1 Αναδρομή

Στη Haskell επιτρέπεται κατά τον καθορισμό της τιμής μίας συνάρτησης να επικαλεστούμε την ίδια τη συνάρτηση. Αυτό ονομάζεται αναδρομή και η συνάρτηση αναδρομική συνάρτηση.

Για να είναι καλά ορισμένη μία αναδρομική συνάρτηση (δηλαδή για να επιστρέφει αποτέλεσμα για όλες τις τιμές που επιτρέπεται να λάβουν τα ορίσματά της) θα πρέπει:

- να υπάρχουν κάποιες τιμές των ορισμάτων της για τις οποίες το αποτέλεσμα υπολογίζεται από μία εξίσωση που δεν χρησιμοποιεί αναδρομή.
- για τις υπόλοιπες τιμές των ορισμάτων, οι αλυσιδωτές χρήσεις αναδρομικών εξισώσεων να έχουν πεπερασμένο μήκους.

Στη Hakell η αναδρομή χρησιμοποιείται για να πραγματοποιήσουμε έναν υπολογισμό που απαιτεί επανάληψη.

Στα παρακάτω παραδείγματα χρησιμοποιούμε αναδρομή για να υλοποιήσουμε συναρτήσεις, οι περισσότερες από τις οποίες ορίζονται για φυσικούς αριθμούς (μη αρνητικούς ακεραίους) ή γενικότερα για ένα υποσύνολο των ακεραίων. Για απλούστευση, κατά τον ορισμό των συναρτήσεων Haskell που υλοποιουν αυτές τις μαθηματικές συναρτήσεις, υποθέτουμε ότι οι παράμετροι τους έχουν τιμές που ικανοποιούν τους απαραίτητους περιορισμούς. Αργότερα θα δούμε πώς μπορούμε να χειριστούμε τις υπόλοιπες περιπτώσεις.

Η πιό απλή περίπτωση αναδρομικής συνάρτησης μπορεί να οριστεί πάνω σε φυσικούς αριθμούς με τον παρακάτω τρόπο:

- η τιμή της συνάρτησης για το 0 ορίζεται χωρίς αναδρομή.
- η τιμή της συνάρτησης για $n > 0$ ορίζεται χρησιμοποιώντας την τιμή της συνάρτησης για το $(n-1)$.

Υπάρχουν όμως και άλλες μορφές αναδρομικών συναρτήσεων, ορισμένες από τις οποίες εμφανίζονται στα επόμενα παραδείγματα.

Παράδειγμα 18: παραγοντικό.

Το παραγοντικό ορίζεται για $n \geq 0$ από τον παρακάτω αναδρομικό τύπο:

$$n! = \begin{cases} 1 & \text{αν } n = 0 \\ n \cdot (n-1)! & \text{αν } n > 0 \end{cases}$$

Ο παραπάνω ορισμός μεταφράζεται άμεσα σε Haskell:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Η Haskell αποτιμάει το `fact 4` με τον παρακάτω τρόπο:

```
  fact 4
     #    4 <<< 0
      <=> "no"
= 4 * (fact (4-1))
     #    4-1 <<< 0
      <=> 3 <<< 0
      <=> "no"
= 4 * (3 * (fact (3-1)))
     #    3-1 <<< 0
      <=> 2 <<< 0
      <=> "no"
= 4 * (3 * (2 * (fact (2-1))))
     #    2-1 <<< 0
      <=> 1 <<< 0
      <=> "no"
= 4 * (3 * (2 * (1 * (fact (1-1)))))
     #    1-1 <<< 0
      <=> 0 <<< 0
      <=> "yes"
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

∎

Παράδειγμα 19: υπολογισμός του αθροίσματος $\sum_{i=1}^{n} i^i$.

Σε μία προστακτική γλώσσα (C, Pascal) θα υπολογίζαμε το παραπάνω άθροισμα με μία εντολή for. Στη Haskell πρέπει να χρησιμοποιήσουμε αναδρομή.

Παρατηρούμε ότι το παραπάνω άθροισμα μπορεί να οριστεί χρησιμοποιώντας αναδρομή:

$$\sum_{i=1}^{n} i^i = \begin{cases} 0 & n = 0 \\ (\sum_{i=1}^{n-1} i^i) + n^n & \text{αν } n > 0 \end{cases}$$

και υπολογίζεται από την παρακάτω συνάρτηση Haskell:

```
sum1 :: Int -> Int
sum1 0 = 0
sum1 n = sum1 (n-1) + n^n
```

∎

Η αναδρομική ισότητα δεν είναι απαραίτητο να επικαλείται την τιμή της συνάρτησης γαι το $(n-1)$. Επίσης ενδέχεται μία μη αναδρομική εξίσωση στον ορισμό της συνάρτησης να ορίζει το αποτέλεσμα όταν η παράμετρος έχει τιμή διαφορετική του 0.

Παράδειγμα 20: άθροισμα ψηφίων φυσικού αριθμού.

Μπορούμε να χρησιμοποιήσουμε τον τελεστή `‘mod‘` για να απομονώσουμε του τελευταίο ψηφίο του αριθμού και τον `‘div‘` για να διαγράψουμε το ψηφιο αυτό από τον αριθμό:

```
sumOfDigits :: Int -> Int
sumOfDigits 0 = 0
sumOfDigits n =
      sumOfDigits (n ‘div‘ 10) + n ‘mod‘ 10
```

Η παραπάνω συνάρτηση μπορεί να τροποποιηθεί έτσι ώστε για όλους τους μονοψήφιους αριθμούς το αποτέλεσμα να ορίζεται χωρίς αναδρομή. Στην περιπτωση αυτή η συνάρτηση γράφεται πιο κομψα χρησιμοποιώντας συνθήκες φρουρούς.

