Πολυμορφισμός και Υπερφόρτωση

Γιάννης Κασσιός

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

Δομή Σημερινής Διάλεξης

- Πολυμορφισμός (polymorphism)
  - ορισμός / χρησιμότητα
  - μεταβλητές τύπων
  - πολυμορφικοί τύποι Haskell
  - γενικότητα
- Υπερφόρτωση (overloading)
  - ορισμός / χρησιμότητα
  - κλάσεις τύπων
  - δήλωση / χρήση κλάσεων
  - προκαθορισμένες τιμές / κληρονομικότητα
  - κλάσεις Haskell
  - συσχέτιση με αντικειμενοστρεφή προγραμματισμό

- Πολυμορφική τιμή: μία τιμή με πολλούς τύπους
  - `[]::String` αλλά και `[]::[Int]`

    `"Hi"++[]` `[1,2,3]++[]`
  - `length::[Int]->Int` αλλά και

    `length::[(Char,[Int])]->Int`
- Χρησιμότητα: μας γλιτώνει από άπειρους εντελώς όμοιους ορισμούς

- Πολυμορφική τιμή: μία τιμή με πολλούς τύπους
  - `[]::String` αλλά και `[]::[Int]`

    `"Hi"++[]  [1,2,3]++[]`
  - `length::[Int]->Int` αλλά και

    `length::[(Char,[Int])]->Int`
- Χρησιμότητα: μας γλιτώνει από άπειρους εντελώς όμοιους ορισμούς

```
emptyBool :: [Bool]
lengthBool :: [Bool]->Int
lengthBool emptyBool = 0
lengthBool (_:xs) = 1+lengthBool xs
```

- Πολυμορφική τιμή: μία τιμή με πολλούς τύπους
  - `[]::String` αλλά και `[]::[Int]`

    `"Hi"++[]  [1,2,3]++[]`
  - `length::[Int]->Int` αλλά και

    `length::[(Char,[Int])]->Int`
- Χρησιμότητα: μας γλιτώνει από άπειρους εντελώς όμοιους ορισμούς

```
emptyInt :: [Int]
lengthInt :: [Int]->Int
lengthInt emptyInt = 0
lengthInt (_:xs) = 1+lengthInt xs
```

- Πολυμορφική τιμή: μία τιμή με πολλούς τύπους
  - `[]::String` αλλά και `[]::[Int]`

    `"Hi"++[]` `[1,2,3]++[]`
  - `length::[Int]->Int` αλλά και

    `length::[(Char,[Int])]->Int`
- Χρησιμότητα: μας γλιτώνει από άπειρους εντελώς όμοιους ορισμούς

```
length [] = 0
length (_:xs) = 1 + length xs
```

Πολυμορφισμός
Μεταβλητές Τύπων - I

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[]::[a]
length :: [a]->Int
```

Πολυμορφισμός
Μεταβλητές Τύπων - I

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[]::[a]
length :: [a]->Int
```

Πολυμορφισμός
Μεταβλητές Τύπων - I

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[]::[a]
length :: [a]->Int
```

Πολυμορφισμός
Μεταβλητές Τύπων - I

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[]::[a]
length :: [a]->Int
```

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[] :: [a]
length :: [a]->Int
```

- Τύποι της `length`: όλοι οι τύποι της μορφής `[a]->Int`
- Τύποι της `[]`: όλοι οι τύποι της μορφής `[a]`
- Μεταβλητή τύπου (type variable) `a`: αντικαθίσταται από τύπο
  - πολυμορφικοί τύποι Haskell: χρησιμοποιούν μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από σταθερούς τύπους
  - συνήθως: `a,b,c,...`
- Καθορισμοί τύπου:

```
[]::[a]

length ::  [a]->Int
```

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

Πολυμορφισμός
Μεταβλητές Τύπων - II

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

Πολυμορφισμός
Μεταβλητές Τύπων - II

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

Πολυμορφισμός
Μεταβλητές Τύπων - II

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

Πολυμορφισμός
Μεταβλητές Τύπων - II

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

- Πολυμορφικός Τύπος `a`: περιγράφει όλους τους τύπους
  - τύπος της `undef`
- Κάθε μεταβλητή αντικαθίσταται από ένα μόνο τύπο

  `(++) :: a->a->a`

  ο τύπος περιγράφει:

  `[Int]->[Int]->[Int]`

  `[Bool]->[Bool]->[Bool]`

  αλλά όχι

  `[Int]->[Char]->[Bool]`
- Δύο διαφορετικές μεταβλητές: δεν υπάρχει περιορισμός

  `zip::[a]->[b]->[(a,b)]`

  `zip::[Int]->[Char]->[(Int,Char)]`

  αλλά και

  `zip::[Int]->[Int]->[(Int,Int)]`

```
dovetail3 = dovetail3aux 0
    where
    dovetail3aux n l1 l2 l3 =
        [ ((l1!!i1), (l2!!i2), (l3!!i3))
        | i1<-[0..n], i2<-[0..n], i3<-[0..n],
          i1+i2+i3 == n ]
     ++ dovetail3aux (n+1) l1 l2 l3
dovetail3 :: [a]->[b]->[c]->[(a,b,c)]
```

```
dovetail3 = dovetail3aux 0
   where
   dovetail3aux n l1 l2 l3 =
       [ ((l1!!i1), (l2!!i2), (l3!!i3))
       | i1<-[0..n], i2<-[0..n], i3<-[0..n],
         i1+i2+i3 == n ]
    ++ dovetail3aux (n+1) l1 l2 l3
dovetail3 :: [a]->[b]->[c]->[(a,b,c)]
```

Πολυμορφισμός

Πολυμορφικοί Τύποι Τιμών Haskell

