TYYPPILUOKAT

1. Taustaa

- Ohjelmointikielissä ilmenee kahdentyyppistä polymorfismia: ad-hoc ja parametristä.

- Ad-hoc polymorfismia on se, kun funktio on määritelty usealle eri tyypille, toimien tyypistä riippuen eri tavoin.

  Esim. kertolaskuoperaattorin ylikirjoitus int ja float tyypeille:

  ```
  3 * 3
  3.14 * 3.14
  ```

- Parametrista polymorfismia on se, kun funktio on määritelty jonkin tyyppialueen yli, toimien samalla tavalla kaikille tyypeille.

  Esim. lenght -funktio, joka kertoo listan alkioiden määrän riippumatta mitä tyyppiä alkio on.

- Parametriselle polymorfismille yleisesti hyväksytty toteutustapa on Hindley/Milner.

- Ad-hoc polymorfismille ei yleisesti hyväksyttyä tapaa ollut - ennen tyyppiluokkia.

- Ad-hoc polymorfismia tarvitaan mm. aritmeettisten sekä yhtäläisyysoperaattoreiden kanssa.

  Täysin polymorfisina:

  Aritmetiikka:

  Aritmeettiset operaattorit määritelty vain numeerisille tyypeille.

  Yhtäläisyys:

  Operaattorin käyttö funktiotyypeille tai abstrakteille tyypeille ei aiheuta tyyppivirhettä.

  Funktiotyypeille, abstrakteille luokille:

  Vertaa esityksiä, ei arvoja.

Ilman polymorfismia:

```
square x = x * x    -- ei sallittu
```

Aritmetiikka:

```
squareInt x   = x *int x
squareFloat x = x *float x
```

Yhtäläisyys:

```
x ==int y
x ==float y
```

2. Mitä tyyppiluokat ovat

- Tyyppiluokka on ryhmä tyyppejä, joita yhdistää se, että niille on määritelty tietyt funktiot
- Yleisimmät käyttökohteet: aritmetiikka ja yhtäläisyys

- Yleisesti hyväksytty ad-hoc polymorfismin toteutustapa
- Yhdistää parametrisen ja ad-hoc polymorfismin
- Peräisin Haskellista
- Hindley/Milner tyyppijärjestelmän yleistys
    - Tyypit voidaan päätellä kuten Hindley/Milnerissä
- Ohjelmat voidaan muuttaa käyttämään ainoastaan H/M:ää
    - Ylikirjoitusten poistoon translaatiosäännöt
    - Helppoa toteuttaa H/M:ää käyttävissä kielissä: preprosessoimalla

3. Tyyppiluokkien esimerkkejä

Tyyppiluokan määrittely:

```
class  Num a  where
  (+), (*)  ::  a -> a -> a
  negate    ::  a -> a
```

"Tyyppi a voi kuulua tyyppiluokkaan Num, jos sille on määritelty oikean tyyppiset (+), (*) ja negate -funktiot."

Tyypin lisääminen tyyppiluokkaan.

```
instance  Num Int  where
  (+)     =  addInt
  (*)     =  mulInt
  negate  =  negInt
```

"Int on tyypin Num instanssi. Funktioina (+), (*) ja negate käytetään funktioita addInt, mulInt ja negInt vastaavasti."

Tyypin lisääminen tyyppiluokkaan.

```
instance  Num Float  where
  (+)     =  addFloat
  (*)     =  mulFloat
  negate  =  negFloat
```

"Float on tyypin Num instanssi. Funktioina (+), (*) ja negate käytetään funktioita addFloat, mulFloat ja negFloat vastaavasti."

Funktion tyyppi

```
square x  = x * x
square   :: Num a => a -> a
```

"Square on tyyppiä 'a -> a' kaikille tyypeille, jotka kuuluvat tyyppiluokkaan Num."

joten

```
square 3     -- ok
square 3.14  -- ok
square 'x'   -- ei onnistu, Char ei ole Num –tyypin instanssi
```

- Tyypinpäättely tarkistaa, että instantoinnissa annetut funktiot ovat oikeaa tyyppiä.

Tyyppiluokan funktioiden default määritelmiä:

```
class  Eq a  where
  (==), (/=) :: a -> a -> bool

  x /= y      = not (x == y)
  x == y      = not (x /= y)
```

- Instanssi käyttää default määritelmää, ellei se ylikirjoita sitä
- Tyypin tulee toteuttaa ainakin toinen funktioista (==), (/=) voidakseen olla tyyppiluokan Eq instanssi

Eq:n instansseja:

```
instance  Eq Int  where
  (==)  = eqInt

instance  Eq Float  where
  (/=)  = neqFloat

instance  Eq a, Eq b => Eq (a, b)  where
  (u, v) == (x, y)   = (u == x) & (v == y)
```

Listalle:

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


member           :: Eq a => [a] -> a -> bool
member [] y       = False
member (x:xs) y  = (x == y) \/ member xs y
```

- Member funktio saa tyyppirajoitteensa käyttämästään '==' operaattorista

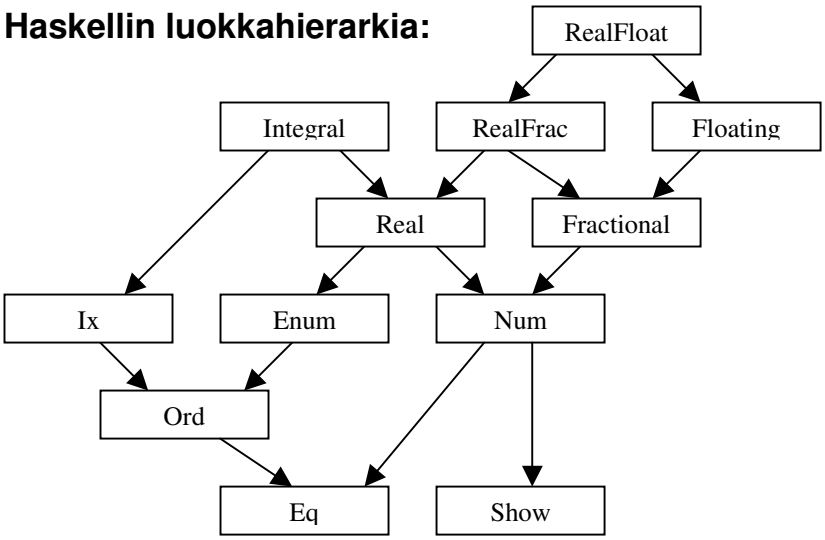
Joukolle:

```
data  Set a  = MkSet [a]

instance Eq a => Eq (Set a)  where
   MkSet xs == MkSet ys  =  and (map (member xs) ys)
                          & and (map (member ys) xs)
```

4. Aliluokat, Käyttäjän määrittelemät tietotyypit

Haskellin luokkahierarkia:



Esimerkkejä periytyvistä luokista:

Ord -tyyppiluokka periytyy Eq:sta:

```
class  Eq a => Ord a  where
  (<), (<=), (>), (>=)  :: a -> a -> bool
  max, min              :: a -> a -> a
  compare               :: a -> a -> Ordering
```

Show -tyyppiluokka:

```
class Show a where
  show      :: a -> String
```

Num -tyyppiluokka periytyy Eq:sta ja Show:sta:

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

  x - y           = x + negate y
  fromInt         = fromIntegral
```

Useampia tyyppiparametreja:

```
class  Coerce a b  where
  coerce :: a -> b

instance  Coerce Int Float  where
  coerce = convertIntToFloat
```

Käyttäjän määrittelemä tietotyyppi:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
           deriving (Eq, Ord, Text)
```

- Kääntäjä generoi ja ylikirjoittaa tarvittavat operaattorit
- Sallittu vain rajoitetusti tietyille tyyppiluokille: Eq, Ord, Text, Ix, Enum, Binary

5. Translaatio

- Translaatiolla voidaan muuttaa ohjelma, joka sisältää tyyppiluokka ja -instanssi määrittelyjä ohjelmaksi, joka ei sisällä niitä.
- Muutettu ohjelma on Hindley/Milner tyyppijärjestelmän mukainen.

Tyyppiluokkia käyttävä ohjelma:

```
class  Num a  where
  (+), (*)  :: a -> a -> a
  negate    :: a -> a

instance  Num Int  where
  (+)         = addInt
  (*)         = mulInt
  negate      = negInt

instance  Num Float  where
  (+)         = addFloat
  (*)         = mulFloat
  negate      = negFloat

square       :: Num a => a -> a
square x      = x * x
```

Muutettu versio, joka ei käytä tyyppiluokkia:

```
-- class Num
data  NumD a  = NumDict (a -> a -> a) (a -> a -> a) (a -> a)

add (NumDict a m n)  = a
mul (NumDict a m n)  = m
neg (NumDict a m n)  = n

-- instance Num Int
numDInt     :: NumD Int
numDInt      = NumDict addInt mulInt negInt

-- instance Num Float
numDFloat   :: NumD Float
numDFloat    = NumDict addFloat mulFloat negFloat

-- square
square'          :: NumD a -> a -> a
square' numDa x   = mul numDa x x
```

6. Yhteys olio-ohjelmointiin

Yhteys olio-ohjelmointiin:

Haskell:

```
show :: Show a => a -> String
show :: Bool -> String
show :: Char -> String
```

C++:

```
class ShowType {
  public:
    virtual String show(ShowType);
}

class Bool : ShowType { ... }
class Char : ShowType { ... }
```

Java:

```
public interface ShowType {
    public String show(ShowType);
}

public class Bool implements ShowType { ... }
public class Char implements ShowType { ... }
```

Staattinen vs. dynaaminen sidonta

Haskell: Staattinen. Vain homogeenisiä listoja.
C++, Java: Dynaaminen.

Esim.

```
[True, 'N', False] :: [ShowType]
concat . map show  :: [ShowType] -> String
```

Haskell: Ei.
C++, Java: Kyllä.

Post-hoc laajennukset tyypeille

Esim.

```
class  Num a  where ...
instance  Num Int  where
(+)     =  addInt ...
```

Haskell: Kyllä.
C++, Java: Ei.

7. Esimerkkejä tyyppiluokkien laajennuksia

Usemman parametrin tyyppiluokat

Esim. rajoiteparametrit

```
class  Collection c a  where
...

instance  Eq a => Collection ListSet a  where
...

instance  Ord a => Collection TreeSet a  where
...
```

- Collection tyyppiluokan rajoite instanssikohtaisesti

Limittyvät (overlap) instanssit

Esim.

```
class  MyShow a  where
  myShow :: a -> String

instance  MyShow a => MyShow [a] where
  myShow = myShow1

instance  MyShow [Char]  where
  myShow = myShow2
```

- [Char] tyyppi sopisi molempiin instansseihin
- Tarkoituksena: [Char] tyypille myShow2, muille myShow1
- Löytyy Goferista, Haskellista ei

Konstruktoriluokat

- Yhdistää tyyppiluokat ja korkeamman asteen Hindley/Milnerin
- Ilmaisuvoimainen