivad olla "märgendatud"

```
data Tree a = Empty
     | Branch {val :: a, left, right :: Tree a}
```

"Uustüübid"

- Ühe unaarse konstruktoriga tüüp

```
newtype Table a b = MkTable [(a,b)]
```

- Erinevalt tüübisünonüümist võib olla rekursiivne ja saab osaliselt rakendada
- Erinevalt ühe unaarse konstruktoriga andmetüübist on konstruktor "virtuaalne"
- Konstruktori argument võib olla "märgendatud"

Geomeetrilised kujundid

- Kujundite esitamine

```
data Shape = Rectangle Side Side
           | Ellipse Radius Radius
           | RtTriangle Side Side
           | Polygon [Vertex]
     deriving Show

type Radius = Float
type Side   = Float
type Vertex  = (Float,Float)
```

Geomeetrilised kujundid

- Uute kujundite defineerimine:

```
square s = Rectangle s s
circle r = Ellipse r r
```

- Kujundite pindala:

```
area :: Shape -> Float
area (Rectangle  s1 s2) = s1*s2
area (RtTriangle s1 s2) = s1*s2/2
area (Ellipse r1 r2)    = pi*r1*r2
```

Geomeetrilised kujundid

- Kolmnurga pindala:

```
triArea :: Vertex -> Vertex -> Vertex -> Float
triArea v1 v2 v3 = let a = distBetween v1 v2
                       b = distBetween v2 v3
                       c = distBetween v3 v1
                       s = 0.5*(a+b+c)
                   in sqrt (s*(s-a)*(s-b)*(s-c))
```

- Lõigu pikkus

```
distBetween :: Vertex -> Vertex -> Float
distBetween (x1,y1) (x2,y2)
            = sqrt ((x1-x2)^2 + (y1-y2)^2)
```

Geomeetrilised kujundid

- Polügoni pindala:

```
area (Polygon (v1:vs))  = polyArea vs
   where polyArea :: [Vertex] -> Float
         polyArea (v2:v3:vs')
                          = triArea v1 v2 v3
                          + polyArea (v3:vs')
         polyArea _       = 0
```

Tüübiklassid

- Aritmeetilised operaatorid:

```
(+) :: Int   -> Int   -> Int
(+) :: Float -> Float -> Float
```

  - Tüüp `(+) :: a -> a -> a` on liiga üldine!

- Polümorfne võrdus:

```
(==) :: a -> a -> Bool
```

  - Kuidas kontrollida näiteks funktsioonide võrdust?

- Polümorfne väljastus:

```
show :: a -> String
```

  - Funktsioon `show` on igal tüübil defineeritud erinevalt!

Tüübiklassid

- Ülemääratud operaatoreid defineeritakse tüübiklassidega

  `class` [*context* `=>`] *classname tvar* `where` {*cbody*}

- Klassidefinitsiooni keha koosneb meetodide tüübisignatuuridest ja default-definitsioonidest.
- Näide — klass `Eq` (definitsioon prelüüdist):

```
class  Eq a  where
    (==), (/=)  :: a -> a -> Bool
    x /= y      =  not (x == y)
```

- Meetodide tüüpides esinevad tüübimuutujad on kitsendatud vastava klassikontekstiga.

```
(==) :: Eq a => a -> a -> Bool
```

Tüübiklassid

- Tüüpe saab kuulutada klassi kuuluvaks

  `instance` [*context* =>] *classname type* `where` {*ibody*}

- Deklareeritav tüüp peab olema kujul $tcon\ tvar_1 \ldots tvar_n$, kusjuures kõik tüübimuutujad peavad olema erinevad
- Deklaratsiooni keha koosneb meetodide definitsioonidest
- Default-definitsiooniga meetodi pole vaja (aga võib) uuesti defineerida

Tüübiklassid

- Näide — binaarpuude võrdus:

```
data IntBtree = Lf Int | Br IntBtree IntBtree

instance Eq IntBtree where
      Lf i1 ==  Lf i2       = i1 == i2
  Br t11 t12 == Br t21 t22  = t11 == t21 && t12 == t22
          _ == _            = False
```