```
id::a->a
fst::(a,b)->a
snd::(a,b)->b
[]::[a]
(:)::a->[a]->[a]
(++)::[a]->[a]->[a]
concat::[[a]]->[a]
length::[a]->Int
head::[a]->a
tail::[a]->[a]
last::[a]->a
init::[a]->[a]
replicate::Int->a->[a]
```

```
take :: Int->[a]->[a]
drop :: Int->[a]->[a]
splitAt :: Int->[a]->([a],[a])
reverse :: [a]->[a]
zip :: [a]->[b]->[(a,b)]
unzip :: [(a,b)]->([a],[b])
map :: (a->b)->[a]->[b]
filter :: (a->Bool)->[a]->[a]
zipWith::(a->b->c)->[a]->[b]->[c]
foldl::(a->b->a)->a->[b]->a
foldr::(a->b->b)->b->[a]->b
takeWhile::(a->Bool)->[a]->[a]
dropWhile::(a->Bool)->[a]->[a]
```

Πολυμορφισμός
Πολυμορφικοί Τύποι Τιμών Haskell

```
id::a->a
fst::(a,b)->a
snd::(a,b)->b
[]::[a]
(:)::a->[a]->[a]
(++)::[a]->[a]->[a]
concat::[[a]]->[a]
length::[a]->Int
head::[a]->a
tail::[a]->[a]
last::[a]->a
init::[a]->[a]
replicate::Int->a->[a]
```

```
take :: Int->[a]->[a]
drop :: Int->[a]->[a]
splitAt :: Int->[a]->([a],[a])
reverse :: [a]->[a]
zip :: [a]->[b]->[(a,b)]
unzip :: [(a,b)]->([a],[b])
map :: (a->b)->[a]->[b]
filter :: (a->Bool)->[a]->[a]
zipWith::(a->b->c)->[a]->[b]->[c]
foldl::(a->b->a)->a->[b]->a
foldr::(a->b->b)->b->[a]->b
takeWhile::(a->Bool)->[a]->[a]
dropWhile::(a->Bool)->[a]->[a]
```

Πολυμορφισμός
Πολυμορφικό folding

- Τις συναρτήσεις `foldl,foldr` τις συνηθίσαμε να συσσωρεύουν τιμές ίδιου τύπου με αυτές της λίστας:

```
sum :: [Int]->Int
sum = foldl (+) 1
```

- Οι συναρτήσεις είναι πιο γενικές
  - `foldl::(a->b->a)->a->[b]->a`
  - συσσώρευση αποτελέσματος τύπου `a` σε λίστα τύπου `[b]`
- Γραμμική αναζήτηση χωρίς `map`. Συνάρτηση συσσώρευσης:

```
\found-> \current-> found || current == x
```

- Νέα συνάρτηση γραμμικής αναζήτησης:

```
foldl (\found-> \current-> found || current == x)
   False
```

Πολυμορφισμός
Πολυμορφικό folding

- Τις συναρτήσεις `foldl`,`foldr` τις συνηθίσαμε να συσσωρεύουν τιμές ίδιου τύπου με αυτές της λίστας:

```
sum :: [Int]->Int
sum = foldl (+) 1
```

- Οι συναρτήσεις είναι πιο γενικές
  - `foldl::(a->b->a)->a->[b]->a`
  - συσσώρευση αποτελέσματος τύπου `a` σε λίστα τύπου `[b]`
- Γραμμική αναζήτηση χωρίς `map`. Συνάρτηση συσσώρευσης:

```
\found-> \current-> found || current == x
```

- Νέα συνάρτηση γραμμικής αναζήτησης:

```
foldl (\found-> \current-> found || current == x)
   False
```

Πολυμορφισμός
Πολυμορφικό folding

- Τις συναρτήσεις `foldl,foldr` τις συνηθίσαμε να συσσωρεύουν τιμές ίδιου τύπου με αυτές της λίστας:

```
sum :: [Int]->Int
sum = foldl (+) 1
```

- Οι συναρτήσεις είναι πιο γενικές
  - `foldl::(a->b->a)->a->[b]->a`
  - συσσώρευση αποτελέσματος τύπου `a` σε λίστα τύπου `[b]`
- Γραμμική αναζήτηση χωρίς `map`. Συνάρτηση συσσώρευσης:

```
\found-> \current-> found || current == x
```

- Νέα συνάρτηση γραμμικής αναζήτησης:

```
foldl (\found-> \current-> found || current == x)
   False
```

Πολυμορφισμός
Πολυμορφικό folding

- Τις συναρτήσεις `foldl`,`foldr` τις συνηθίσαμε να συσσωρεύουν τιμές ίδιου τύπου με αυτές της λίστας:

```
sum :: [Int]->Int
sum = foldl (+) 1
```

- Οι συναρτήσεις είναι πιο γενικές
  - `foldl::(a->b->a)->a->[b]->a`
  - συσσώρευση αποτελέσματος τύπου `a` σε λίστα τύπου `[b]`
- Γραμμική αναζήτηση χωρίς `map`. Συνάρτηση συσσώρευσης:

```
\found-> \current-> found || current == x
```

- Νέα συνάρτηση γραμμικής αναζήτησης:

```
foldl (\found-> \current-> found || current == x)
   False
```

- Τις συναρτήσεις `foldl`,`foldr` τις συνηθίσαμε να συσσωρεύουν τιμές ίδιου τύπου με αυτές της λίστας:

```
sum :: [Int]->Int
sum = foldl (+) 1
```

- Οι συναρτήσεις είναι πιο γενικές
  - `foldl::(a->b->a)->a->[b]->a`
  - συσσώρευση αποτελέσματος τύπου `a` σε λίστα τύπου `[b]`
- Γραμμική αναζήτηση χωρίς `map`. Συνάρτηση συσσώρευσης:

