Πολυμορφισμός και Υπερφόρτωση

Γιάννης Κασσιός

Σε αυτές τις σημειώσεις, θα ασχοληθούμε με πιο προχωρημένα θέματα του συστήματος τύπων της Haskell και πιο συγκεκριμένα με τις έννοιες του *πολυμορφισμού (polymorphism)* και της *υπερφόρτωσης (overloading)*. Αυτές οι έννοιες γενικεύουν το σύστημα τύπων της Haskell κάνοντάς το πολύ πιο εύχρηστο.

Στην Ενότ. 1 εξετάζουμε την ιδέα του πολυμορφισμού, δηλ. τη χρήση ενός ορισμού για πολλούς διαφορετικούς τύπους. Εξηγούμε γιατί ο πολυμορφισμός είναι απαραίτητος και εισάγουμε τις *μεταβλητές τύπων* και τους *πολυμορφικούς τύπους* της Haskell. Επίσης, δίνουμε τους πολυμορφικούς τύπους τιμών της Haskell που έχουμε ήδη χρησιμοποιήσει και που έχουν να κάνουν κατά κύριο λόγο με λίστες. Τέλος, εισάγουμε την έννοια της *γενικότητας* τύπων, καθώς και της *εξαγωγής* τύπων από το διερμηνέα της Haskell.

Στην Ενότ. 2 συζητάμε για την υπερφόρτωση, τη χρήση ενός κοινού ονόματος σε διαφορετικούς τύπους και με διαφορετικό ορισμό. Εξηγούμε γιατί αυτό είναι σημαντικό και γιατί δεν καλύπτεται πλήρως από τον πολυμορφισμό. Εισάγουμε τις *κλάσεις* και εξηγούμε τους μηχανισμούς δήλωσης, χρήσης, προκαθορισμένων τιμών και κληρονομικότητας που υποστηρίζουν. Τέλος, αφού παρουσιάσουμε μερικές κλάσεις που υποστηρίζονται από τη Haskell, συγκρίνουμε το μηχανισμό υπερφόρτωσης με το επικρατέστερο μοντέλο αντικειμενοστρεφούς προγραμματισμού που συμπεριλαμβάνει γλώσσες όπως η C++ και η Java.

1 Πολυμορφισμός

1.1 Τι είναι Πολυμορφισμός

Μία τιμή είναι *πολυμορφική (polymorphic)* όταν ανήκει σε *πολλούς τύπους*. Έχουμε ήδη δει πολυμορφικές τιμές. Η κενή λίστα `[]` είναι μία τιμή που ανήκει σε όλους τους τύπους λιστών. Μπορούμε πχ. να τη χρησιμοποιήσουμε είτε σαν κενή συμβολοσειρά

```
"Hi" ++ ""
```

είτε σαν κενή λίστα ακεραίων

```
[1,2,3] ++ []
```

κ.ο.κ. Οι συναρτήσεις για λίστες της Haskell εφαρμόζουν σε κάθε τύπο λίστας και επομένως είναι επίσης πολυμορφικές. Για παράδειγμα η `length` εφαρμόζει σε κάθε τύπο

λίστας και επιστρέφει έναν ακέραιο, ανήκει σε κάθε έναν από τους τύπους `[Int]->Int`, `String->Int` και `[(Char,[Int])]->Int` κλπ.

Μετά από την ενασχόλησή μας με τις λίστες, ο λόγος για τον οποίο έχουμε πολυμορφισμό θα πρέπει να είναι λίγο-πολύ προφανής. Χωρίς πολυμορφισμό, για να έχουμε τη λειτουργικότητα τιμών όπως η `[]` και η `length`, θα έπρεπε να ορίσουμε άπειρο αριθμό αντικειμένων. Κάθε ένα από αυτά τα αντικείμενα θα έπρεπε να έχει διαφορετικό όνομα και τύπο, αλλά οι ορισμοί τους θα ήταν εντελώς ίδιοι:

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

lengthString :: [String]->Int
lengthString emptyString = 0
lengthString (_:xs) = 1+lengthString xs

lengthFunListTupleCharListIntInt :: [(Char,[Int])]->Int
lengthFunListTupleCharListIntInt emptyFunListTupleCharListIntInt
  = 0
lengthFunListTupleCharListIntInt (_:xs)
  = 1+lengthFunListTupleCharListIntInt xs