- Näide — listide võrdus (defineeritud prelüüdis):

```
instance Eq a => Eq [a] where
     [] == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
      _ == _      = False
```

Tüübiklassid

- Näide — põõsaste võrdus:

```
data Bush a = One a | Two (Bush a) (Bush a)
                    | Many [Bush a]


instance Eq a => Eq (Bush a) where
      One x == One y      =     x == y
  Two x1 x2 == Two y1 y2  =    x1 == y1 && x2 == y2
    Many xs == Many ys    =    xs == ys
          _ == _          =    False
```

- On kasutatud kolme erinevat võrdust!!!

```
(==) :: a -> a -> Bool
(==) :: Bush a -> Bush a -> Bool
(==) :: [Bush a] -> [Bush a] -> Bool
```

Tüübiklassid

- Klassidest võib mõelda, kui predikaatidest üle tüüpide
- Tüübid, mis rahuldavad neid predikaate, omavad "lisa funktsionaalsust"
- Klasside deklaratsioonid defineerivad mis liiki "lisa funktsionaalsusega" on tegemist
- "Isendite" deklaratsioonid defineerivad selle "lisa funktsionaalsuse" konreetse tüübi jaoks

Tüübiklassid

- Klassikuuluvuse tuletamine:

```
data Bool = False | True
          deriving (Eq,Ord,Ix,Enum,Read,Show,Bounded)
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Read, Show)
data Either a b = Left a | Right b
                deriving (Eq, Ord, Read, Show)
data Ordering = LT | EQ | GT
          deriving (Eq,Ord,Ix,Enum,Read,Show,Bounded)
```

- `deriving` abil saab tuletada ainult eeldefineeritud klasse `Eq`, `Ord`, `Enum`, `Bounded`, `Show` ja `Read`.

Klass Ord

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
    compare x y | x == y    =  EQ
                | x <= y    =  LT
                | otherwise =  GT
     x <= y     = compare x y /= GT
     x <  y     = compare x y == LT
     x >= y     = compare x y /= LT
     x >  y     = compare x y == GT
     max x y    = if x >= y then x else y
     min x y    = if x <  y then x else y
```

Klass Enum

```
class  Enum a  where
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFrom        :: a -> [a]             -- [n..]
    enumFromThen    :: a -> a -> [a]        -- [n,n'..]
    enumFromTo      :: a -> a -> [a]        -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]    -- [n,n'..m]

succ, pred :: Enum a => a -> a
succ       =  toEnum . (+1) . fromEnum
pred       =  toEnum . (subtract 1) . fromEnum
```

Arvudega seotud klassid

- Klass `Num`

```
class  (Eq a, Show a) => Num a  where
     (+), (-), (*)    :: a -> a -> a
     negate           :: a -> a
     abs, signum      :: a -> a
     fromInteger      :: Integer -> a

     x - y            =  x + negate y
```

- Klass `Real`

```
class  (Num a, Ord a) => Real a  where
    toRational       ::  a -> Rational
```

Arvudega seotud klassid

- Klass `Integral`

```
class  (Real a, Enum a) => Integral a  where
     quot, rem          :: a -> a -> a
     div, mod           :: a -> a -> a
     quotRem, divMod    :: a -> a -> (a,a)
     toInteger          :: a -> Integer
```

- Klass `Fractional`

```
class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
    recip               :: a -> a
    fromRational        :: Rational -> a
```

- Lisaks on eeldefineeritud `RealFrac`, `Floating`, `RealFloat`

Klassid Show ja Read

```
type  ShowS    = String -> String

class  Show a  where
    showsPrec :: Int ->  a  -> ShowS
    showList  ::         [a] -> ShowS
    show      :: a -> String
    show x    =  showsPrec 0 x ""


class  Read a  where ...

read :: (Read a) => String -> a
```