```
\found-> \current-> found || current == x
```

- Νέα συνάρτηση γραμμικής αναζήτησης:

```
foldl (\found-> \current-> found || current == x)
   False
```

- Αντικατάσταση (substitution): πεπερασμένη συσχέτιση μεταξύ ονομάτων και πολυμορφικών τύπων
- Αν σ είναι αντικατάσταση και T πολυμορφικός τύπος, ο τύπος σT είναι αυτός που θα πάρουμε αν κάνουμε όλες τις αντικαταστάσεις ονομάτων της σ στον T
- Παράδειγμα: αν

$$\sigma \quad = \quad \texttt{a} \mapsto (\texttt{c->Bool}) \mid \texttt{b} \mapsto \texttt{Int}$$

τότε

$$\sigma(\texttt{[(a,b,c)]}) = \texttt{[(c->Bool , Int, c)]}$$

- Τύπος A πιο γενικός (more general) από B αν υπάρχει σ ώστε:

$$B = \sigma A$$

- Γενικότερος όλων: `a`

- Αντικατάσταση (substitution): πεπερασμένη συσχέτιση μεταξύ ονομάτων και πολυμορφικών τύπων
- Αν σ είναι αντικατάσταση και T πολυμορφικός τύπος, ο τύπος σT είναι αυτός που θα πάρουμε αν κάνουμε όλες τις αντικαταστάσεις ονομάτων της σ στον T
- Παράδειγμα: αν

$$\sigma = \texttt{a} \mapsto \texttt{(c->Bool)} \mid \texttt{b} \mapsto \texttt{Int}$$

τότε

$$\sigma(\texttt{[(a,b,c)]}) = \texttt{[(c->Bool , Int, c)]}$$

- Τύπος A πιο γενικός (more general) από B αν υπάρχει σ ώστε:

$$B = \sigma A$$

- Γενικότερος όλων: `a`

- Αντικατάσταση (substitution): πεπερασμένη συσχέτιση μεταξύ ονομάτων και πολυμορφικών τύπων
- Αν σ είναι αντικατάσταση και T πολυμορφικός τύπος, ο τύπος σT είναι αυτός που θα πάρουμε αν κάνουμε όλες τις αντικαταστάσεις ονομάτων της σ στον T
- Παράδειγμα: αν

$$\sigma \quad = \quad \texttt{a} \mapsto (\texttt{c->Bool}) \mid \texttt{b} \mapsto \texttt{Int}$$

τότε

$$\sigma(\texttt{[(a,b,c)]}) = \texttt{[(c->Bool , Int, c)]}$$

- Τύπος A πιο γενικός (more general) από B αν υπάρχει σ ώστε:

$$B = \sigma A$$

- Γενικότερος όλων: a

- Αντικατάσταση (substitution): πεπερασμένη συσχέτιση μεταξύ ονομάτων και πολυμορφικών τύπων
- Αν σ είναι αντικατάσταση και T πολυμορφικός τύπος, ο τύπος σT είναι αυτός που θα πάρουμε αν κάνουμε όλες τις αντικαταστάσεις ονομάτων της σ στον T
- Παράδειγμα: αν

$$\sigma \quad = \quad \texttt{a} \mapsto (\texttt{c->Bool}) \mid \texttt{b} \mapsto \texttt{Int}$$

τότε

$$\sigma(\texttt{[(a,b,c)]}) = \texttt{[(c->Bool , Int, c)]}$$

- Τύπος A πιο γενικός (more general) από B αν υπάρχει σ ώστε:

$$B = \sigma A$$

- Γενικότερος όλων: a

- Αντικατάσταση (substitution): πεπερασμένη συσχέτιση μεταξύ ονομάτων και πολυμορφικών τύπων
- Αν σ είναι αντικατάσταση και T πολυμορφικός τύπος, ο τύπος σT είναι αυτός που θα πάρουμε αν κάνουμε όλες τις αντικαταστάσεις ονομάτων της σ στον T
- Παράδειγμα: αν

$$\sigma \quad = \quad \texttt{a} \mapsto (\texttt{c->Bool}) \mid \texttt{b} \mapsto \texttt{Int}$$

τότε

$$\sigma(\texttt{[(a,b,c)]}) = \texttt{[(c->Bool , Int, c)]}$$

- Τύπος A πιο γενικός (more general) από B αν υπάρχει σ ώστε:

$$B = \sigma A$$

- Γενικότερος όλων: `a`

- Γενικότερος τύπος ⇒ περισσότερη ευελιξία
  - πχ. `[]` έχει τύπο `[a]` και όχι `[a->b]`
- Αναζητάμε πάντα το γενικότερο κατάλληλο τύπο για τις τιμές μας
- Εξαγωγή τύπου (type inference): ελλείψει καθορισμού τύπου η Haskell βρίσκει τον πιο γενικό πολυμορφικό τύπο
  - ομοίως με λ-εκφράσεις
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της:
  `:type dovetail3` δίνει `[a] -> [b] -> [c] -> [(a,b,c)]`
  `:type \x->[x]` δίνει `a -> [a]`

- Γενικότερος τύπος $\Rightarrow$ περισσότερη ευελιξία
  - πχ. `[]` έχει τύπο `[a]` και όχι `[a->b]`
- Αναζητάμε πάντα το γενικότερο κατάλληλο τύπο για τις τιμές μας
- Εξαγωγή τύπου (type inference): ελλείψει καθορισμού τύπου η Haskell βρίσκει τον πιο γενικό πολυμορφικό τύπο
  - ομοίως με λ-εκφράσεις
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της:
  `:type dovetail3` δίνει `[a] -> [b] -> [c] -> [(a,b,c)]`
  `:type \x->[x]` δίνει `a -> [a]`

- Γενικότερος τύπος ⇒ περισσότερη ευελιξία
  - πχ. `[]` έχει τύπο `[a]` και όχι `[a->b]`
- Αναζητάμε πάντα το γενικότερο κατάλληλο τύπο για τις τιμές μας
- Εξαγωγή τύπου (type inference): ελλείψει καθορισμού τύπου η Haskell βρίσκει τον πιο γενικό πολυμορφικό τύπο
  - ομοίως με λ-εκφράσεις
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της:
  `:type dovetail3` δίνει `[a] -> [b] -> [c] -> [(a,b,c)]`
  `:type \x->[x]` δίνει `a -> [a]`

- Γενικότερος τύπος $\Rightarrow$ περισσότερη ευελιξία
  - πχ. `[]` έχει τύπο `[a]` και όχι `[a->b]`
- Αναζητάμε πάντα το γενικότερο κατάλληλο τύπο για τις τιμές μας
- Εξαγωγή τύπου (type inference): ελλείψει καθορισμού τύπου η Haskell βρίσκει τον πιο γενικό πολυμορφικό τύπο
  - ομοίως με λ-εκφράσεις
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της:
  `:type dovetail3` δίνει `[a] -> [b] -> [c] -> [(a,b,c)]`
  `:type \x->[x]` δίνει `a -> [a]`

Υπερφόρτωση
Υπερφορτωμένες τιμές

- Υπερφορτωμένο όνομα:
  - περισσότεροι από ένας τύποι
  - διαφορετικός ορισμός για κάθε τύπο
- Παραδείγματα: `(==)`,`(+)` κτλ.

  Για `(==)::Bool->Bool->Bool`

