

Uute tüüptide defineerimine

- Tüübisünonüümid:

$$\text{type } tcon\ tv_1 \dots tv_n = \text{texpr} \quad n \geq 0$$

- Algebralised andmetüübid:

$$\begin{array}{lcl} \text{data } tcon\ tv_1 \dots tv_n & = & dcon_1\ \text{texpr}_{11} \dots \text{texpr}_{1m_1} \\ & | & \dots \quad n, m_1, \dots, m_n \geq 0 \\ & | & dcon_n\ \text{texpr}_{n1} \dots \text{texpr}_{nm_n} \end{array}$$

- “Uustüübid”:

$$\text{newtype } tcon\ tv_1 \dots tv_n = dcon\ \text{texpr} \quad n \geq 0$$

Uute tüüpide defineerimine

Tüübisünonüümid

type *tcon* $tv_1 \dots tv_n = texpr$ $n \geq 0$

- Annab tüübiavaldisele nime, mis on esialgese tüübiavaldisega täiesti ekvivalentne
- Tüüp võib olla polümorfne, kuid mitte rekursiivne
- 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

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

Algebraised and metatheory

Funktsioonid defineeritakse näidiste sobitamise abil

weekday :: *Day* → *Bool*

weekday Mon = *True*

weekday Tue = *True*

weekday Wed = *True*

weekday Thu = *True*

weekday Fri = *True*

weekday Sat = *False*

weekday 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

Maybe tüüp (eeldefineeritud)

data *Maybe* *a* = *Just* *a* | *Nothing*

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

Just :: *a* → *Maybe* *a*

Nothing :: *Maybe* *a*

- *Maybe* tüüpi kasutatakse ebaõnnestumiste modelleerimiseks

Algebraised andmetüübid

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
```

NB!

Kui võtmele vastavat elementi pole, siis on viga!

Algebraised andmetüübid

Ebaõnnestumise modelleerimine

Otsida tabelist võtmele vastav väärtus

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

Algebralised andmetüübid

Rekursiivsed andmetüübid

```
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
```

Algebraised andmetüübid

Kirjed

Konstruktori argumendid võivad olla “märgendatud”

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

- *val*, *left* ja *right* on selektorfunktsioonid

val :: *Tree a* → *a*

left :: *Tree a* → *Tree a*

right :: *Tree a* → *Tree a*

- Väärtusi võib defineerida tavaliselt (identifitseerides argumendi positsiooni kaudu) või kasutades märgendeid

tree1 = *Branch* 1 *Empty Empty*

tree2 = *Branch*{ *left* = *Empty*
 , *val* = 1
 , *right* = *Empty* }

Uute tüüpide defineerimine

"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"

Näide: 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)
```

Näide: 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*

Näide: 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)
```

Näide: 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
```

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste esitamine

```
type Name = String
data Expr = Num Double
          | Var Name
          | Expr : + : Expr
          | Expr : * : Expr
```

Näited

```
expr1 = Num 3 : + : (Num 2 : * : Num 7)
expr2 = (Var "x" : + : Num 2) : * : Num 4
```

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste väärtustamine

$eval :: Expr \rightarrow Double$

$eval (Num\ n) = n$

$eval (e1 : + : e2) = eval\ e1 + eval\ e2$

$eval (e1 : * : e2) = eval\ e1 * eval\ e2$

NB!

Muutujate korral ei ole defineeritud!

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste väärtustamine

```
evalE :: Expr → [(Name, Double)] → Double  
evalE (Num n) env      = n  
evalE (e1 : + : e2) env = evalE e1 env + evalE e2 env  
evalE (e1 : * : e2) env = evalE e1 env * evalE e2 env  
evalE (Var x) env      = case lookup x env of  
                                Just v      → v  
                                Nothing → error (msg x)  
where msg x = "Unbound variable: " ++ x
```

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste näitamine

```
showExp :: Expr → String  
showExp (Num n)      = show n  
showExp (Var v)      = v  
showExp (e1 : + : e2) = showExp e1 ++ "+" ++  
                           showExp e2  
showExp (e1 : * : e2) = showExp e1 ++ "*" ++  
                           showExp e2
```

NB!

Sulud puudu!

```
Main > showExp expr2  
"x+2.0*4.0"
```

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste näitamine

```
showExp :: Expr → String
showExp (Num n)      = show n
showExp (Var v)      = v
showExp (e1 : + : e2) = "(" ++ showExp e1 ++ "+" ++
                             showExp e2 ++ ")"
showExp (e1 : * : e2) = "(" ++ showExp e1 ++ "*" ++
                             showExp e2 ++ ")"
```

NB!

Liiga palju sulge!

```
Hugs> showExp expr1
"(3.0+(2.0*7.0))"
```

Näide: aritmeetilised avaldised

Aritmeetiliste avaldiste näitamine

```
showExp :: Expr → String  
showExp (Num n)      = show n  
showExp (Var v)      = v  
showExp (e1 : + : e2) = showExp e1 ++ "+" ++  
                           showExp e2  
showExp (e1 : * : e2) = showArg e1 ++ "*" ++  
                           showArg e2  
showArg (e1 : + : e2) = "(" ++  
                           showExp (e1 : + : e2) ++  
                           ")"  
showArg e              = showExp e
```