...
```

Η δυνατότητα πολυμορφισμού της Haskell επιτρέπει την ύπαρξη μίας ενιαίας λίστας `[]` και μίας ενιαίας τιμής `length` με ένα μόνο ορισμό για όλους τους τύπους:

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

1.2 Μεταβλητές Τύπων

Το γεγονός ότι μία τιμή είναι πολυμορφική δε σημαίνει ότι ανήκει σε όλους τους υπάρχοντες τύπους. Για παράδειγμα, η `length` δε μπορεί να είναι ακέραιος αριθμός. Η `length` είναι πάντα μία συνάρτηση, που εφαρμόζεται πάντα σε κάποια λίστα και επιστρέφει πάντα έναν ακέραιο αριθμό. Δηλαδή, όλοι οι τύποι της `length` είναι της μορφής `[a]->Int`, όπου `a` είναι μια *μεταβλητή τύπων (type variable)*, δηλαδή μια μεταβλητή που αντικαθίσταται από οποιονδήποτε τύπο. Ομοίως, η `[]` έχει τύπους της μορφής `[a]`, καθώς είναι πάντα μία λίστα.

Στη Haskell, ο τρόπος απόδοσης τύπου σε πολυμορφικές τιμές είναι η χρήση μεταβλητών τύπου. Οι μεταβλητές τύπου είναι ονόματα που ξεκινάνε με *πεζό* γράμμα για να ξεχωρίζουν από τους σταθερούς τύπους όπως ο `Int`. Έτσι οι ορισμοί τύπων της `length` και της `[]` είναι:

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

Μία έκφραση με μεταβλητές τύπου ονομάζεται *πολυμορφικός τύπος*. Οι μεταβλητές τύπων που χρησιμοποιούμε σε πολυμορφικούς τύπους είναι κατά σύμβαση οι `a,b,c,...`.

Μία μεταβλητή τύπων `a` αποτελεί από μόνη της πολυμορφικό τύπο. Μάλιστα, αποτελεί ένα πολύ γενικό πολυμορφικό τύπο, αφού μπορεί να αντικατασταθεί από οποιονδήποτε άλλο τύπο. Ο πολυμορφικός τύπος `a` είναι ο τύπος της `undef` που είδαμε στις προηγούμενες σήμειώσεις.

Ένας πολυμορφικός τύπος, μπορεί να περιέχει μία μεταβλητή τύπου που να εμφανίζεται περισσότερες από μία φορές. Όπως θα ήταν αναμενόμενο, αυτό σημαίνει ότι η μεταβλητή αυτή θα πρέπει να αντικατασταθεί από τον *ίδιο τύπο* σε κάθε μία από τις εμφανίσεις της. Για παράδειγμα, ο τύπος της `(++)` είναι:

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

Η μεταβλητή `a` θα πρέπει να αντικατασταθεί με τον ίδιο τύπο σε κάθε μία από της τρεις εμφανίσεις της. Έτσι, ο τύπος αυτός περιγράφει τύπους όπως `[Int]->[Int]->[Int]` αλλά όχι `[Int]->[Char]->[Bool]`. Αυτό σημαίνει ότι η `(++)` εφαρμόζει μεν σε λίστες κάθε τύπου, αλλά θα πρέπει τα δύο της ορίσματα να είναι του ιδίου τύπου.

Αντίθετα, σε περίπτωση που έχουμε διαφορετικές μεταβλητές τύπων, αυτές μπορεί να αντικατασταθούν ανεξάρτητα. Για παράδειγμα, η συνάρτηση `zip` εφαρμόζεται σε δύο λίστες, όχι απαραίτητα του ιδίου τύπου. Επομένως οι τύποι των παραμέτρων της θα πρέπει να χρησιμοποιούν δύο διαφορετικές μεταβλητές τύπων:

```
zip :: [a]->[b]->...
```

Το αποτέλεσμα της `zip` είναι μία λίστα ζευγών, των οποίων το πρώτο στοιχείο προέρχεται από την πρώτη παράμετρο και το δεύτερο στοιχείο από τη δεύτερη παράμετρο. Επομένως, ο τύπος του αποτελέσματος θα πρέπει να είναι `[(a,b)]`:

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

Καθώς χρησιμοποιούνται δύο διαφορετικές μεταβλητές τύπου, η `zip` μπορεί να πάρει τον τύπο `[Int] -> [Char] -> [(Int,Char)]`. Φυσικά, κανείς δε μας εμποδίζει να αντικαταστήσουμε τις `a` και `b` με τον ίδιο τύπο, π.χ. η `zip` έχει και τον τύπο `[Int] -> [Int] -> [(Int,Int)]`.

Για να βρούμε έναν κατάλληλο πολυμορφικό τύπο για μία τιμή που ορίζουμε, θα πρέπει να σκεφτούμε τη μορφή της τιμής αυτής, καθώς και το ποιοι περιορισμοί υπάρχουν σχετικά με τους τύπους των επιμέρους στοιχείων που αυτή περιέχει. Για παράδειγμα, έστω η συνάρτηση `dovetail3` των προηγούμενων σημειώσεων:

```
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
```

Παρατηρούμε κατ' αρχήν ότι η `dovetail3` παίρνει τιμές `l1,l2,l3`, οι οποίες πρέπει να είναι λίστες (εφαρμόζονται στον τελεστή `(!!)` που είναι τύπου `[a]->Int->a`). Τα περιεγχόμενα αυτών των λιστών δε συσχετίζονται μεταξύ τους με τρόπο που να δημιουργεί περιορισμό στους τύπους τους. Πράγματι, μπορούμε να έχουμε τρεις λίστες τριών ανεξάρτητων μεταξύ τους τύπων. Όσον αφορά το αποτέλεσμα, από την έκφραση μετασχηματισμού παρατηρούμε ότι έχουμε μία λίστα με τριάδες στοιχείων, ένα από την πρώτη λίστα, ένα από τη δεύτερη και ένα από την τρίτη. Επομένως, αν οι τύποι των τριών παραμέτρων της `dovetail3` είναι `[a]`, `[b]`, `[c]`, τότε ο τύπος του αποτελέσματος είναι `[(a,b,c)]`. Τελικά, ο πολυμορφικός τύπος της `dovetail3` είναι:

```
a->b->c->[(a,b,c)]
```

1.3 Πολυμορφικοί Τύποι Ορισμένων Τιμών της Haskell

Ας δούμε τώρα ένα κατάλογο πολυμορφικών τύπων τιμών της Haskell που έχουμε ήδη χρησιμοποιήσει. Οι περισσότερες περιπτώσεις είναι συναρτήσεις που έχουν να κάνουν με λίστες:

- `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]`

Οι παραπάνω τύποι δεν προκαλούν έκπληξη, εκτός από τις συναρτήσεις `foldl` και `foldr` στις οποίες οι τύποι είναι πιο γενικοί από αυτούς που ίσως θα περιμέναμε με δεδομένο το πώς χρησιμοποιούσαμε αυτές τις συναρτήσεις μέχρι τώρα. Συγκεκριμένα, εμείς χρησιμοποιούσαμε τις συναρτήσεις στην ειδική τους περίπτωση που οι τύποι `a` και `b` είναι ίσοι π.χ. η

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

χρησιμοποιεί τη συνάρτηση `(+) :: Int->Int`.

Στη γενική περίπτωση, αυτό δεν είναι απαραίτητο. Η `foldl` μπορεί να συσσωρεύει αποτέλεσμα τύπου `a` πάνω σε λίστα τύπου `[b]`, χρησιμοποιώντας μία συνάρτηση `a->b->a` (παίρνει το ήδη συσσωρευμένο αποτέλεσμα και το εκάστοτε στοιχείο της λίστας και επιστρέφει το νέο συσσωρευμένο αποτέλεσμα). Για παράδειγμα, ένας τρόπος εύρεσης ενός ακεραίου σε μία λίστα χωρίς τη χρήση της `map` είναι η αντικατάσταση του `b` με `Int` και του `a` με `Bool`. Η συνάρτηση συσσώρευσης είναι:

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

(όπου x είναι το στοιχείο που ψάχνουμε). Η συνάρτηση αυτή ελέγχει για κάθε στοιχείο της λίστας αν το x έχει ήδη βρεθεί (η τιμή της found έχει ήδη γίνει True) ή αν το τρέχον στοιχείο current είναι ίσο με το x. Η νέα συνάρτηση γραμμικής αναζήτησης newlSearch είναι:

```
newlSearch :: Int->[Int]->Bool
newlSearch x l
  = foldl (\found-> \current-> found || current == x) False l
```

Ομοίως, η foldr μπορεί να συσσωρεύει διαφορετικού τύπου στοιχεία από αυτά που βρίσκονται στη λίστα.

1.4 Γενικότητα των Πολυμορφικών Τύπων

Μία *αντικατάσταση (substitution)* είναι μία συνάρτηση που συσχετίζει πεπερασμένο αριθμό μεταβλητών τύπων με (πιθανώς πολυμορφικούς) τύπους. Αν σ είναι μία αντικατάσταση και T ένας πολυμορφικός τύπος, τότε θα γράφουμε σT τον πολυμορφικό τύπο που προκύπτει αν αντικαταστήσουμε κάθε μεταβλητή τύπων στον T με τη συσχέτισή της από την σ. Για παράδειγμα, αν

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

τότε

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

Υπάρχει μία σημαντική σχέση διάταξης μεταξύ των πολυμορφικών τύπων, η σχέση της *γενικότητας*. Θα λέμε ότι ένας πολυμορφικός τύπος A είναι *πιο γενικός* από έναν άλλο B και ότι ο B είναι *πιο ειδικός* από τον A, αν υπάρχει αντικατάσταση σ ώστε $\sigma A = \sigma B$. Ο πιο γενικός τύπος είναι η απλή μεταβλητή τύπων a, η οποία μπορεί να μετατραπεί σε οποιονδήποτε άλλο τύπο T από την αντικατάσταση $\texttt{a} \mapsto T$.

Ένας πιο γενικός τύπος δίνει περισσότερη ελευθερία επιλογής συγκεκριμένου μονομορφικού τύπου για μία τιμή. Για παράδειγμα, ο τύπος (a,b) είναι πιο γενικός από τον (a,a). Μαθηματικά, αυτό δείχνεται χρησιμοποιώντας την αντικατάσταση $\texttt{b} \mapsto \texttt{a}$. Πρακτικά, αυτό σημαίνει ότι ο τύπος (a,b) επιτρέπει περισσότερους μονομορφικούς τύπους από τον (a,a). Πράγματι, ο (a,b) περιγράφει όλους τους τύπους ζεύγους, όπως (Int,Char) και (Int,Int), ενώ ο (a,a) περιγράφει μόνο τους τύπους ζεύγους ομοειδών στοιχείων, όπως (Int,Int).

Όσο πιο γενικό τύπο δώσουμε σε μία τιμή, τόσο λιγότερους περιορισμούς δίνουμε για τη χρήση αυτής της τιμής. Για παράδειγμα, αν η [] είχε λιγότερο γενικό τύπο, π.χ. [a->b], τότε θα μπορούσε να χρησιμοποιηθεί μόνο για λίστες συναρτήσεων και όχι για όλες τις λίστες. Επομένως μας ενδιαφέρει να δίνουμε όσο το δυνατό πιο γενικό πολυμορφικό τύπο στις τιμές που ορίζουμε.

Σε περίπτωση που δε δίνουμε τύπο σε μία τιμή που ορίζουμε, η Haskell κάνει *εξαγωγή τύπου (type inference)* φροντίζοντας να βρει το γενικότερο πολυμορφικό τύπο

που τη χαρακτηρίζει. Το ίδιο συμβαίνει και με τις λ-εκφράσεις. Για να δούμε τον τύπο που έχει αποδώσει η Haskell σε κάποια τιμή, μπορούμε να δώσουμε την εντολή `:type` στον Hugs. Για παράδειγμα, η εντολή