  ```
  True == True = True
  False == False = True
  _ == _ = False
  ```

  Για `(==)::Int->Int->Bool`: built-in

  Για `(==)::(a,b)->(a,b)->Bool`:

  ```
  (x,y) == (z,w) = x==z && y==w
  ```

Υπερφόρτωση
Υπερφορτωμένες τιμές

- Υπερφορτωμένο όνομα:
  - περισσότεροι από ένας τύποι
  - διαφορετικός ορισμός για κάθε τύπο
- Παραδείγματα: `(==)`, `(+)` κτλ.

  Για `(==)::Bool->Bool->Bool`

  ```
  True == True = True
  False == False = True
  _ == _ = False
  ```

  Για `(==)::Int->Int->Bool`: built-in

  Για `(==)::(a,b)->(a,b)->Bool`:

  ```
  (x,y) == (z,w) = x==z && y==w
  ```

- Υπερφορτωμένο όνομα:
  - περισσότεροι από ένας τύποι
  - διαφορετικός ορισμός για κάθε τύπο
- Παραδείγματα: `(==)`, `(+)` κτλ.

  Για `(==)::Bool->Bool->Bool`

```
True == True = True
False == False = True
_ == _ = False
```

  Για `(==)::Int->Int->Bool`: built-in

  Για `(==)::(a,b)->(a,b)->Bool`:

```
(x,y) == (z,w) = x==z && y==w
```

Υπερφόρτωση
Σημασία Υπερφόρτωσης

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για a με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

Υπερφόρτωση
Σημασία Υπερφόρτωσης

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για a με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

Υπερφόρτωση
Σημασία Υπερφόρτωσης

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για `a` με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για `a` με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για `a` με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

- Μας γλυτώνει από τη χρήση διαφορετικών συμβόλων για την ίδια πράξη
- Έστω:

```
lSearch x [] = False
lSearch x (y:ys) = x==y || lSearch x ys
```

- Τύπος: `lSearch :: a->[a]->Bool`

  μόνο για `a` με: `(==)::a->a->Bool`
- Ο περιορισμός αυτός δεν εκφράζεται πολυμορφικά
- Μία πολυμορφική λύση:

```
lSearch :: (a->a->Bool)->a->[a]->Bool
lSearch eq x [] = False
lSearch eq x (y:ys) = eq x y || lSearch eq x ys
```

  - γενικεύει και περιπλέκει χωρίς λόγο

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ.
  `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής:
  συμφραζόμενα=>πολυμορφικός_τύπος
- `lSearch :: Eq a => a->[a]->Bool`

Υπερφόρτωση
Κλάσεις - I

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ.
  `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής:
  συμφραζόμενα=>πολυμορφικός_τύπος
- `lSearch :: Eq a => a->[a]->Bool`

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ. `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής: *συμφραζόμενα=>πολυμορφικός_τύπος*
- `lSearch :: Eq a => a->[a]->Bool`

Υπερφόρτωση
Κλάσεις - I

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ.
  `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής:
  συμφραζόμενα=>πολυμορφικός_τύπος
- `lSearch :: Eq a => a->[a]->Bool`

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ.
  `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής:
  συμφραζόμενα=>πολυμορφικός_τύπος
- `lSearch :: Eq a => a->[a]->Bool`

- Κλάση (class): σύνολο τύπων για τους οποίους υπάρχουν συγκεκριμένα υπερφορτωμένα ονόματα
- Παράδειγμα: κλάση `Eq`: σύνολο τύπων `a` για τους οποίους ορίζεται
  `(==) :: a->a->Bool`
- Τύπος `a` ανήκει σε κλάση `C` γράφεται:
  `C a` περιορισμός (constraint)
- Πολλοί περιορισμοί = συμφραζόμενα (context) πχ.
  `(Eq a,Eq b)`
- Πολυμορφικοί τύποι δέχονται συμφραζόμενα ως εξής:
  συμφραζόμενα=>πολυμορφικός_τύπος
- `lSearch :: Eq a => a->[a]->Bool`

Υπερφόρτωση
Κλάσεις - II

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

Υπερφόρτωση
Κλάσεις - II

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

Υπερφόρτωση
Κλάσεις - II

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

Υπερφόρτωση
Κλάσεις - II

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

Υπερφόρτωση
Κλάσεις - II

- Επιτρέπονται:

```
lSearch True [False,True]
lSearch ('a',50)
    [(c,i) | c<-['a','z'], i<-[0..100]]
```

- Απαγορεύεται: `lSearch (1+) [\x->1+x]`

```
ERROR - Cannot infer instance
*** Instance : Eq (a -> a)
*** Expression : lsearch ((+) 1) [\x -> x + 1]
```

- Η εξαγωγή τύπου ανακαλύπτει και τα συμφραζόμενα:
  - σβήνουμε τον καθορισμό τύπου της `lSearch`
  - γράφουμε `:type lSearch` στο διερμηνέα
  - απάντηση: `lSearch :: Eq a => a -> [a] -> Bool`

- Σύνταξη δήλωσης κλάσης:
  `class` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
- Υπογραφή (signature) της κλάσης
- Παράδειγμα: συνάρτηση `cond`
  - δουλεύει σαν την `if`
  - δέχεται ακεραίους (C/C++) και λίστες (scripting lang.) ως συνθήκες
- `class Condition a where cond :: a->b->b->b`

- Σύνταξη δήλωσης κλάσης:
  `class` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
- Υπογραφή (signature) της κλάσης
- Παράδειγμα: συνάρτηση `cond`
  - δουλεύει σαν την `if`
  - δέχεται ακεραίους (C/C++) και λίστες (scripting lang.) ως συνθήκες
- `class Condition a where cond :: a->b->b->b`

- Σύνταξη δήλωσης κλάσης:
  `class` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
- Υπογραφή (signature) της κλάσης
- Παράδειγμα: συνάρτηση `cond`
  - δουλεύει σαν την `if`
  - δέχεται ακεραίους (C/C++) και λίστες (scripting lang.) ως συνθήκες
- `class Condition a where cond :: a->b->b->b`

- Σύνταξη δήλωσης κλάσης:
  `class` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
- Υπογραφή (signature) της κλάσης
- Παράδειγμα: συνάρτηση `cond`
  - δουλεύει σαν την `if`
  - δέχεται ακεραίους (C/C++) και λίστες (scripting lang.) ως συνθήκες
- `class Condition a where cond :: a->b->b->b`

Υπερφόρτωση
Δήλωση Στιγμιότυπου - I

- Στιγμιότυπο (instance) κλάσης: ένας τύπος που ανήκει στην κλάση
- Σύνταξη δήλωσης στιγμιοτύπου:
  `instance` *όνομα_κλάσης τύπος* `where` *δηλώσεις_τιμών*
- Παράδειγμα:

```
instance Condition Bool where
   cond c t e = if c then t else e
instance Condition Int where
   cond c t e = if c/=0 then t else e
instance Condition[a] where
   cond c t e = if c/="" then t else e
```

- Χρήση:

```
cond (length"") 1 2 = 2    cond "a" 1 2 = 1
```

  όχι: `cond 1 1 2`

Υπερφόρτωση
Δήλωση Στιγμιότυπου - I

- Στιγμιότυπο (instance) κλάσης: ένας τύπος που ανήκει στην κλάση
- Σύνταξη δήλωσης στιγμιοτύπου:
  `instance` *όνομα_κλάσης τύπος* `where` *δηλώσεις_τιμών*
- Παράδειγμα:

```
instance Condition Bool where
   cond c t e = if c then t else e
instance Condition Int where
   cond c t e = if c/=0 then t else e
instance Condition[a] where
   cond c t e = if c/="" then t else e
```

- Χρήση:

```
cond (length"") 1 2 = 2    cond "a" 1 2 = 1
```

όχι: `cond 1 1 2`

Υπερφόρτωση
Δήλωση Στιγμιότυπου - I

- Στιγμιότυπο (instance) κλάσης: ένας τύπος που ανήκει στην κλάση
- Σύνταξη δήλωσης στιγμιοτύπου:
  `instance` *όνομα_κλάσης τύπος* `where` *δηλώσεις_τιμών*
- Παράδειγμα:

```
instance Condition Bool where
   cond c t e = if c then t else e
instance Condition Int where
   cond c t e = if c/=0 then t else e
instance Condition[a] where
   cond c t e = if c/="" then t else e
```

- Χρήση:

```
cond (length"") 1 2 = 2    cond "a" 1 2 = 1
```

όχι: `cond 1 1 2`

Υπερφόρτωση
Δήλωση Στιγμιότυπου - I

- Στιγμιότυπο (instance) κλάσης: ένας τύπος που ανήκει στην κλάση
- Σύνταξη δήλωσης στιγμιοτύπου:
  `instance` *όνομα_κλάσης τύπος* `where` *δηλώσεις_τιμών*
- Παράδειγμα:

```
instance Condition Bool where
   cond c t e = if c then t else e
instance Condition Int where
   cond c t e = if c/=0 then t else e
instance Condition[a] where
   cond c t e = if c/="" then t else e
```

- Χρήση:

```
cond (length"") 1 2 = 2    cond "a" 1 2 = 1
```

  όχι: `cond 1 1 2`

Υπερφόρτωση
Δήλωση Στιγμιότυπου - II

- Στιγμιότυπο επιτρέπεται να είναι:
  - βασικός τύπος, πχ. `Int`,`Bool` κτλ.
  - τύπος πλειάδας ή λίστας που να περιέχει μόνο μεταβλητές τύπων, πχ. `[a]`
- Στην τελευταία περίπτωση, επιτρέπονται συμφραζόμενα
- Παράδειγμα: υπερφόρτωση της `cond` με ζεύγη τύπων-στιγμιοτύπων της `Condition`
  - βλέπουμε το `(x,y)` ως `x` και `y`

