Why Functional Programming Matters

http://www.math.chalmers.se/~rjmh/Papers/whyfp.html

Yazılımlar karmaşıklaştıkça, yapı giderek daha çok önem kazanıyor. İyi yapılandırılmış bir yazılımın yazılması, debug edilmesi ve gelecekteki değişikliklere uyum sağlaması daha kolay.

Fonksiyonel programlama bu şekilde adlandırılmıştır. Çünkü program tümüyle fonksiyonlardan oluşur.

Ana fonksiyon, alt bazı fonksiyonlardan oluşur. Bunlar daha alt fonksiyonlardan oluşur. Ta ki primitiflere kadar gider. Fonksiyonlar, sıradan matematiksel fonksiyonlar gibidir. Burada bunlara sıradan denklemler denilecek.

Fonksiyonel programlarda atama (assignment) cümleleri yoktur. Değişkenler de yoktur. Bir değer veridi mi hiç değişmez. Dolayısıyla, fonksiyonel programların yan etkileri yoktur. Fonksiyon kendi değerini hesaplamak dışında bir etkiye sahip olamaz.

Bu yüzden programların çağrılma zamanı önemsizdir.

Fonksiyonel programlar çok verimlidir. Çünkü çok kısadırlar. Niçin? Çünkü alışılmış programların %90´ı atama işlemlerinden oluşur, değil. Daha mantıklı sebepleri olmalı.

Yapısal Programlamayla Bir Benzerlik

Eskiden yapısal programlamanın faydaları şu şekilde sunuluyordu: Yapısal programlar, goto ifadesi içermez. Blokların birden çok giriş ve çıkışı yoktur. Bu faydaların anlatım şekli fonksiyonel programlamanın bir önceki kısımda anlatım şekline benziyor. Her ikisi de olumsuz cümleler.

Yapısal programlamanın en büyük faydası, modüler tasarımdı. Küçük modüller hızlı bir şekilde programlanabilir. Genel modüller yeniden kullanılabilir. Ve modüller birbirinden bağımsız geliştirilebilir.

Gotoların olmayışı, bundan farklı bir şey.

Ancak önemli bir nokta gözden kaçırılıyor. Modüler programlamayla bir problemi çözmek, bir yaklaşım gerektiriyor. Önce büyük problemi alt problemlere bölüyoruz. Sonra onu daha alt problemlere bölüyoruz. Sonra bunların aralarındaki bağımlılık ilişkilerini gözeterek, birbirlerine yapıştırıyoruz. Halbuki yapısal programlama dillerinin araçları, bu noktada önemli bir katkı sağlamıyor. Önemli olan tasarım şekli.

Yapıştırabilmenin önemini göstermek için, marangozluğu düşünelim. Bir sandalyeyi, iki türlü yapabilirsiniz. Ayaklarını, oturağını, desteğini ayrı ayrı imal edip, sonra bunları bağlayıp yapıştırarak. Veya büyük bir tahta kütlesini oyarak.

Modüler bir şekilde yapmanın kolaylığı apaçık görünüyor burada.

Bu makalenin anafikri şu: Fonksiyonel diller, çok önemli iki yapışkan türü sağlıyor. Fonksiyonel programlamanın esas gücünü sağlayan da bu iki araç.

Fonksiyonları birbirine yapıştırmak

Fonksiyonları birbiren yapıştırmak:

Bir örnek üzerinden inceleyelim. Basit bir liste işleme problemini ele alalım. Listenin elemanlarının birbirine

eklenebilmesini istiyoruz. Listeleri şu şekilde tanımlıyoruz:

```
listof X ::= nil | cons X (listof X)
```

Yukarıdaki ifadenin anlamı şu: Xlerden oluşan bir liste (X herhangi bir şey olabilir, mesela integer gibi), ya nil ´dir (yani boştur), ya da Xin bir cons´u ve Xlerin başka bir listesidir. Cons, ilk elemanı X olan, ikince ve sonraki elemnları Xlerin başka listelerinin elemanları olan bir listedir.

```
[] means nil
[1] means cons 1 nil
[1,2,3] means cons 1 (cons 2 (cons 3 nil))
```