```
:type dovetail3
```

θα δώσει

```
dovetail3 :: a -> b -> c -> [(a,b,c)]
```

Η εντολή

```
:type \x->[x]
```

θα δώσει

```
\x -> [x] :: a -> [a]
```

2 Υπερφόρτωση και Κλάσεις Τύπων

2.1 Τι είναι η Υπερφόρτωση

Ένα όνομα λέμε ότι είναι *υπερφορτωμένο (overloaded)* όταν (α) έχει περισσότερους από έναν τύπους και (β) για κάθε έναν από τους τύπους του έχει διαφορετικό ορισμό. Η υπερφόρτωση λοιπόν διαφέρει από τον πολυμορφισμό, στον οποίο ο ίδιος ορισμός εφαρμόζεται σε όλους τους τύπους της πολυμορφικής τιμής.

Έχουμε ήδη χρησιμοποιήσει υπερφορτωμένα ονόματα της Haskell. Ο τελεστής `(+)` εφαρμόζεται και σε ακεραίους και σε πραγματικούς αριθμούς, αλλά έχει διαφορετική υλοποίηση σε κάθε μία από τις περιπτώσεις αυτές. Ομοίως τελεστές όπως οι `(==)` και `(<)` κ.α. είναι υπερφορτωμένοι. Για παράδειγμα, ο τελεστής `(==)` για ακεραίους είναι υλοποιημένος από το διερμηνέα της Haskell (built-in), ενώ ο τελεστής για ζεύγη δίνεται από τον τύπο

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

Εδώ προϋποτείθεται ότι ο `(==)` είναι ορισμένος για τον τύπο των `x,z` καθώς και για τον τύπο των `y,w`. Η εφαρμογή του σε αυτά τα στοιχεία αναφέρεται στον τελεστή `(==)` των αντίστοιχων τύπων.

Ο λόγος που η Haskell υποστηρίζει υπερφόρτωση, ξεκινάει από τις βασικές τιμές: είναι άβολο να έχουμε διαφορετικά σύμβολα για την ισότητα, την πρόσθεση κτλ. για κάθε βασικό τύπο που θέλουμε να υποστηρίζει αυτές τις πράξεις. Αλλά η σημασία της υπερφόρτωσης των τελεστών της Haskell επεκτείνεται και πέρα από τις βασικές τιμές.

Για παράδειγμα, ας πάρουμε πάλι την αγαπημένη μας γραμμική αναζήτηση, αυτή τη φορά γραμμένη πιο απλά:

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

Οι λίστες που ψάχνει αυτή η συνάρτηση είναι λίστες στοιχείων τύπων για τους οποίους ο τελεστής ισότητας `(==)` υποστηρίζεται. Έτσι είναι δυνατόν να αναζητήσουμε στοιχεία σε λίστες ακεραίων, χαρακτήρων ή και πλειάδων από τέτοια στοιχεία, αλλά δεν είναι δυνατό να αναζητήσουμε στοιχεία σε λίστες συναρτήσεων, καθώς ο τελεστής ισότητας δεν υποστηρίζεται για συναρτήσεις.

Χωρίς την υπερφόρτωση του τελεστή `(==)` θα έπρεπε να γράψουμε μία συνάρτηση γραμμικής αναζήτησης για κάθε τύπο που υποστηρίζει κάποια ισότητα και επιπλέον ο ορισμός αυτών των συναρτήσεων θα ήταν πάντα ο ίδιος. Αυτή η κατάσταση, όπως είδαμε και στην περίπτωση του πολυμορφισμού είναι απαράδεκτη.

Ένα θέμα που προκύπτει τώρα είναι τι τύπο θα δώσουμε στην `lSearch`. Η `lSearch` έχει βέβαια τον πολυμορφικό τύπο `a -> [a] -> Bool`, αλλά χρειάζεται να εκφράσουμε έναν επιπλέον περιορισμό: για τον τύπο `a` πρέπει να υποστηρίζεται τελεστής ισότητας. Αυτός ο περιορισμός θα εξασφαλίσει ότι η `lSearch` έχει μεν τύπο `Int -> [Int] -> Bool`, αλλά όχι και `(Int->Int) -> [Int->Int] -> Bool`. Με το μηχανισμό των πολυμορφικών τύπων που εξηγήσαμε στην προηγούμενη ενότητα, αυτός ο περιορισμός δε μπορεί να εκφραστεί.

Φυσικά θα μπορούσαμε να παρέχουμε στη συνάρτηση `lSearch` τη συνάρτηση ισότητας ως μια έξτρα παράμετρο:

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

αλλά αυτό θα έδινε μια πολύ γενικότερη συνάρτηση από αυτή που θέλουμε να φτιάξουμε (μπορούμε τώρα να περάσουμε οποιαδήποτε συνάρτηση και όχι απλά ισότητα) και θα περιέπλεκε χωρίς ουσιαστικό αντίκρυσμα όχι μόνο τον ορισμό, αλλά και τη χρήση της `lSearch`.

Στο υπόλοιπο αυτής της ενότητας, θα δούμε πώς ο μηχανισμός των *κλάσεων* της Haskell αντιμετωπίζει το θέμα έκφρασης αυτών των περιορισμών και κάνει δυνατή τη συμβίωση της υπερφόρτωσης με το πολυμορφικό σύστημα.

2.2 Κλάσεις τύπων

Μία *κλάση (class)* είναι ένα σύνολο τύπων για τους οποίους υποστηρίζονται υπερφορτωμένες εκδόσεις κάποιων ονομάτων. Για παράδειγμα "οι τύποι για τους οποίους υποστηρίζεται ένας τελεστής ισότητας" είναι μία κλάση που στη Haskell ονομάζεται `Eq`. Αν ένας τύπος `a` ανήκει στην κλάση `Eq`, τότε γράφουμε `Eq a` και εννοούμε ότι υπάρχει έκδοση της συνάρτησης `(==)` με τύπο `a->a->Bool`.

Πιο πάνω είδαμε ότι θέλουμε στον ορισμό τύπου της `lSearch` να εκφράσουμε την προϋπόθεση "για τον τύπο `a` υποστηρίζεται τελεστής ισότητας". Αυτή η προϋπόθεση μπορεί να διατυπωθεί με λόγια "ο τύπος `a` ανήκει στην κλάση `Eq`". Στη Haskell μπορεί να γραφεί `Eq a` και να μπει μπροστά από τον τύπο της `lSearch` ως εξής:

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

Το παραπάνω λέει ότι η `lSearch` παίρνει όλους τους τύπους `a->[a]->Bool` για τους οποίους το a είναι στην κλάση `Eq`. Βλέπουμε ότι μεταξύ της δήλωσης `Eq a` και του πολυμορφικού τύπου εισάγεται ο καινούριος για εμάς τελεστής `=>`.

Η δήλωση `Eq a` ονομάζεται *περιορισμός (constraint)*. Ένας καθορισμός τύπου μπορεί να έχει περισσότερους από έναν περιορισμούς. Σε αυτήν την περίπτωση οι περιορισμοί μπαίνουν σε παρένθεση και χωρίζονται με κόμμα, π.χ.