```
instance (Condition a,Condition b)
         => Condition (a,b)
   where cond (x,y) t e = cond x (cond y t e) e
```

- Χρήση:

```
cond (True, "") 1 2 = 2    cond ("a", 2==2) 1 2 = 1
```

Υπερφόρτωση
Δήλωση Στιγμιότυπου - II

- Στιγμιότυπο επιτρέπεται να είναι:
  - βασικός τύπος, πχ. `Int`,`Bool` κτλ.
  - τύπος πλειάδας ή λίστας που να περιέχει μόνο μεταβλητές τύπων, πχ. `[a]`
- Στην τελευταία περίπτωση, επιτρέπονται συμφραζόμενα
- Παράδειγμα: υπερφόρτωση της `cond` με ζεύγη τύπων-στιγμιοτύπων της `Condition`
  - βλέπουμε το `(x,y)` ως `x` και `y`

```
instance (Condition a,Condition b)
         => Condition (a,b)
   where cond (x,y) t e = cond x (cond y t e) e
```

- Χρήση:

```
cond (True, "") 1 2 = 2    cond ("a", 2==2) 1 2 = 1
```

Υπερφόρτωση
Δήλωση Στιγμιότυπου - II

- Στιγμιότυπο επιτρέπεται να είναι:
  - βασικός τύπος, πχ. `Int`,`Bool` κτλ.
  - τύπος πλειάδας ή λίστας που να περιέχει μόνο μεταβλητές τύπων, πχ. `[a]`
- Στην τελευταία περίπτωση, επιτρέπονται συμφραζόμενα
- Παράδειγμα: υπερφόρτωση της `cond` με ζεύγη τύπων-στιγμιοτύπων της `Condition`
  - βλέπουμε το `(x,y)` ως `x` και `y`

```
instance (Condition a,Condition b)
         => Condition (a,b)
   where cond (x,y) t e = cond x (cond y t e) e
```

- Χρήση:

```
cond (True, "") 1 2 = 2    cond ("a", 2==2) 1 2 = 1
```

Υπερφόρτωση
Δήλωση Στιγμιότυπου - II

- Στιγμιότυπο επιτρέπεται να είναι:
  - βασικός τύπος, πχ. `Int`,`Bool` κτλ.
  - τύπος πλειάδας ή λίστας που να περιέχει μόνο μεταβλητές τύπων, πχ. `[a]`
- Στην τελευταία περίπτωση, επιτρέπονται συμφραζόμενα
- Παράδειγμα: υπερφόρτωση της `cond` με ζεύγη τύπων-στιγμιοτύπων της `Condition`
  - βλέπουμε το `(x,y)` ως `x` και `y`

```
instance (Condition a,Condition b)
         => Condition (a,b)
   where cond (x,y) t e = cond x (cond y t e) e
```

- Χρήση:

```
cond (True, "") 1 2 = 2    cond ("a", 2==2) 1 2 = 1
```

Υπερφόρτωση
Δήλωση Στιγμιότυπου - II

- Στιγμιότυπο επιτρέπεται να είναι:
  - βασικός τύπος, πχ. `Int`,`Bool` κτλ.
  - τύπος πλειάδας ή λίστας που να περιέχει μόνο μεταβλητές τύπων, πχ. `[a]`
- Στην τελευταία περίπτωση, επιτρέπονται συμφραζόμενα
- Παράδειγμα: υπερφόρτωση της `cond` με ζεύγη τύπων-στιγμιοτύπων της `Condition`
  - βλέπουμε το `(x,y)` ως `x` και `y`

```
instance (Condition a,Condition b)
         => Condition (a,b)
   where cond (x,y) t e = cond x (cond y t e) e
```

- Χρήση:

```
cond (True, "") 1 2 = 2    cond ("a", 2==2) 1 2 = 1
```

Υπερφόρτωση
Προκαθορισμένες Τιμές

- Η ορισμός τιμής σε μία κλάση, την κάνει προκαθορισμένη (default)
- Τα στιγμιότυπα κρατούν την προκαθορισμένη τιμή εκτός αν την ξαναορίσουν
  - αυτό ονομάζεται ακύρωση (overriding) της προκαθορισμένης δήλωσης

```
class CheckIfBool a where
  notBool :: a->Bool
  notBool _ = True

instance CheckIfBool Bool where
  notBool _ = False

instance CheckIfBool Int
```

Αποτιμήσεις:

```
notBool True = False
notBool(length "") = True
```

Υπερφόρτωση
Προκαθορισμένες Τιμές

- Η ορισμός τιμής σε μία κλάση, την κάνει προκαθορισμένη (default)
- Τα στιγμιότυπα κρατούν την προκαθορισμένη τιμή εκτός αν την ξαναορίσουν
  - αυτό ονομάζεται ακύρωση (overriding) της προκαθορισμένης δήλωσης

```
class CheckIfBool a where
  notBool :: a->Bool
  notBool _ = True
instance CheckIfBool Bool where
  notBool _ = False
instance CheckIfBool Int
```

Αποτιμήσεις:

```
notBool True = False
notBool(length "") = True
```

Υπερφόρτωση
Προκαθορισμένες Τιμές

- Η ορισμός τιμής σε μία κλάση, την κάνει προκαθορισμένη (default)
- Τα στιγμιότυπα κρατούν την προκαθορισμένη τιμή εκτός αν την ξαναορίσουν
  - αυτό ονομάζεται ακύρωση (overriding) της προκαθορισμένης δήλωσης

```
class CheckIfBool a where
  notBool :: a->Bool
  notBool _ = True
instance CheckIfBool Bool where
  notBool _ = False
instance CheckIfBool Int
```

Αποτιμήσεις:

```
notBool True = False
notBool(length "") = True
```

- Η ορισμός τιμής σε μία κλάση, την κάνει προκαθορισμένη (default)
- Τα στιγμιότυπα κρατούν την προκαθορισμένη τιμή εκτός αν την ξαναορίσουν
  - αυτό ονομάζεται ακύρωση (overriding) της προκαθορισμένης δήλωσης

```
class CheckIfBool a where
  notBool :: a->Bool
  notBool _ = True
instance CheckIfBool Bool where
  notBool _ = False
instance CheckIfBool Int
```

Αποτιμήσεις:

```
notBool True = False
notBool(length "") = True
```

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

Υπερφόρτωση
Template Design Pattern

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

Υπερφόρτωση
Template Design Pattern

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

Υπερφόρτωση
Template Design Pattern

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

Υπερφόρτωση
Template Design Pattern

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

- Template: μία προκαθορισμένη τιμή `f` χρησιμοποιεί ένα υπερφορτωμένο όνομα `g`
- Δήλωση της `g` σε ένα στιγμιότυπο, αλλάζει τη συμπεριφορά της `f` αναλόγως

```
class Condition a where
   toBool :: a->Bool
   cond :: a->b->b->b
   cond c t e = if toBool c then t else e
instance Condition Bool where toBool = id
instance Condition Int where toBool i = i/=0
instance
  (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

- Μία τάξη κληρονομεί (inherits) μία άλλη:
  `class` *συμφραζόμενα* `=>` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
  - κληρονομούμενες κλάσεις

```
class ConvBool a where
  toBool :: a->Bool
class ConvBool a => Condition a where
  cond :: a->b->b->b
  cond c t e = if toBool c then t else e
instance ConvBool Bool where toBool = id
instance Condition Bool
...
```

- Μία τάξη κληρονομεί (inherits) μία άλλη:
  `class` *συμφραζόμενα* `=>` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
  - κληρονομούμενες κλάσεις

```
class ConvBool a where
  toBool :: a->Bool
class ConvBool a => Condition a where
  cond :: a->b->b->b
  cond c t e = if toBool c then t else e
instance ConvBool Bool where toBool = id
instance Condition Bool
...
```

- Μία τάξη κληρονομεί (inherits) μία άλλη:
  `class` *συμφραζόμενα* `=>` *όνομα_κλάσης μεταβλητή_τύπου*
  `where` *καθορισμοί_τύπων*
  - κληρονομούμενες κλάσεις

```
class ConvBool a where
  toBool :: a->Bool
class ConvBool a => Condition a where
  cond :: a->b->b->b
  cond c t e = if toBool c then t else e
instance ConvBool Bool where toBool = id
instance Condition Bool
...
```

Υπερφόρτωση

Κλάσεις Haskell: `Eq` και `Ord`

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

class Eq a => Ord a where
  (<) :: a->a->Bool
  (<=) :: a->a->Bool
  ...
  min :: a->a->a
  max :: a->a->a
```

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

class Read a where
  read :: String->a
  ...
```

Ο διερμηνέας χρησιμοποιεί τη `show` για να μας δείξει το αποτέλεσμα ενός υπολογισμού. Στις συναρτήσεις δεν υπάρχει `show`:

```
ERROR - Cannot find "show" function for:
*** Expression : \x -> x
*** Of type : a -> a
```

Κλάση `Num`

- Κλάση `Integral`
  - Τύπος `Int` (4 bytes)
  - Τύπος `Integer` (απεριόριστος)
- Κλάση `Fractional`
  - Τύπος `Rational` (ρητός)
  - Κλάση `Floating`
    - Τύπος `Float`
    - Τύπος `Double` (διπλής ακρίβειας)

Η σταθερά `1` είναι πολυμορφική: `1::Num a=>a`

Περιορισμός έκφρασης `e` σε τύπο `a`: `e::a`

Π.χ. `cond (1::Int) 2 3 = 2`

Κλάση `Num`

- Κλάση `Integral`
  - Τύπος `Int` (4 bytes)
  - Τύπος `Integer` (απεριόριστος)
- Κλάση `Fractional`
  - Τύπος `Rational` (ρητός)
  - Κλάση `Floating`
    - Τύπος `Float`
    - Τύπος `Double` (διπλής ακρίβειας)

Η σταθερά `1` είναι πολυμορφική: `1::Num a=>a`

Περιορισμός έκφρασης `e` σε τύπο `a`: `e::a`

Π.χ. `cond (1::Int) 2 3 = 2`

Κλάση `Num`

- Κλάση `Integral`
  - Τύπος `Int` (4 bytes)
  - Τύπος `Integer` (απεριόριστος)
- Κλάση `Fractional`
  - Τύπος `Rational` (ρητός)
  - Κλάση `Floating`
    - Τύπος `Float`
    - Τύπος `Double` (διπλής ακρίβειας)

Η σταθερά `1` είναι πολυμορφική: `1::Num a=>a`

Περιορισμός έκφρασης `e` σε τύπο `a`: `e::a`

Π.χ. `cond (1::Int) 2 3 = 2`

Υπερφόρτωση

- Standard OOP: single-dispatch class-based (C++/Java)
  - κλάση = τύπος = σύνολο τιμών
- Haskell
  - κλάση = σύνολο τύπων. τύπος = σύνολο τιμών
- Ετερογενείς δομές
  - Java: αναπαριστά `[true,2]`: `true` και `2` είναι αντικείμενα της τάξης `Object`
  - Haskell: όχι! `True::Bool` και `2::Int`
  - Το γεγονός ότι `Eq Bool` και `Eq Int` δεν έχει σχέση
- Δυαδικές μέθοδοι
  - Haskell: μπορεί να περιορίσει το `(==)` σε τιμές ιδίου τύπου
  - Java: όχι! η `equals` της `Object` δεν εισάγει περιορισμό: τα πάντα είναι αντικείμενα της `Object`