```
sumOfDigits’ :: Int -> Int
sumOfDigits’ n
     | n <= 9
          = n
     | otherwise
          = sumOfDigits’ (n ‘div‘ 10) + n ‘mod‘ 10
```

∎

Για συναρτήσεις που έχουν περισσότερες από μία παραμέτρους, υπάρχουν πολλές εναλλακτικές επιλογές για το ποια τιμή της συνάρτηση θα χρησιμοποιεί η αναδρομική ισότητα.

Παράδειγμα 21: υπολογισμός δύναμης.

Η παρακάτω συνάρτηση υπολογίζει (όχι με ιδιαίτερα αποδοτικό τρόπο) το n^k, για $k \geq 0$.

```
powerSlow :: Int -> Int -> Int
powerSlow n 0 = 1
powerSlow n k = n * powerSlow n (k-1)
```

Για την αποτίμηση του το n^k με χρήση της `powerSlow` απαιτείται ένα πλήθος βημάτων που είναι ανάλογο του k.

Μια καλύτερη υλοποίηση στηρίζεται στην παρακάτω παρατήρηση:

- αν $k = 2\lambda$ τότε $n^k = n^{2\lambda} = (n^\lambda)^2$
- αν $k = 2\lambda + 1$ τότε $n^k = n^{2\lambda+1} = (n^\lambda)^2 \cdot n$.

```
power :: Int -> Int -> Int
power n 0 = 1
power n k
    | k `mod` 2 == 0
        = p*p
    | otherwise
        = p*p*n
    where p = power n (k `div` 2)
```

Για την αποτίμηση του το n^k με χρήση της `power` απαιτείται ένα πλήθος βημάτων που είναι ανάλογο του λογάριθμου του k.

Για παράδειγμα, η αποτίμηση της παράστασης `power 2 9` γίνεται με τον παρακάτω τρόπο:

```
  power 2 9
     #    9 <<< 0
      <=> "no"
     ?  (9 `mod` 2) == 0
      = 1 == 0
      = False
     ?  otherwise
      = True
= (p*p)*2 where p = power 2 (9 `div` 2)
      { power 2 (9 `div` 2)
           #    9 `div` 2 <<< 0
            <=> 4 <<< 0
            <=> "no"
           ?  (4 `mod` 2) == 0
            = 0 == 0
            = True
       = p*p where p = power 2 (4 `div` 2)
           { power 2 (4 `div` 2)
                #    4 `div` 2 <<< 0
                 <=> 2 <<< 0
                 <=> "no"
                ?  (2 `mod` 2) == 0
                 = 0 == 0
                 = True
            = p*p where p = power 2 (2 `div` 2)
                { power 2 (2 `div` 2)
```

```
                    #    2 `div` 2 <<< 0
                     <=> 1 <<< 0
                     <=> "no"
                    ?  (1 `mod` 2) == 0
                     = 1 == 0
                     = False
                    ?  otherwise
                     = True
                  = (p*p)*2 where p = power 2 (1 `div` 2)
                       { power 2 (1 `div` 2)
                            #    1 `div` 2 <<< 0
                             <=> 0 <<< 0
                             <=> "yes"
                        = 1
                       }
                  = (1*1)*2
                  = 1*2
                  = 2
                 }
           = 2*2
           = 4
          }
       = 4*4
       = 16
      }
= (16*16)*2
= 256*2
= 512
```

∎

Παράδειγμα 22: υπολογισμός συνδυασμών.

Έστω m και n φυσικοί αριθμοί με $n \geq m$. Συμβολίζουμε με $\binom{n}{m}$ τους συνδυασμούς των n ανά m δηλαδή το πλήθος των διαφορετικών υποσυνόλων με m στοιχεία ένός συνόλου με n στοιχεία.

Είναι γνωστό ότι ισχύει ο παρακάτω αναδρομικό τύπος:

$$\binom{n}{m} = \begin{cases} 1 & m = 0,\ n \geq 0 \\ \frac{n \cdot \binom{n-1}{m-1}}{m} & \text{αν } n \geq m \geq 1 \end{cases}$$

Ο παραπάνω τύπος κωδικοποιείται άμεσα σε Haskell:

```
comb :: Int -> Int -> Int
comb n 0  = 1
comb n m  =  comb (n-1) (m-1) * n `div` m
```

∎

Παράδειγμα 23: υπολογισμός του αθροίσματος $\sum_{i=m}^{n} i^i$, για $m \leq n$.

Υπολογίζεται από την παρακάτω συνάρτηση Haskell:

```
sum2 :: Int -> Int -> Int
sum2 m n
        | m>n
          = 0
        | otherwise
          = m^m + sum2 (m+1) n
```

Ο υπολογισμός του `sum2 m n` με βάση την αναδρομική ισότητα, απαιτεί τον υπολογισμό του `sum2 (m+1) n`. Συνεπώς κατά την αποτίμη, οι τιμές των παραμέτρων της `sum2` ποτέ δεν μειώνονται. Ωστόσο, η αποτίμηση ολοκληρώνεται πάντα σε πεπερασμένο αριθμό βημάτων, καθώς η διαφορά των τιμών των παραμέτρων της `sum2` μειώνεται μετά από κάθε εφαρμογή της αναδρομικής εξίσωσης και όταν αυτή γίνει αρνητική (`m>n`) η αναδρομή σταματάει. ∎

Σε ορισμένες περιπτώσεις ενδέχεται να μην είναι τελείως προφανής ο τρόπος με τον οποίο μία συνάρτηση θα οριστεί αναδρομικά.

Παράδειγμα 24: υπολογισμός του μέγιστου κοινού διαιρέτη δύο θετικών αριθμών.

Δίνουμε πρώτα μία λιγότερη καλή λύση, η τεχνική της οποίας όμως είναι γενικότερα εφαρμόσιμη: δεδομένων δύο θετικών αριθμών m και n με $m \leq n$, γνωρίζουμε ότι ο μέγιστος κοινός διαιρέτης είναι ένας αριθμός μεταξύ 1 και m (αφού θα πρέπει να διαιρεί ακριβώς τον m).

Μπορούμε να βρούμε τον αριθμό εξετάζοντας όλους τους αριθμούς από το m μέχρι το 1, μέχρι να βρούμε κάποιον που να διαιρεί ταυτόχρονα τον m και τον n.

Η αναζήτηση γίνεται με τη βοήθεια της αναδρομικής συνάρτηση `seekGCD`, η οποία με είσοδο τους θετικούς m, n, k επιστρέφει τον μεγαλύτερο αριθμό μεταξύ του 1 και του k που διαιρεί τους m και n.

```
gcdSlow :: Int -> Int -> Int
gcdSlow m n
           | m <= n
              = seekGCD m n m
           | otherwise
              = seekGCD n m n
seekGCD :: Int -> Int -> Int -> Int
seekGCD m n k
           | m `mod` k == 0 && n `mod` k == 0
              = k
           | otherwise
              = seekGCD m n (k-1)
```

Η seekGCD επιστρέφει αποτέλεσμα για οποιαδήποτε τριάδα θετικών τιμών m, n, k: στη χειρότερη περίπτωση μετά από $k-1$ εφαρμογές της αναδρομικής ισότητας το τρίτο όρισμα της συνάρτηση θα είναι 1 και πρώτη συνθήκη-φρουρός θα ικανοποιηθεί.

Μία καλύτερη λύση στηρίζεται στον αλγόριθμο του Ευκλείδη, η ορθότητα του οποίου εξασφαλίζεται από το παρακάτω μαθηματικό θεώρημα: αν m, n είναι θετικοί αριθμοί με $m < n$ τότε το σύνολο των κοινών διαιρετών των m και n ισούται με το σύνολο των κοινών διαιρετών των m και $n - m$.

```
gcdEuc :: Int -> Int -> Int
gcdEuc m n
    | m==n
       = n
    | m<n
       = gcdEuc m (n-m)
    | otherwise
       = gcdEuc n (m-n)
```

Η τιμή της συνάρτησης για δύο θετικούς ακέραιους m και n διαφορετικούς μεταξύ τους ορίζεται χρησιμοποιώντας την τιμή της συνάρτησης για δύο επίσης θετικούς ακεραίους (ενδεχομένως ίσους) που έχουν άθροισμα μικρότερο του $m+n$. Επειδή το άθροισμα δύο θετικών ακεραίων δεν μπορεί να είναι μικρότερο του 2, συμπεραίνουμε ότι η αναδρομή πάντοτε σταματάει. Για παράδειγμα, η Haskell αποτιμάει το gcdEuc 64 24 με τον παρακάτω τρόπο:

```
  gcdEuc 64 24
     ?  64 == 24
      = False
     ?  64 > 24
      = True
= gcdEuc 24 (64-24)
     ?  24 == (64-24)
      = 24 == 40
      = False
```

```
     ?  24 > 40
      = False
     ?  otherwise
      = True
= gcdEuc 24 (40-24)
     ?  24 == (40-24)
      = 24 == 16
      = False
     ?  24 > 16
      = True
= gcdEuc 16 (24-16)
     ?  16 == (24-16)
      = 16 == 8
      = False
     ?  16 > 8
      = True
= gcdEuc 8 (16-8)
     ?  8 == (16-8)
      = 8 == 8
      = True
= 8
```

Μπορούμε να τροποποιήσουμε την παραπάνω συνάρτηση έτσι ώστε η τιμή της να ορίζεται αναδρομικά ακόμη και όταν τα ορίσματα της έχουν ίσες τιμές. Σε αυτή την περίπτωση η αναδρομή σταματάει όταν το δεύτερο όρισμα γίνει 0:

```
gcdEuc' :: Int -> Int -> Int
gcdEuc' m 0 = m
gcdEuc' m n
    | m<n
        = gcdEuc' m (n-m)
    | otherwise
        = gcdEuc' n (m-n)
```

Προσοχή: Η παράσταση `gcdEuc' 0 0` αποτιμάται σε 0, το οποίο δεν είναι σωστό αποτέλεσμα. Ωστόσο, ο σκοπός μας είναι `gcdEuc'` να δουλεύει σωστά μόνο για θετικές τιμές των παραμέτρων. Η πρώτη ισότητα είναι τεχνητή ώστε να δουλεύει σωστά η αναδρομή.

Ο μέγιστος κοινός διαιρέτης μπορεί να υπολογιστεί με πιο αποδοτικό τρόπο με βάση τις παρακάτω παρατηρήσεις.

Ας υποθέσουμε ότι $n > m$ και ότι η διαίρεση n δια m δίνει πηλίκο q και υπόλοιπο r. Συνεπώς $n = q \cdot m + r$. Ισχύει $gcd(m, n) = gcd(m, q \cdot m + r)$. Εφαρμόζοντας το θεώρημα q φορές έχουμε:

$$\begin{aligned} gcd(m, q \cdot m + r) &= gcd(m, (q-1) \cdot m + r) \\ &= \ldots \\ &= gcd(m, m + r) \\ &= gcd(m, r) \end{aligned}$$

Κάθε μία από τις παραπάνω q ισότητες αντιστοιχεί και σε μία εφαρμογή της αναδρομικής ισότητας στον ορισμό της `gcdEuc'`. Από τις παραπάνω ισότητες προκύπτει ότι $gcd(m, n) = gcd(m, n \bmod m)$. Η παρακάτω συνάρτηση `gcdFast` παρακάμπτει όλες τις ενδιάμεσες εφαρμογές της αναδρομικής ισότητας, αντικαθιστώντας απ' ευθείας το n με το $n \bmod m$:

```
gcdFast :: Int -> Int -> Int
gcdFast m 0 = m
gcdFast m n
    | m < n
        = gcdFast m (n `mod` m)
    | otherwise
        = gcdFast n (m `mod` n)
```

∎

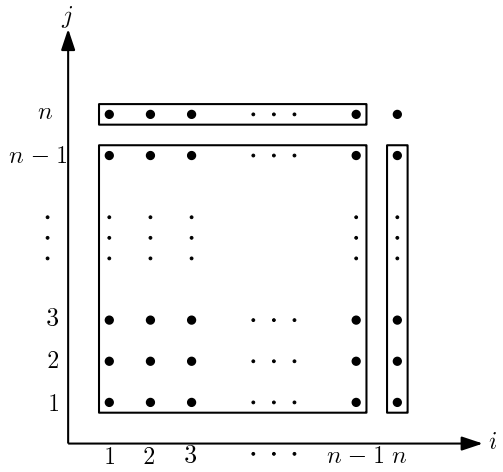
Παράδειγμα 25: υπολογισμός του διπλού αθροίσματος $\sum_{i=1}^{n} \sum_{j=1}^{n} i^j$.

Γενικά για να υπολογίσουμε ένα άθροισμα χρησιμοποιώντας αναδρομή προσπαθούμε να χωρίσουμε τους όρους που πρέπει να προστεθούν σε ξένα υποσύνολα, καποια από τα οποία θα σχηματίζουν αθροίσματα τις ίδιας μορφής με λιγότερους προσθετέους τα οποία μπορούν να υπολογιστούν με αναδρομή, ενώ τα υπόλοιπα θα σχηματίζουν απλούστερης μορφής αθροίσματα (π.χ. απλά αθροίσματα) ή θα είναι μεμονωμένοι όροι.

Παρατηρούμε ότι το παραπάνω άθροισμα ικανοποιεί την αναδρομική ισότητα:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} i^j = \begin{cases} 0 & n = 0 \\ (\sum_{i=1}^{n-1} \sum_{j=1}^{n-1} i^j) + \sum_{i=1}^{n-1} i^n + \sum_{j=1}^{n-1} n^j + n^n & \text{αν } n > 0 \end{cases}$$

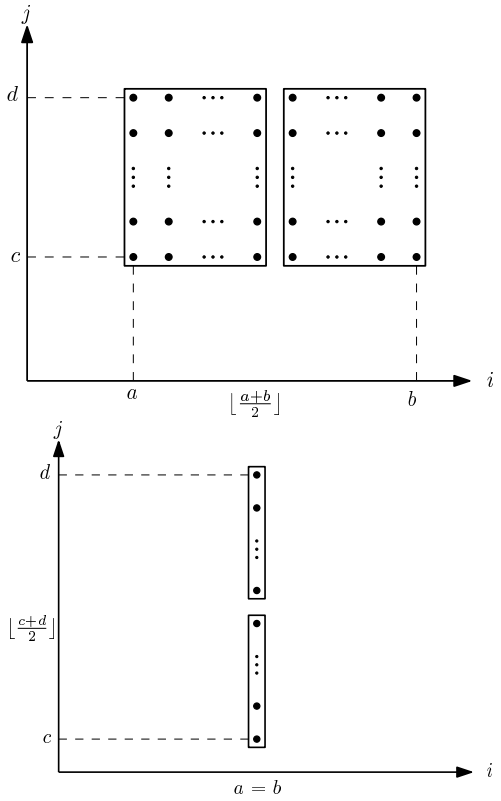
Σχηματικά αυτό παριστάνεται με τον παρακάτω τρόπο:



Μπορούμε γράψουμε μια συνάρτηση Haskell για υπολογισμό του αθροίσματος με βάση τον παραπάνω τύπο, γράφοντας και δύο βοηθητικές συναρτήσεις για τον υπολογισμό των απλών αθροισμάτων. Θα αναβάλουμε για αργότερα την περιγραφή μίας τέτοιας συνάρτησης.

Σε ορισμένες περιπτώσεις είναι πιο βολικό να υλοποιήσουμε μία πιο γενική συνάρτηση από αυτή που μας ενδιαφέρει. Στη συγκεκριμένη περίπτωση θα σχεδιάσουμε μία συνάρτηση για υπολογισμό του το άθροίσματος $\sum_{i=a}^{b} \sum_{j=c}^{d} i^j$ με $a \leq b$ και $c \leq d$.

Στην παρακάτω συνάρτηση `sumabcd`, αν το πλήθος των προσθετέων είναι μεγαλύτερο του ένα, το αποτέλεσμα ορίζεται αναδρομικά με διαχωρισμό των τιμών για τον έναν από του δύο δείκτες σε δύο περίπου ίσα διαστήματα. Σχηματικά:




```
sumabcd :: Int -> Int -> Int -> Int -> Int
sumabcd a b c d =
    if a==b then
       if c==d then a^c
               else sumabcd a b c n
                  + sumabcd a b (n + 1) d
       else sumabcd a m c d
          + sumabcd (m + 1) b c d
    where m = (a + b) `div` 2
          n = (c + d) `div` 2
```

Μπορούμε να αποφύγουμε τελείως τη διαίρεση, χωρίζοντας διαφορετικά τις τιμές των δεικτών:

```
sumabcd' :: Int -> Int -> Int -> Int -> Int
sumabcd' a b c d =
    if a==b then
       if c==d then a^c
               else sumabcd' a b c c
                  + sumabcd' a b (c + 1) d
            else sumabcd' a a c d
               + sumabcd' (a + 1) b c d
```

Η `sumabcd'` δεν χρησιμοποιεί διαίρεση, ωστόσο σχηματίζει μεγαλύτερες αναδρομικές αλυσίδες.

Το ζητούμενο άθροισμα $\sum_{i=1}^{n}\sum_{j=1}^{n} i^j$ μπορεί να υπολογιστεί από την παρακάτω συνάρτηση:

```
doubleSum0n :: Int -> Int
doubleSum0n n = doubleSumabcd 1 n 1 n
```

∎

Σε ορισμένες περιπτώσεις, η απευθείας χρήση ενός αναδρομικού τύπου μπορεί να επιβαρύνει τον χρόνο υπολογισμού. Αυτό συμβαίνει όταν ο ίδιος υπολογισμός επαναλαμβάνεται πολλές φορές.

Παράδειγμα 26: αριθμοί Fibonacci. Οι αριθμοί Fibonacci ορίζονται από τον παρακάτω αναδρομικό τύπο:

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-2} + f_{n-1}, \text{ για } n \geq 2 \end{aligned}$$

Ο παραπάνω αναδρομικός τύπος μεταφράζεται άμεσα σε Haskell:

```
fibSlow :: Int -> Int
fibSlow 0 = 1
fibSlow 1 = 1
fibSlow n = fibSlow (n-2) + fibSlow (n-1)
```

Η παραπάνω υλοποίηση αποτελεί παράδειγμα κακής χρήσης της αναδρομής, καθώς οι ίδιες τιμές υπολογίζονται πάρα πολλές φορές. Για να γίνει αυτό εμφανές δίνουμε την αποτίμηση του `fibslow 5`:

```
fibslow 5
   #    5 <<< 0
    <=> "no"
   #    5 <<< 1
```

```
      <=> "no"
= (fibslow (5-2)) + (fibslow (5-1))
     #    (5-2) <<< 0
      <=> 3 <<< 0
      <=> "no"
     #    3 <<< 1
      <=> "no"
= ((fibslow (3-2)) + (fibslow (3-1))) + (fibslow (5-1))
     #    (3-2) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= (1 + (fibslow (3-1))) + (fibslow (5-1))
     #    (3-1) <<< 0
      <=> 2 <<< 0
      <=> "no"
     #    2 <<< 1
      <=> "no"
= (1 + ((fibslow (2-2)) + (fibslow (2-1)))) + (fibslow (5-1))
     #    (2-2) <<< 0
      <=> 0 <<< 0
      <=> "yes"
= (1 + (1 + (fibslow (2-1)))) + (fibslow (5-1))
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= (1 + (1 + 1)) + (fibslow (5-1))
= (1 + 2) + (fibslow (5-1))
= 3 + (fibslow (5-1))
     #    (5-1) <<< 0
      <=> 4 <<< 0
      <=> "no"
     #    4 <<< 1
      <=> "no"
= 3 + ((fibslow (4-2)) + (fibslow (4-1)))
     #    (4-2) <<< 0
      <=> 2 <<< 0
      <=> "no"
     #    2 <<< 1
      <=> "no"
= 3 + (((fibslow (2-2)) + (fibslow (2-1))) + (fibslow (4-1)))
     #    (2-2) <<< 0
      <=> 0 <<< 0
```