```
somevalue :: (Eq a, Eq b) => ...
```

Το σύνολο όλων των περιορισμών ενός καθορισμού τύπου (που μπαίνει αριστερά από τον τελεστή `=>`) ονομάζεται *συμφραζόμενα (context)* του καθορισμού τύπου.

Ας δούμε τώρα τι μπορούμε να κάνουμε με την `lSearch`, της οποίας ο πλήρης ορισμός είναι:

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

Οι ερωτήσεις

```
lSearch True [False,True]
lSearch 20 [1..10]
lSearch ('a',50) [(c,i) | c<-['a','z'], i<-[0..100]]
```

δίνουν αντίστοιχα τις αναμενόμενες απαντήσεις `True,False,True`. Ο έλεγχος τύπου δηλαδή επιτυγχάνει και ο διερμηνέας προχωράει στον υπολογισμό. Αν όμως προσπαθήσουμε την εξής ερώτηση:

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

παίρνουμε την απάντηση:

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

Εδώ βλέπουμε ότι ο έλεγχος τύπου αποτυγχάνει. Ο διερμηνέας αναφέρει ότι ο τύπος `a->a` δεν είναι στην κλάση `Eq` όπως ζητάει ο περιορισμός που θέσαμε στον καθορισμό τύπου της `lSearch`.

Σε περίπτωση που δεν υπάρχει καθορισμός τύπου για ένα οριζόμενο όνομα, η διαδικασία εξαγωγής τύπου ανακαλύπτει και τα συμφραζόμενα που χρειάζεται ένα όνομα για να λειτουργήσει. Για παράδειγμα, αν στον παραπάνω ορισμό της `lSearch` βγάλουμε τον καθορισμό τύπου, και έπειτα δώσουμε

```
:type lSearch
```

στο διερμηνέα, η απάντηση που θα πάρουμε είναι

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

Εδώ βλέπουμε πως η εξαγωγή τύπου παρατήρησε τη χρήση του τελεστή ισότητας πάνω σε στοιχεία τύπου `a` και άρα την ανάγκη για τον περιορισμό `Eq a`.

2.3 Δήλωση και Χρήση Κλάσεων

Για να δημιουργήσουμε μία δική μας κλάση, χρησιμοποιούμε τη λέξη-κλειδί `class` με την εξής σύνταξη:
`class` *όνομα_κλάσης μεταβλητή_τύπου* `where` *καθορισμοί_τύπων*
Στους καθορισμούς τύπων συμμετέχει η μεταβλητή τύπου της δήλωσης κλάσης. Οι καθορισμοί αυτοί εισάγουν ονόματα τα οποία πρέπει να υποστηρίζονται για κάποιο τύπο που ανήκει στην κλάση την οποία δημιουργούμε. Οι καθορισμοί αυτοί ονομάζονται *υπογραφή (signature)* της κλάσης.

Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να φτιάξουμε μία υπερφορτωμένη συνάρτηση `cond` η οποία να λειτουργεί σαν την `if then else` της Haskell, όμως να παίρνει σα συνθήκη όχι μόνο μία τιμή `Bool`, αλλά και ακέραιους αριθμούς (όπως στις C/C++) ή ακόμα και λίστες (όπως σε script γλώσσες). Όταν η συνθήκη `c` είναι ένας αριθμός, τότε, όπως στις C/C++, το αποτέλεσμα της `cond c t e` θα είναι `t` αν ο αριθμός είναι διάφορος του μηδενός και `e` αν είναι ίσος με το `0`. Όταν η `c` είναι λίστα, το αποτέλεσμα θα είναι `e` αν η λίστα είναι κενή, διαφορετικά θα είναι `t`.

Θα ορίσουμε μία κλάση `Condition` που θα περιέχει τους τύπους που μπορούν να χρησιμοποιηθούν ως συνθήκες στην `cond`. Η υπογραφή της `Condition` θα περιέχει την `cond`, που είναι μία πολυμορφική συνάρτηση, καθώς τα δύο τελευταία της ορίσματα μπορεί να ανήκουν σε οποιονδήποτε τύπο:

```
class Condition a where
  cond :: a->b->b->b
```

Αυτή η δήλωση λέει: Ορίζω μία κλάση `Condition`. Ένας τύπος `a` ανήκει σε αυτήν την κλάση αν και μόνον αν υποστηρίζεται η εξής υπερφορτωμένη συνάρτηση `cond :: a->b->b->b`. Ο τύπος της συνάρτησης αυτής μας λέει ότι η `cond` παίρνει μία παράμετρο του τύπου `a` (που ανήκει στην τάξη `Condition`) καθώς και δύο ακόμα παραμέτρους του ιδίου τύπου `b` και επιστρέφει μία τιμή τύπου `b`.

Ένας τύπος που ανήκει σε μια κλάση `C` ονομάζεται *στιγμιότυπο (instance)* της `C`. Για να δημιουργήσουμε στιγμιότυπα μία κλάσης, χρησιμοποιούμε τη λέξη-κλειδί `instance` στην εξής σύνταξη:
`instance` *όνομα_κλάσης τύπος* `where` *δηλώσεις_συναρτήσεων*
Στις δηλώσεις συναρτήσεων, δηλώνονται ακριβώς οι συναρτήσεις που προβλέπει η κλάση.

Για παράδειγμα, η `Bool` δηλώνεται ως στιγμιότυπο της `Condition` με προφανή ορισμό:

```
instance Condition Bool where
  cond c t e = if c then t else e
```

Για την περίπτωση των ακεραίων αριθμών, πρέπει να δηλώσουμε την `Int` ως στιγμιότυπο της `Condition` ως εξής:

```
instance Condition Int where
  cond c t e = if c/=0 then t else e
```

Τώρα μπορούμε να δοκιμάσουμε την ερώτηση

```
cond (length "a") "It worked!" "Nope"
```

και, φυσικά, το αποτέλεσμα είναι `"It worked"`. Όμως μη δοκιμάσετε (ακόμα) την ερώτηση `cond 1 2 3`. Στην πραγματικότητα, η σταθερά `1` είναι πολυμορφική, και ο διερμηνέας θα μπερδευτεί.

Για τις λίστες γράφουμε:

```
instance Condition [a] where
  cond [] t e = e
  cond (_:_) t e = t
```

Και τώρα η παραπάνω ερώτηση μπορεί να γίνει:

```
cond "a" "It worked!" "Nope"
```

Στις δηλώσεις στιγμιοτύπων, υπάρχουν συγκεκριμένοι περιορισμοί σχετικά με τους τύπους που μπορούμε να δηλώσουμε. Μπορούμε να δηλώσουμε είτε βασικούς τύπους (`Bool`, `Int` κτλ.) είτε τύπους λιστών ή πλειάδων που όμως περιέχουν μόνο μεταβλητές τύπων, πχ. `[a]`.

Στην τελευταία περίπτωση, επιτρέπεται να υπάρχουν ακόμα και συμφραζόμενα που περιορίζουν τους τύπους στους οποίους γίνεται η υπερφόρτωση. Θα μπορούσαμε για παράδειγμα να υπερφορτώσουμε την `cond` για ζεύγη τιμών, οι τύποι των οποίων ανήκουν στην κλάση `Condition`, και να ελέγχουμε τη σύζευξη αυτών των τύπων:

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

Αυτό σημαίνει ότι ένα ζεύγος `(x,y)` συμπεριφέρεται σαν `True` αν και μόνον αν και το `x` και το `y` συμπεριφέρονται σαν `True`. Στην ερώτηση

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

η Haskell θα απαντήσει `2`, ενώ στην

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

θα απαντήσει `1`.

2.4 Προκαθορισμένες Δηλώσεις και Κληρονομικότητα

Στη δήλωση μίας κλάσης, γίνεται να δίνεται μια *προκαθορισμένη (default)* τιμή για κάποια από τα υπερφορτωμένα ονόματα. Αν η δήλωση ενός στιγμιοτύπου παραλείψει τον ορισμό μιας τιμής που έχει προκαθορισμένη τιμή στην κλάση, τότε η προκαθορισμένη τιμή χρησιμοποιείται. Σε περίπτωση όμως που το στιγμιότυπο ορίζει το όνομα, τότε η προκαθορισμένη τιμή *ακυρώνεται (γίνεται override)* και ισχύει η δήλωση του στιγμιοτύπου.

Για παράδειγμα, στην παρακάτω δήλωση κλάσης έχουμε μία προκαθορισμένη τιμή για τη συνάρτηση `notBool`. Στις δηλώσεις στιγμιοτύπων που ακολουθούν, η προκαθορισμένη τιμή ακυρώνεται μόνο για τον τύπο `Bool`. Ο τύπος `Int` κρατάει την προκαθορισμένη τιμή:

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

instance CheckIfBool Bool where
  notBool _ = False

instance CheckIfBool Int
```

Τώρα οι αποτιμήσεις των

```
notBool True
notBool (length"")
```

δίνουν αντίστοιχα `False` και `True`. Το παράδειγμα αυτό δεν έχει ιδιαίτερη πρακτική αξία, αλλά γίνεται για λόγους επίδειξης.

Μία δυνατότητα που μας δίνεται με τις προκαθορισμένες τιμές και την ακύρωση είναι κάτι που μοιάζει με το Template Design Pattern [GHJV95] που συναντάμε κυρίως στον αντικειμενοστρεφή προγραμματισμό. Αν ο προκαθορισμένος ορισμός ενός ονόματος περιλαμβάνει άλλα υπερφορτωμένα ονόματα, τότε η συμπεριφορά του ορισμού αυτού αλλάζει, ανάλογα με τον ορισμό που δίνουν στα ονόματα τα εκάστοτε στιγμιότυπα.

Στην περίπτωση της κλάσης `Condition`, θα μπορούσαμε να χρησιμοποιήσουμε μία συνάρτηση `toBool` η οποία θα μετατρέπει το όρισμά της σε τιμή `Bool`, και να δώσουμε ένα προκαθορισμένο ορισμό στην `cond` που να χρησιμοποιεί αυτή τη συνάρτηση:

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

Τώρα, τα στιγμιότυπά μας χρειάζεται να ορίσουν μόνο τη συνάρτηση `toBool`. Η συνάρτηση `cond` θα κρατήσει την προκαθορισμένη τιμή της και θα συμπεριφέρεται σε κάθε περίπτωση ανάλογα:

```
instance Condition Bool where
  toBool = id

instance Condition Int where
  toBool i = i/=0

instance Condition [a] where
  toBool [] = False
  toBool (_:_) = True

instance (Condition a,Condition b) => Condition (a,b) where
  toBool (x,y) = toBool x && toBool y
```

Οι δηλώσεις αυτές είναι πιο εύκολες και κατανοητές και όλα τα παραδείγματα χρήσης της `cond` που δείξαμε στην προηγούμενη ενότητα δουλεύουν.

Μία κλάση μπορεί να *κληρονομήσει (inherit)* μία ή περισσότερες κλάσεις, υιοθετώντας όλους τους ορισμούς των δηλώσεων των κλάσεων αυτών. Σε αυτήν την περίπτωση, οι κλάσεις που κληρονομούνται μπαίνουν σε συμφραζόμενα μπροστά από την κλάση που δηλώνεται ως εξής:
`class` *συμφραζόμενα* `=>` *όνομα_κλάσης μεταβλητή_τύπου* `where` *καθορισμοί_τύπων*
Στο παράδειγμά μας, θα μπορούσαμε να είχαμε δηλώσει δύο κλάσεις `ConvBool` και `Condition` με την `Condition` να κληρονομεί την `ConvBool`:

```
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
```

Στις δηλώσεις στιγμιοτύπων θα έπρεπε να δηλώσουμε και τις δύο κλάσεις:

```
instance ConvBool Bool where
  toBool = id
instance Condition Bool

instance ConvBool Int where
  toBool i = i/=0
instance Condition Int
```

```
instance ConvBool [a] where
  toBool [] = False
  toBool (_:_) = True
instance Condition [a]

instance (ConvBool a,ConvBool b) => ConvBool (a,b) where
  toBool (x,y) = toBool x && toBool y
instance (Condition a, Condition b) => Condition (a,b)
```

κάτι που δε βολεύει στο συγκεκριμένο παράδειγμα, καθώς όλα τα στιγμιότυπα της `ConvBool` είναι και στιγμιότυπα της `Condition`.