Υπερφόρτωση

- Standard OOP: single-dispatch class-based (C++/Java)
  - κλάση = τύπος = σύνολο τιμών
- Haskell
  - κλάση = σύνολο τύπων. τύπος = σύνολο τιμών
- Ετερογενείς δομές
  - Java: αναπαριστά `[true,2]`: `true` και `2` είναι αντικείμενα της τάξης `Object`
  - Haskell: όχι! `True::Bool` και `2::Int`
  - Το γεγονός ότι `Eq Bool` και `Eq Int` δεν έχει σχέση
- Δυαδικές μέθοδοι
  - Haskell: μπορεί να περιορίσει το `(==)` σε τιμές ιδίου τύπου
  - Java: όχι! η `equals` της `Object` δεν εισάγει περιορισμό: τα πάντα είναι αντικείμενα της `Object`

Υπερφόρτωση
Κλάσεις και Αντικειμενοστρεφής Προγραμματισμός

- Standard OOP: single-dispatch class-based (C++/Java)
  - κλάση = τύπος = σύνολο τιμών
- Haskell
  - κλάση = σύνολο τύπων. τύπος = σύνολο τιμών
- Ετερογενείς δομές
  - Java: αναπαριστά `[true,2]`: `true` και `2` είναι αντικείμενα της τάξης `Object`
  - Haskell: όχι! `True::Bool` και `2::Int`
  - Το γεγονός ότι `Eq Bool` και `Eq Int` δεν έχει σχέση
- Δυαδικές μέθοδοι
  - Haskell: μπορεί να περιορίσει το `(==)` σε τιμές ιδίου τύπου
  - Java: όχι! η `equals` της `Object` δεν εισάγει περιορισμό: τα πάντα είναι αντικείμενα της `Object`

Υπερφόρτωση

- Standard OOP: single-dispatch class-based (C++/Java)
  - κλάση = τύπος = σύνολο τιμών
- Haskell
  - κλάση = σύνολο τύπων. τύπος = σύνολο τιμών
- Ετερογενείς δομές
  - Java: αναπαριστά `[true,2]`: `true` και `2` είναι αντικείμενα της τάξης `Object`
  - Haskell: όχι! `True::Bool` και `2::Int`
  - Το γεγονός ότι `Eq Bool` και `Eq Int` δεν έχει σχέση
- Δυαδικές μέθοδοι
  - Haskell: μπορεί να περιορίσει το `(==)` σε τιμές ιδίου τύπου
  - Java: όχι! η `equals` της `Object` δεν εισάγει περιορισμό: τα πάντα είναι αντικείμενα της `Object`

Υπερφόρτωση

- Standard OOP: single-dispatch class-based (C++/Java)
  - κλάση = τύπος = σύνολο τιμών
- Haskell
  - κλάση = σύνολο τύπων. τύπος = σύνολο τιμών
- Ετερογενείς δομές
  - Java: αναπαριστά `[true,2]`: `true` και `2` είναι αντικείμενα της τάξης `Object`
  - Haskell: όχι! `True::Bool` και `2::Int`
  - Το γεγονός ότι `Eq Bool` και `Eq Int` δεν έχει σχέση
- Δυαδικές μέθοδοι
  - Haskell: μπορεί να περιορίσει το `(==)` σε τιμές ιδίου τύπου
  - Java: όχι! η `equals` της `Object` δεν εισάγει περιορισμό: τα πάντα είναι αντικείμενα της `Object`

Επισκόπηση
Πολυμορφισμός

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

Επισκόπηση
Πολυμορφισμός

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

Επισκόπηση
Πολυμορφισμός

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

Επισκόπηση
Πολυμορφισμός

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

Επισκόπηση
Πολυμορφισμός

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

- Πολυμορφισμός: μία τιμή έχει παραπάνω από ένα τύπους, αλλά τον ίδιο ορισμό
  - απαραίτητος όπου υπάρχουν πολλοί τύποι και ένας ορισμός πχ. συναρτήσεις λιστών
- Οι πολυμορφικοί τύποι στη Haskell εκφράζονται με μεταβλητές τύπων
  - ξεκινούν με πεζό γράμμα, συνήθως `a,b,c,...`
- Κανόνες αντικατάστασης μεταβλητών τύπων όπως και στις μεταβλητές τιμών
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη γενικότητας
  - ορισμός βασισμένος στις αντικαταστάσεις
  - γενικότερος τύπος = πιο ευέλικτος τύπος
- Η Haskell εξάγει το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της

Επισκόπηση
Υπερφόρτωση

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής `(::)` περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος

Επισκόπηση
Υπερφόρτωση

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής `(::)` περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος

Επισκόπηση
Υπερφόρτωση

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής `(::)` περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος

Επισκόπηση
Υπερφόρτωση

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής (::) περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος

Επισκόπηση
Υπερφόρτωση

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής `(::)` περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος

- Υπερφόρτωση: ένα όνομα έχει παραπάνω από έναν τύπους και ορισμούς
  - απαραίτητη στις βασικές συναρτήσεις, αλλά η χρησιμότητά της επεκτείνεται
- Κλάσεις: μηχανισμός υπερφόρτωσης
  - σύνολα τύπων με μερικά κοινή υπογραφή
  - εισάγουν περιορισμούς σε πολυμορφικούς τύπους
  - η εξαγωγή τύπων τους ανακαλύπτει
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές, ακύρωση και κληρονομικότητα
  - το Template Design Pattern υποστηρίζεται επαρκώς
- Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί
- Ο τελεστής `(::)` περιορίζει τον τύπο μίας πολυμορφικής έκφρασης
- Διαφορά με sd/cb/OOP: κλάση vs. τύπος