Bir listenin elemanlarının toplamı, recursive `sum` fonksiyonuyla tanımlanır. Sum, iki tip argüman için tanımlanmalıdır: Boş liste (nil) ve bir cons.

```
sum nil = 0

sum (cons num list) = num + sum list
```

Bu tanımı incelediğimizde, sadece iki yerin sum fonksiyonuna has olduğunu görürüz:

```
          +---+
sum nil = | 0 |
          +---+
                            +---+
sum (cons num list) = num | + | sum list
                            +---+
```

Dolayısıyla, sum fonksiyonu, iki kısma bölünebilir:

1. Genel bir recursion kalıbı
2. Kutulu kısımlar

Bu recursion kalıbına, indirgemek (`reduce`) denir.

`(reduce f a)` ifadesinin anlamı şu: Listedeki `cons` terimlerinin her birini `f` ile değiştiren, `nil` terimlerinin her birini de `a` ile değiştiren fonksiyon.

```
sum = reduce add 0

add x y = x + y

(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Yukarıda reduce metoduna add metodunu ve 0 değerini argüman olarak yolluyoruz. reduce metodu, kendi içinde recursion algoritmasının mantığını, kendisine gönderilen argümanlara uyguluyor.

Benzer şekilde aşağıda bir listedeki değerlerin çarpımı tanımlanıyor:

```
product = reduce multiply 1
```

Listedeki boolean değerlerinin tümünün doğru veya en azından birinin doğruluğunu onaylayan metotlar:

```
anytrue = reduce or false
```

```
alltrue = reduce and true
```

`list [1,2,3]` ifadesini ele alalım:

Bunun açılımı:

```
cons 1 (cons 2 (cons 3 nil))
```

`(reduce add 0)` bunu şu hale dönüştürür:

```
add 1 (add 2 (add 3 0)) = 6
```

`(reduce multiply 1)` şuna dönüştürür:

```
multiply 1 (multiply 2 (multiply 3 1)) = 6
```

Bir listeyi aynen kopyalamak için:

```
reduce cons nil
```

İki listeyi birbiri ardısıra eklemek için:

```
append a b = reduce cons b a

append [1,2] [3,4]      = reduce cons [3,4] [1,2]
                        = (reduce cons [3,4]) (cons 1 (cons 2 nil))
                        = cons 1 (cons 2 [3,4]))
                              (replacing cons by cons and nil by [3,4])
                        = [1,2,3,4]
```

`(reduce cons [3,4]) (cons 1 (cons 2 nil))` Bu adımda dikkat ederseniz, `reduce` metodu üç parametre alıyor. Aslında zaten tüm yukarıdaki örneklerde de reduce üç parametre alıyordu.

Mesela, `sum [1,3,5]  --» (reduce add 0) [1,3,5]`

Bir listedeki değerlerin tümünü ikiyle çarpan bir fonksiyon:

```
doubleall = reduce doubleandcons nil
doubleandcons num list = cons (2*num) list
```

Örnek:

```
doubleall [2,3]   = (doubleandcons 2 (doubleandcons 3) )
                  = [4, 6]
```

Doubleandcons daha ileri seviyede modülerleştirilebilir:

```
doubleandcons = fandcons double
double n = 2*n

fandcons f el list = cons (f el) list
fandcons f = cons . f
```

. fonksiyon birleştirme operatörü şu şekilde tanımlanır:

```
(f . g) h = f (g h)
```

Bu durumda fandcons şu hale gelir:

```
fandcons f el     = (cons . f) el
                  = cons (f el)
```

Öyleyse

```
fandcons f el list = cons (f el) list
```

Son olarak:

```
doubleall = reduce (cons . double) nil
```

Buradan bir modülarizasyon daha yapabiliriz:

```
doubleall = map double
map f = reduce (cons . f) nil
```

Bir matrisin bütün elemanlarını toplayarak yeni bir matris oluşturabiliriz:

```
summatrix = sum .map sum
```

Örnek:

```
list = [1 2 3 4]

summatrix list    = sum . map sum list
            = sum (map sum) list
            = sum (cons . sum 1 (cons . sum 2(cons . sum 3(cons . sum 4))))
            = sum [10 9 7 4]
            = 30
```

Pardon böyle değilmiş. `list` farklı bir şekilde belirtiliyormuş:

```
list = [ [1 2] [3 4] ]

summatrix list = sum . map sum list
            = sum (map sum) list
            = sum [ 3 7 ]
            = 10
```

Modülarizasyonun ne kadar faydalı olduğunu yukarıdaki örnekler gösteriyor. Listelerdeki fonksiyonların dışında, mesela ordered labelled treelerde de fonksiyonlardan yararlanabiliriz:

```
treeof X ::= node X (listof (treeof X))

            1 o
             / \
            /   \
           /     \
          2 o    o 3
                  |
                  |
                  |
                 o 4
```

Yukarıdaki ağaç şu şekilde ifade edilir:

```
node 1
      (cons (node 2 nil)
            (cons (node 3
                  (cons (node 4 nil) nil))
                        nil))
```

reduce´un benzeri bir metot tanımlarız: redtree. Ağaçlar node, cons ve nil´den oluştuğu için, redrtee üç argüman almalı.

```
redtree f g a (node label subtrees) =
f label (redtree' f g a subtrees)
redtree' f g a (cons subtree rest) =
g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

Yukarıdaki tanımı ben de anlayamadım. Ama aşağıdaki örnekler anlaşılıyor:

Bir ağaçtaki bütün elemanların toplamını bulmak için:

```
sumtree = redtree add add 0
```

Örnek:

```
add 1
      (add (add 2 0)
            (add (add 3
                  (add (add 4 0) 0))
            0))
= 10
```

Ağaçtaki tüm etiketlerin listesini oluşturmak:

```
labels = redtree cons append nil
```

Örnek:

```
cons 1
      (append (cons 2 nil)
            (append (cons 3
                  (append (cons 4 nil) nil))
            nil))
= [1,2,3,4]
```

Ağaçtaki bütün değerlere bir fonksiyon uygulamak (map)

```
maptree f = redtree (node . f) cons nil
```

Bütün bunların yapılmasını sağlayan, fonksiyonların fonksiyonların birleşimiyle tanımlanabilmesidir. Yeni bir veri tipi tanımlandığında, onu işlemek için gerekli üst seviye fonksiyonlar da tanımlanmalıdır.

Programları birbirine yapıştırmak

Programları birbirine yapıştırmak

Fonksiyonel dillerdeki diğer yapıştırıcı, bütün olarak programların birbirine yapıştırılmasını sağlar. Bir fonksiyonel program, girdisinden çıktısına kadar tek bir fonksiyondur. Eğer f ve g böyle iki programsa, (g . f) inputuna uygulandığında şu sonucu verir:

g (f input)

f´in çıktısı, g´nin girdisi olur. Normalde bunu sağlamak için, geçici bir dosya oluşturulurdu. Ancak bu performans açısından ciddi maliyet oluşturur. Fonksiyonel diller farklı bir çözüm uygular. f ve g birlikte olarak, senkronize bir şekilde çalışır. f yalnızca eğer ki g bir girdi okumaya çalışırsa çalışır. Ve sadece g´yi girdisini verinceye kadar çalışır. Ne zaman ki, g daha fazla veri istemez, f otomatik olarak sonlanmış olur. Bu yüzden f hiç sona ermeyen bir program bile olabilir. Bu şekilde, sonlandırma şartlarını döngü kodundan ayırmak mümkün olur. Bu da modülarizasyonun önemli bir özelliğidir.