2.5 Οι Κλάσεις της Haskell

Σε αυτήν την ενότητα παρουσιάζουμε μερικές από τις κλάσεις που υποστηρίζονται από τη Haskell.

2.5.1 `Eq`

Έχουμε δει την κλάση `Eq` που ορίζεται ως εξής:

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

Οι προκαθορισμένοι ορισμοί είναι τέτοιοι ώστε να χρειάζεται ο ορισμός μόνο μίας εκ των `(==)` και `(/=)` για να λειτουργήσουν και οι δύο.

2.5.2 `Ord`

Η κλάση `Ord` κληρονομεί την `Eq` και προσθέτει τελεστές σύγκρισης `> >= < <=` και συναρτήσεις `min` και `max` για εύρεση ελαχίστου και μεγίστου. Ένα κομμάτι της δήλωσής της έχει ως εξής:

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

2.5.3 `Show` και `Read`

Η κλάση `Show` συμπεριλαμβάνει τη συνάρτηση `show` που μετατρέπει ένα όρισμα σε `String`:

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

Ο διερμηνέας της Haskell χρησιμοποιεί τη `show` οποιασδήποτε τιμής θέλει να παρουσιάσει στο χρήστη. Έτσι, οι βασικές τιμές, οι λίστες και οι πλειάδες, που ανήκουν στην τάξη `Show` μπορούν να εμφανιστούν. Αν όμως ζητήσουμε την τιμή μιας συνάρτησης, πχ. `\x->x`, ο διερμηνέας θα διαμαρτυρηθεί ότι δεν ξέρει πως να εμφανίσει αυτή τη συνάρτηση, καθώς οι συναρτήσεις δεν ανήκουν στην κλάση `Show`:

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

Η κλάση `Read` υποστηρίζει, αντίστροφα, μία συνάρτηση `read` που μετατρέπει μία συμβολοσειρά στον αντίστοιχο τύπο:

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

2.5.4 Αριθμητικές κλάσεις

Η Haskell υποστηρίζει ένα πολύ σύνθετο σύνολο κλάσεων για αριθμητικούς τύπους και πολλούς βασικούς αριθμητικούς τύπους (εμείς μέχρι στιγμής χρησιμοποιούσαμε μόνο το `Int` και το `Float`). Η γενικότερη αριθμητική κλάση είναι η `Num`, η οποία υπόστηρίζει συναρτήσεις όπως `(+)`,`(-)`,`(*)` κτλ.

Η κλάση `Integral` περιέχει τους τύπους ακεραίων αριθμών `Int` (ακρίβεια 4 bytes) και `Integer` (όλοι οι ακέραιοι).

Η κλάση `Fractional` περιέχει τον τύπο `Rational` (ρητοί) και κληρονομείται από την κλάση `Floating` που περιέχει τους τύπους `Float` και `Double` (αριθμοί κινητής υποδιαστολής διπλής ακρίβειας).

Αριθμητικές σταθερές όπως η `2` είναι υπερφορτωμένες, όπως ακριβώς και οι συναρτήσεις `(+)`,`(-)` κτλ. Για παράδειγμα, ο τύπος της `2` είναι `Num a => a` που σημαίνει ότι το `2` έχει όλους τους τύπους της `Num` (όπως `Int`,`Rational`,`Float` κτλ.). Ο τύπος της `(+)` είναι `Num a => a->a->a`. Το αποτέλεσμα είναι να μπορούμε να γράψουμε εκφράσεις όπως `2+2.0`.

Αν θέλουμε να περιορίσουμε τεχνητά τον τύπο μίας πολυμορφικής έκφρασης όπως η σταθερά 2, τότε μπορούμε να χρησιμοποιήσουμε τον τελεστή `::` ακολουθούμενο από τον τύπο στον οποίο θέλουμε να τη χρησιμοποιήσουμε. Για παράδειγμα, για να κάνουμε την έκφραση που είδαμε πιο πάνω `cond 1 2 3` να δουλέψει, πρέπει να ενημερώσουμε τη Haskell ότι εννοούμε την `Int` έκδοση της σταθεράς `1`:

```
cond (1::Int) 2 3
```

που αποτιμάται σε 2.

2.6 Κλάσεις και Αντικειμενοστρεφής Προγραμματισμός

Σε αυτές τις σημειώσεις είδαμε πολλούς όρους όπως κλάση, ακύρωση, πολυμορφισμός, κληρονομικότητα κτλ. που παραπέμπουν στο στυλ του αντικειμενοστρεφούς προγραμματισμού. Όντως, οι μηχανισμοί που είδαμε είναι πολύ κοντά στους αντίστοιχους μηχανισμούς αντικειμενοστρεφών γλωσσών.

Σε σύγκριση με τις μακράν πιο διαδεδομένες αντικειμενοστρεφείς γλώσσες προγραμματισμού, δηλαδή τις single-dispatch class-based γλώσσες όπως η C++ και η Java, το μοντέλο της Haskell διαφέρει στο ότι διαχωρίζει τους *τύπους* από τις *κλάσεις*, έννοιες που συμπίπτουν σε αυτές τις γλώσσες.

Ένα μειονέκτημα της φιλοσοφίας του συστήματος της Haskell είναι ότι δε μπορεί να κατασκευάσει δομές που να περιέχουν αντικείμενα πολλών τύπων. Για παράδειγμα, η λίστα `[True,2]` είναι δυνατό να παρασταθεί στη Java. Στη Java, και τα δύο αντικείμενα (`true` και 2) είναι αντικείμενα της ίδιας κλάσης (`Object`) και άρα ανήκουν στον ίδιο τύπο. Στη Haskell, και τα δύο αντικείμενα ανήκουν σε τύπους που ανήκουν στην κλάση `Eq`, αλλά δεν υπάρχει τύπος στον οποίο ανήκουν και τα δύο αντικείμενα.