```
      <=> "yes"
= 3 + ((1 + (fibslow (2-1))) + (fibslow (4-1)))
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + ((1 + 1) + (fibslow (4-1)))
= 3 + (2 + (fibslow (4-1)))
     #    (4-1) <<< 0
      <=> 3 <<< 0
      <=> "no"
     #    3 <<< 1
      <=> "no"
= 3 + (2 + ((fibslow (3-2)) + (fibslow (3-1))))
     #    (3-2) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + (2 + (1 + (fibslow (3-1))))
     #    (3-1) <<< 0
      <=> 2 <<< 0
      <=> "no"
     #    2 <<< 1
      <=> "no"
= 3 + (2 + (1 + ((fibslow (2-2)) + (fibslow (2-1)))))
     #    (2-2) <<< 0
      <=> 0 <<< 0
      <=> "yes"
= 3 + (2 + (1 + (1 + (fibslow (2-1)))))
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + (2 + (1 + (1 + 1)))
= 3 + (2 + (1 + 2))
= 3 + (2 + 3)
= 3 + 5
= 8
```

Παρατηρούμε ότι το f_3 υπολογίζεται 2 φορές, το f_2 υπολογίζεται 3 φορές, και το το f_1 υπολογίζεται 5 φορές. Γενικότερα, για να υπολογιστεί το f_n, υπολογίζεται αναδρομικά f_i φορές το f_{n-i}, για όλες τις τιμές του i από 1 έως n. Η συνάρτηση f_n αυξάνει όμως πάρα πολύ γρήγορα σε σχέση με το n και αυτό έχει ως αποτέλεσμα η παραπάνω υλοποίηση να μην είναι αποδοτική.

Η παρακάτω υλοποίηση της συνάρτησης, υπολογίζει αποδοτικά το f_n κάνοντας διαδοχικές αθροίσεις, από "κάτω προς τα πάνω":

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fibhlp (n-1) 1 1
    where fibhlp :: Int -> Int -> Int -> Int
          fibhlp 1 a b = a + b
          fibhlp k a b = fibhlp (k-1) b (a+b)
```

Ο υπολογισμός του f_5 αποτιμώντας την παράσταση `fib 5` είναι σαφώς γρηγορότερος:

```
  fib 5
     #    5 <<< 0
      <=> "no"
     #    5 <<< 1
      <=> "no"
= fibhlp (5-1) 1 1
     #    (5-1) <<< 1
      <=> 4 <<< 1
      <=> "no"
= fibhlp (4-1) 1 (1+1)
     #    (4-1) <<< 1
      <=> 3 <<< 1
      <=> "no"
= fibhlp (3-1) (1+1) (1+(1+1))
     #    (3-1) <<< 1
      <=> 2 <<< 1
      <=> "no"
= fibhlp (2-1) (1+(1+1)) ((1+1)+(1+(1+1)))
     #    (2-1) <<< 1
      <=> 1 <<< 1
      <=> "yes"
= (1+(1+1)) + ((1+1)+(1+(1+1)))
= (1+2) + (2+(1+2))
= 3 + (2+3)
= 3 + 5
= 8
```

Στον παραπάνω υπολογισμό, η αποτίμηση του ορίσματος `1+1` γίνεται μία μόνο φορά, όταν διαπιστωθεί ότι αυτό είναι απαραίτητο για τον υπολογισμό του αποτελέσματος και είναι ορατή σε όλα τα τμήματα της παράστασης που το χρησιμοποιούν.

Παρόμοια, το όρισμα `(1+(1+1))` υπολογίζεται μία μόνο φορά, αφού ενδιάμεσα πάρει τη μορφή `(1+2)`. ∎

Στο παρακάτω παράδειγμα αυτό θα φανεί η σημασία των τοπικών ορισμών, σε περίπτωση που μία τιμή που υπολογίζεται αναδρομικά χρησιμοποιείται σε περισσότερα από ένα σημεία του ορισμού της συνάρτησης.

Παράδειγμα 27: ακέραιο μέρος της τετραγωνικής ρίζας ενός αριθμού.

Χρησιμοποιούμε την παρακάτω ιδιότητα:

- αν ο αριθμός n είναι τέλειο τετράγωνο, τότε $\lfloor\sqrt{n}\rfloor = \sqrt{n} = \lfloor\sqrt{n-1}\rfloor+1$. Σε αυτή την περίπτωση ισχύει $n = (\lfloor\sqrt{n-1}\rfloor + 1)^2$.
- αν ο n δεν είναι τέλειο τετράγωνο, τότε $\lfloor\sqrt{n}\rfloor = \lfloor\sqrt{n-1}\rfloor$ και $n < (\lfloor\sqrt{n-1}\rfloor + 1)^2$

Η παρακάτω συνάρτηση δεν χρησιμοποιεί τοπικούς ορισμούς:

```
sqrtIntSlow :: Int -> Int
sqrtIntSlow 0 = 0
sqrtIntSlow n
     | (sqrtIntSlow (n-1) + 1)^2 == n
         = sqrtIntSlow (n-1) + 1
     | otherwise
         = sqrtIntSlow (n-1)
```

Στην παραπάνω υλοποίηση γίνεται κακή χρήση της αναδρομής: κατά τον υπολογισμό του $\lfloor\sqrt{n}\rfloor$ το $\lfloor\sqrt{(n-i)}\rfloor$ υπολογίζεται 2^i φορές, συνεπώς το συνολικό πλήθος βημάτων είναι εκθετικό ως προς το n.