Bu mekanizmaya tembel işleme (lazy evaluation) denir. Bu yolla mesela üretici (generator) bir program yapılabilir. Bu program çok sayıda olası çözümü oluşturur. Bir başka program da seçici olur. Uygun bir tanesini seçer. Bu yolla modülarizasyon bir derece daha artırılır. Bu özellik, modülarizasyonun en güçlü aracıdır.

Newton-Raphson Kare Kökü

Newton-Raphson Kare Kökü

Numerik bir algoritmada tembel işlemenin gücünü görelim.

Bu algoritma bir N sayısının kare kökünü bulur. En başta a0 diye bir ilk yaklaşık değer bulur. Daha sonra şu kuralı uygulayarak, yaklaşık değeri giderek iyileştirir:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

Nihai olarak bulunan a, karekök değeridir.

Kare kök programları belirli bir tolerans değerini (eps) kabul eder. Eğer arka arkaya üretilmiş iki yaklaşık değer eps´den daha az bir farka sahipse, algoritma durur.

Bu algoritma genellikle şu şekilde programlanır:

```
C N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
      X = A0
      Y = A0 + 2.*EPS
C THE VALUE OF Y DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
      100 IF (ABS(X-Y).LE.EPS) GOTO 200
            Y = X
            X = (X + ZN/X) / 2.
      GOTO 100
      200 CONTINUE
C THE SQUARE ROOT OF ZN IS NOW IN X
```

Bu program, geleneksel dillerde daha fazla bölünemez. Tembel işlemeyi kullanarak, bunu daha modüler bir forma sokacağız.

```
next N x = (x + N/x) / 2
```

Dolayısıyla (next N) bir yaklaşık değeri bir sonrakine bağlayan (map) fonksiyondur. f fonksiyonunu çağırmak şu yaklaşık değerler dizisini oluşturur:

```
[a0, f a0, f(f a0), f(f(f a0)), ..]
```

Bunu diziyi oluşturması için bir fonksiyon tanımlayabiliriz:

```
repeat f a = cons a (repeat f (f a))
```

Örneğimizde şu argümanları yollarız:

```
repeat (next N) a0
```

repeat metodu, sonsuz bir çıktı üretir.

Programın kalan kısmı, sonlandırma koşulunu tanımlar: within. Bir tolerans değerini ve yaklaşık değerler dizisini argüman olarak alır.

```
within eps (cons a (cons b rest)) =
= b, if abs(a-b) <= eps
= within eps (cons b rest), otherwise
```

Parçaları birleştirdiğimizde:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

Şimdi kare kök bulucunun iki parçasına da sahip olduğumuza göre, farklı şekillerde kombinasyonları da deneyebiliriz. Bir değişiklik, ardışık yaklaşık değerlerin arasındaki oranın bire yaklaşmasını beklemektir. Çok küçük sayılar için bu daha doğru olur. Çünkü bu durumda fark zaten toleransın altına en baştan düşerdi. Yine çok büyük sayılar için de daha iyi olur, çünkü bu durumda yuvarlama hatası, toleransın kendisinden büyük olur.

Yapmamız gereken, within yerine başka bir metot tanımlamak:

```
relative eps (cons a (cons b rest)) =
= b, if abs(a-b) <= eps*abs b
= relative eps (cons b rest), otherwise
```

Şimdi kare kök bulucuyu şu şekilde tanımlarız:

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

Yaklaşık değerleri bulan kısmı, yeniden yazmak gerekmez.

Bütün bu kolaylıkların dışında, `within, relative` programlarını başka algoritmalarda da kullanabiliriz.

Yapay Zekadan Bir Örnek

Yapay Zekadan Bir Örnek

Bu örnek biraz kapsamlı.

Alpha-beta heuristicini inceleyeceğiz. Bu algoritma, bir oyuncunun pozisyonunun ne kadar iyi olduğunu tahmin eden bir algoritma. Algoritma, oynun nasıl seyredeceğini inceleyerek çalışıyor. Ancak faydasız hamleleri eliyor.