Ένα πλεονέκτημα της Haskell και άλλων γλωσσών που διαχωρίζουν κλάσεις από τύπους είναι ότι μπορούν να επιβάλλουν περιορισμούς στη χρήση των υπερφορτωμένων συναρτήσεων που οι single-dispach class-based γλώσσες δε μπορούν. Για παράδειγμα, η κλάση `Eq` υποστηρίζει μία συνάρτηση `(==)` που μπορεί να κληθεί μόνο με παραμέτρους αντικείμενα του ιδίου τύπου. Έτσι ένας ακέραιος δε μπορεί να συγκριθεί με μία αληθοτιμή κτλ. Στις Java/C++ δεν υπάρχει κάποια τέτοια δυνατότητα. Η συμπερίληψη της μεθόδου `equals` στην ανώτατη τάξη `Object` δεν εμποδίζει τέτοιες συγκρίσεις, καθώς όλα τα αντικείμενα ανήκουν σε αυτήν την τάξη. Το πρόβλημα αυτό των αντικειμενοστρεφών γλωσσών είναι μέρος ενός μεγαλύτερου προβλήματος που αντιμετωπίζουν αυτές οι γλώσσες, γνωστό και ως πρόβλημα των *δυαδικών μεθόδων (binary methods)* [BCC+95].

3 Επισκόπηση

Σε αυτές τις σημειώσεις είδαμε τους μηχανισμούς πολυμορφισμού και υπερφόρτωσης που χρησιμοποιεί το σύστημα τύπων της Haskell:

- Η Haskell υποστηρίζει *πολυμορφισμό*. Αυτό σημαίνει ότι υπάρχουν τιμές που έχουν παραπάνω από έναν τύπο. Οι τιμές αυτές λέγονται *πολυμορφικές*.

- Ο πολυμορφισμός είναι απαραίτητος για περιπτώσεις που μία τιμή χρειάζεται να λειτουργήσει με τον ίδιο ορισμό σε πολλούς διαφορετικούς τύπους, όπως είναι οι περιπτώσεις των συναρτήσεων για λίστες της Haskell.
- Ένας *πολυμορφικός τύπος* στη Haskell περιλαμβάνει *μεταβλητές τύπων*. Αυτές ξεκινούν με πεζό γράμμα για να ξεχωρίζουν από τους σταθερούς τύπους.
- Οι μεταβλητές τύπων έχουν τους ίδιους κανόνες αντικατάστασης με τις μεταβλητές τιμών. Αυτό σημαίνει ότι μία μεταβλητή τύπων αντικαθίσταται από τον ίδιο τύπο σε κάθε της εμφάνιση, ενώ τέτοιος περιορισμός δεν υπάρχει για δύο διαφορετικές μεταβλητές τύπων.
- Οι πολυμορφικοί τύποι διαθέτουν διάταξη *γενικότητας* βασισμένη στις *αντικαταστάσεις*. Όσο πιο γενικός είναι ένας τύπος, σε τόσο περισσότερες περιστάσεις μπορούν να χρησιμοποιηθούν οι τιμές του.
- Η Haskell *εξάγει* το γενικότερο τύπο μίας τιμής, αν δε δώσουμε εμείς τύπο.
- Η εντολή `:type` του διερμηνέα επιστρέφει τον τύπο του ορίσματός της.
- Ένα όνομα είναι *υπερφορτωμένο* αν ανήκει σε πολλούς τύπους και έχει διαφορετικό ορισμό για κάθε έναν από αυτούς.
- Η υπερφόρτωση είναι σημαντική σε βασικές τιμές της Haskell όπως `(+)`, `(-)` κτλ. αλλά η σημασία της επεκτείνεται και σε όσες τιμές ορίζονται χρησιμοποιώντας υπερφορτωμένες τιμές.
- Οι *κλάσεις* είναι ο μηχανισμός υπερφόρτωσης της Haskell. Μία κλάση είναι ένα σύνολο τύπων για τους οποίους υποστηρίζονται κάποιες υπερφορτωμένες τιμές.
- Οι υπερφορτωμένες τιμές που επιβάλλει μία κλάση λέγονται *υπογραφή* της κλάσης.
- Με τη χρήση κλάσεων μπορούμε να εισάγουμε περιορισμούς στις μεταβλητές τύπων ενός πολυμορφικού τύπου.
- Η εξαγωγή τύπων της Haskell ανακαλύπτει αυτούς τους περιορισμούς.
- Οι κλάσεις υποστηρίζουν προκαθορισμένες τιμές και ακύρωση, επιτρέποντας τη χρήση μοτίβων προγραμματισμού όπως το Template Design Pattern.
- Οι κλάσεις υποστηρίζουν κληρονομικότητα.
- Η Haskell υποστηρίζει πολλές προκαθορισμένες τάξεις. Υπάρχει πολύπλοκη δομή αριθμητικών κλάσεων. Ακόμα και οι σταθεροί αριθμοί είναι πολυμορφικοί.
- Μπορούμε να χρησιμοποιήσουμε τον τελεστή `(::)` για να περιορίσουμε τον τύπο μίας πολυμορφικής έκφρασης.
- Ο μηχανισμός των κλάσεων έχει μία σημαντική διαφορά σε σχέση με τον αντίστοιχο σε single-dispatch class-based αντικειμενοστρεφείς γλώσσες όπως η C++ και η Java: στη Haskell οι κλάσεις είναι συλλογές τύπων, ενώ στις C++/Java είναι τύποι (και άρα συλλογές αντικειμένων).

- Η διαφορά αυτή έχει ως συνέπεια η Haskell να μην υποστηρίζει ετερογενείς δομές με την ευκολία των C++/Java.
- Η Haskell είναι όμως καλύτερη στη διαχείριση των "δυαδικών μεθόδων" από αυτές τις γλώσσες. Αυτό σημαίνει για παράδειγμα ότι μπορεί να επιβάλλει τον περιορισμό ότι ένας τελεστής μπορεί να εφαρμοστεί μόνο σε δύο (ή περισσότερα) δεδομένα ιδίου τύπου, κάτι που δε γίνεται στις C++/Java.

Αναφορές

[BCC+95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221--242, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.