Η παρακάτω υλοποίηση για να υπολογίσει το $\lfloor\sqrt{n}\rfloor$ υπολογίζει μόνο μία φορά το $\lfloor\sqrt{(n-1)}\rfloor$ και του δίνει τοπικά όνομα με χρήση του where. Με αυτό τον τρόπο το πλήθος των βημάτων γίνεται αναλογο του n.

```
sqrtInt :: Int -> Int
sqrtInt 0 = 0
sqrtInt n
     | (m + 1)^2 == n
         = m + 1
     | otherwise
         = m
     where m = sqrtInt (n-1)
```

Μία ακόμη πιο αποδοτική υλοποίηση στηρίζεται στην δυαδική αναζήτηση της σωστής τιμής στο διάστημα 0 έως n, στην οποία το πλήθος των βημάτων είναι λογαριθμικό ως προς το n.

Ορίζουμε τοπικά μία συνάρτηση `sqrthlp` η οποία παίρνει ως είσοδο ένα διάστημα μέσα στο οποίο βρίσκεται η τετραγωνική ρίζα. Μέσα στην `sqrthlp` ορίζουμε τοπικά το `c`, που είναι το μέσο του διαστήματος. Κάθε εφαρμογή της αναδρομικής ισότητας μειώνει το μήκος του διαστήματος στο οποίο βρίσκεται η τετραγωνική ρίζα στο μισό. Όταν το μήκος αυτό γίνει ένα, η τετραγωνική ρίζα έχει εντοπιστεί και η αναδρομή σταματάει.

```
sqrtIntFast :: Int -> Int
sqrtIntFast n = sqrthlp 0 n
         where sqrthlp :: Int -> Int -> Int
               sqrthlp a b
                   | a==b
                         = a
                   | c*c > n
                         = sqrthlp a (c-1)
                   | otherwise
                          = sqrthlp c b
                     where c = (a+b+1) `div` 2
```

Το `sqrtIntFast 40` αποτιμάται με τον παρακάτω τρόπο:

```
  sqrtIntFast 40
= sqrthlp 0 40
     ?  0 == 40
      = False
     ?  (c*c) > 40 where c = ((0+40)+1) `div` 2
           {  ((0+40)+1) `div` 2
            = (40+1) `div` 2
            = 41 `div` 2
            = 20
           }
      = (20*20) > 40
      = 400 > 40
      = True
= sqrthlp 0 (20-1)
     ?  0 == (20-1)
        0 == 19
      = False
     ?  (c*c) > 40 where c = ((0+19)+1) `div` 2
           {  ((0+19)+1) `div` 2
            = (19+1) `div` 2
            = 20 `div` 2
            = 10
           }
      = (10*10) > 40
      = 100 > 40
      = True
```

```
= sqrthlp 0 (10-1)
     ?  0 == (10-1)
        0 == 9
      = False
     ?  (c*c) > 40 where c = ((0+9)+1) `div` 2
           {  ((0+9)+1) `div` 2
            = (9+1) `div` 2
            = 10 `div` 2
            = 5
           }
      = (5*5) > 40
      = 25 > 40
      = False
     ?  otherwise
      = True
= sqrthlp 5 9
     ?  5 == 9
      = False
     ?  (c*c) > 40 where c = ((5+9)+1) `div` 2
           {  ((5+9)+1) `div` 2
            = (14+1) `div` 2
            = 15 `div` 2
            = 7
           }
      = (7*7) > 40
      = 49 > 40
      = True
= sqrthlp 5 (7-1)
     ?  5 == (7-1)
        5 == 6
      = False
     ?  (c*c) > 40 where c = ((5+6)+1) `div` 2
           {  ((5+6)+1) `div` 2
            = (11+1) `div` 2
            = 12 `div` 2
            = 6
           }
      = (6*6) > 40
      = 36 > 40
      = False
     ?  otherwise
      = True
= sqrthlp 6 6
     ?  6 == 6
      = True
= 6
```

2 Χειρισμός μη αποδεκτών είσόδων

Οι ορισμοί των συναρτήσεων που είδαμε μέχρι τώρα, υποθέτουν ότι οι τιμές των ορισμάτων τους ικανοποιούν κάποιες συνθήκες: Συγκεκριμένα:

- οι ορισμοί των `fact`, `sum1`, `somOfDigits`, `fib`, `sqrtInt` υποθέτουν ότι τα ορίσματα τους είναι θετικοί αριθμοί ή μηδέν.
- οι ορισμοί των συναρτήσεων για υπολογισμό του μέγιστου κοινού διαιρέτη υποθέτουν ότι τα ορίσματα τους είναι θετικοί μηδέν.
- ο ορισμός της `comb` υποθέτει επιπλέον ότι $n \geq m$.
- αντίστοιχες ανισότητες πρέπει να ικανοποιούν και οι τιμές των παραμέτρων των `sum2` και `doublesum`, οι οποίες όμως επιστρέφουν σωστό αποτέλεσμα και για αρνητικές τιμές των ορισμάτων τους.