Oyun pozisyonları, position tipindeki nesnelerle gösterilsin. Bu tip oyundan oyuna değişebilir. Bir pozisyondan hangi hareketlerin yapılacağını bilmenin bir yolu olmalı. Bunu yapan fonksiyon:

moves: position -> listof position

ile tanımlansın. Argüman olarak bir pozisyonu alıyor ve müsait pozisyonların listesini çıktı olarak veriyor. XOX oynunda örnek bir çıktı şöyle olabilir:

file:///C:\Documents%20and%20Settings\mnuhoglu\My%20Documents\Notlarim\Resimler\141.gif

moves fonksiyonunun verili olması durumunda, ilk yapılacak adım oyun ağacını inşa etmektir. Bu ağaçta, düğümler, pozisyonlarla etiketlenir. Öyle ki, düğümün çocukları, tek hamlede bu düğümden erişilebilecek pozisyonlarla etiketlenmiştir. Dolayısıyla, eğer bir düğümün etiketi p pozisyonu ise, çocuklarının etiketleri (moves p) listesidir. Bir oyun ağacı sonsuza kadar uzayabilir, eğer her iki tarafın da hiçbir zaman kazanmaması mümkünse. Oyun ağaçları, daha önce tanımladığımız ağaçlar gibidir. Aynı veri tipini burada da kullanabiliriz.

Oyun ağacı, moves metodunun arka arkaya uygulanmasıyla inşa edilir. Kök pozisyonundan başlayarak, moves kökün alt ağaçlarının etiketlerini oluşturmak için kullanılır. Daha sonra alt ağaçların alt ağaçları oluşturulur ve böyle devam eder. Bu yineleme (recursion) kalıbı şu fonksiyonla ifade edilir:

reptree f a = node a (map (reptree f) (f a))

Bu fonksiyonu kullanarak başka bir fonksiyon daha tanımlarız. Bu fonksiyon da belirli bir pozisyondan başlayarak oyun ağacını inşa eder:

gametree p = reptree moves p

Örnek olarak:

file:///C:\Documents%20and%20Settings\mnuhoglu\My%20Documents\Notlarim\Resimler\142.gif

Bu algoritma, belirli bir pozisyondan başlayarak, oyunun iyi yönde mi kötü yönde mi gelişeceğini öngörür. Ancak bunu yapabilmesi için bir pozisyonun değerini kabaca tahmin edebilmeli. Bu statik değerlendirme ileri bakışın limitinde kullanılmalı. Ve daha önceki algoritmaya yol göstermeli.

Şöyle bir fonksiyonumuz, pozisyonun statik değerini ölçmeli:

static: position -> number

Bir oyun ağacı bir (treeof position)´dır. Dolayısıyla (maptree static) fonksiyonuyla bir (treeof number)´a dönüştürülebilir.

Böyle bir statik değerler ağacı verildiğinde, içindeki pozisyonların gerçek değerleri nedir? Özellikle, kök pozisyonuna hangi değer atanmalı? Statik değeri değil, çünkü bu sadece kaba bir tahmindir. Bir düğüme atanan değer, onun alt düğümlerinin gerçek değerlerinden hesaplanmalı. Bu ancak, bütün oyuncuların en iyi hamleyi yapacaklarının varsayılmasıyla gerçekleştirilir. Yüksek değer, bilgisayar için iyi bir pozisyon demektir. Herhangi bir pozisyondan bilgisayarın hamle sırası geldiğinde, en yüksek gerçek değere sahip alt düğüme hamle yapacaktır. Benzer şekilde, rakip de en düşük gerçek değere sahip alt düğüme hamle yapacaktır. Bilgisayarın ve rakibinin sıralarının arka arkaya değiştiğini varsayarsak, düğümün gerçek değeri maximise fonksiyonuyla

hesaplanır, bilgisayarın sırası geldiğinde, tersi durumda minimise fonksiyonuyla hesaplanır.

maximise (node n sub) = max (map minimise sub)
minimise (node n sub) = min (map maximise sub)

Not: Niçin max içinde minimise metodunu çağırıyoruz? Çünkü yukarıda dediğimiz gibi, rakibin yapabileceği en iyi hamleyi, bizim açımızdan en düşük değerli hareket olarak tanımladığımız için.