Ενδέχεται το πρόγραμμα που χρησιμοποιεί τις συναρτήσεις αυτές να εξασφαλίζει ότι πάντοτε αποτιμούνται με ορίσματα που οι τιμές τους ικανοποιούν τους απαραίτητους περιορισμούς. Αν αυτό δε συμβαίνει τότε θα πρέπει να επεκτείνουμε τις συναρτήσεις μας ώστε να πραγματοποιούν τους απαραίτητους ελέγχους έτσι ώστε το πρόγραμμα να έχει πάντα προβλέψιμη συμπεριφορά.

Τα παρακάτω παραδείγματα δίνουν μερικές ιδέες για το πώς μπορούμε να χειριστούμε τις προβληματικές εισόδους.

Παράδειγμα 28: παραγοντικό.

Αν ζητήσουμε από το διρμηνέα της Haskell να αποτιμήσει τη συνάρτηση `fact` με αρνητικό όρισμα, τότε θα προκληθεί υπερχείλιση στοίβας. Μπορούμε τροποποιήσουμε την `fact` έτσι ώστε όταν το όρισμα της έχει αρνητική τιμή, να επιστρέφει ένα μήνυμα λάθους:

```
fact' :: Int -> Int
fact' n
     | n<0
        = error "fact': negative argument"
     | n=0
        = 1
     | otherwise
        = n * fact' (n-1)
```

Η συνάρτηση `error` προκαλεί άμεση διακοπή της εκτέλεσης του προγράμματος και εμφανίζει ενα μήνυμα λάθους.

Το μειονέκτημα της παραπάνω υλοποίησης είναι ότι ο έλεγχος για αρνητικό όρισμα γίνεται $(n + 1)$ φορές κατά τον υπολογισμό του $n!$ για $n \geq 0$. Μπορούμε να το αποφύγουμε τους πολλαπλούς ελέγχους σχεδιάζοντας μια συνάρτηση που θα κάνει τον απαραίτητο έλεγχο μία μόνο φορά και θα επιστρέφει

αποτέλεσμα χρησιμοποιώντας την `fact` αν το όρισμα της έχει μη αρνητική τιμή. Με αυτό τον τρόπο διαχωρίζουμε τον έλεγχο της εισόδου από την αναδρομή και μπορούμε να επιλέξουμε ποια συνάρτηση θα καλέσουμε ανάλογα με το αν θέλουμε να γίνει έλεγχος της εισόδου ή όχι.

```
factGen :: Int -> Int
factGen n
     | n<0
        = error "factGen: negative argument"
     | otherwise
        = fact n
```

Αν δεν θέλουμε να τερματίστει η αποτίμηση με μήνυμα λάθους, μπορούμε να επιλέξουμε μία τιμή η οποία δεν ανήκει στο πεδίο τιμών της συνάρτησης (π.χ. 0) και να την επιστρέφουμε, κωδικοποιώντας έτσι το λάθος:

```
factGen' :: Int -> Int
factGen' n
     | n<0
        = 0
     | otherwise
        = fact n
```

Μπορούμε να επιστρέφουμε πιο άμεσα την ένδειξη για λάθος τροποποιώντας τον τύπο του αποτελέσματος της συνάρτησης:

```
factGen'' :: Int -> (Bool,Int)
factGen'' n
     | n<0
        = (False,0)
     | otherwise
        = (True,fact n)
```

∎

Παράδειγμα 29: υπολογισμός συνδυασμών.

Το $\binom{n}{m}$ μπορεί να οριστεί ακόμη και όταν $m > n$: επειδή σε αυτή την περίπτωση ένα σύνολο με n στοιχεία δεν μπορεί να έχει υποσύνολο με m στοιχεία ισχύει $\binom{n}{m} = 0$. Αντίθετα, αν κάποιο από τα n και m είναι αρνητικός αριθμός, επειδή αυτά παριστάνουν πληθάριθμους συνόλων, το $\binom{n}{m}$ δεν μπορεί να οριστεί με προφανή τρόπο.

Με βάση τις παρατηρήσεις αυτές, μπορούμε να γράψουμε την παρακάτω γενίκευση της `comb`:

```
combGen :: Int -> Int -> Int
combGen n m
      | n<0 || m<0
         = error "combGen: negative argument"
      | n<m
         = 0
      | otherwise
         = comb n m
```

Θα μπορούσαμε να "χρύψουμε" την comb στο εσωτερικό της combGen:

```
combGen' :: Int -> Int -> Int
combGen' n m
     | n<0 || m<0
        = error "combGen': negative argument"
     | n<m
        = 0
     | otherwise
        = comb n m
     where comb :: Int -> Int -> Int
           comb n 0  = 1
           comb n m  =  comb (n-1) (m-1) * n `div` m
```