Bu tanımlar sonsuza kadar giderler. Base case yoktur. Takipçisi olmayan bir düğümün değerini tanımlamalıyız. Ve onu düğümün statik değeri olarak almalıyız.

Bu aşamada bir pozisyonu alıp onun gerçek değerini döndüren bir fonksiyon yazabiliriz:

evaluate = maximise . maptree static . gametree

Bu tanımda iki sorun vardır: Birincisi, sonsuz ağaçlar için çalışmaz. Eğer son yoksa, maximise hiçbir zaman değer döndürmez. İkinci sorun da bununla ilişkilidir. Sonlu oyun ağaçları bile, çok büyük olabilir. Bütün ouyn ağacının tümünün değerini bulmak, gerçekdışıdır. Arama belirli bazı hamlelerle sınırlandırılmalı. Bunun için ağaç, belirli bir derinlikten budanır.

prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1)) x)

(prune n) bir ağacı alır ve n. alt ağaçtan itibaren olan düğümleri keser.

Bu durumda, evaluate metodu:

evaluate = maximise . maptree static . prune 5 . gametree

Bu 5 hamle ötesini görür.

Burada üst seviye fonksiyonları ve tembel işlemeyi kullanıyoruz. gametree´nin sonsuz bir potansiyeli vardır. Tembel değerlendirme olmasaydı, program asla sona ermezdi.

prune 5 . gametree

yazmak yerine, her iki fonksiyonu birleştirip, beş seviyelik bir ağaç oluşturan tek bir fonksiyon yazmak gerekirdi.

Buraya kadar minimaxing algoritmasını tarif ettik. Alpha beta algoritması, maximise ve minimise metotlarını kullanmadan bütün ağacın değerini tahmin edebilir.

file:///C:\Documents%20and%20Settings\mnuhoglu\My%20Documents\Notlarim\Resimler\143.gif

Yukarıdaki ağacı değerlendirebilmek için, soru işareti ? bulunan yeri bilmek şart değil. Sol minimum 1 eder, sağ minimum 0 veya daha düşüktür. Bu yüzden her ikisinin maksimumu 1 olmalıdır. Bu gözlem, maximise ve minimise algoritmalarının içine genelleştirilebilir.

maximise = max . maximise'

maximise' (node n nil) = cons n nil
maximise' (node n l) = map minimise l
= map (min . minimise') l

= map min (map minimise' l)
= mapmin (map minimise' l)
where mapmin = map min

Yeni bir mapmin metodu yazalım. Bu minimumları önemli olmayan listeleri elesin:

mapmin (cons nums rest) =
= cons (min nums) (omit (min nums) rest)

omit fonksiyonuna potansiyel bir maximum girilir - yani şu ana kadar görülmüş en büyük minimum. Bundan daha düşük olan minimumları eler.

omit pot nil = nil
omit pot (cons nums rest) =
= omit pot rest, if minleq nums pot
= cons (min nums) (omit (min nums) rest), otherwise

minleq şu şekilde tanımlanır:

minleq nil pot = false
minleq (cons num rest) pot = true, if num<=pot
= minleq rest pot, otherwise

Şimdi evaluate metodunu yeniden tanımlayalım:

evaluate = max . maximise' . maptree static . prune 8 . gametree

Bir iyileştirme de en başta en yüksek değerli düğümlerin değerlendirilmesiyle yapılır:

highfirst (node n sub) = node n (sort higher (map lowfirst sub))
lowfirst (node n sub) = node n (sort (not.higher) (map highfirst sub))
higher (node n1 sub1) (node n2 sub2) = n1>n2

evaluate = max . maximise' . highfirst . maptree static .
prune 8 . gametree