Högre ordningens funktioner

Programmering II - Elixir

Kenan Dizdarevic

6 Februari 2023

Inledning

Funktioner av högre ordning är ett centralt koncept inom funktionell programmering. En funktion av högre ordning gör det möjligt för oss att skapa allmän kod, det hälper oss att undvika repetition av kod. Detta är möjligt eftersom funktionerna anpassas för att lösa specifika uppgifter. I denna undersökning kommer vi att implementera två kraftfulla konstruktioner: `map` och `reduce`.

En funktion av högre ordning inom programmering är en funktion som tar andra funktioner som argument och/eller returnerar en funktion. Detta innebär att vi kan anpassa beteendet hos en annan funktion och manipulera data enklare. Vi kan även använda oss av *dynamisk tidsplanering*, vi kan planera exekveringen av andra funktioner.

Undersökning

Transformera en lista rekursivt

Vi skall nu transformera olika listor rekursivt. Vi skall implementera tre olika funktioner: `double/1`, `five/1` och `animal/1`. De skall utföra olika operationer på en lista. `double/1` skall multiplicera alla tal i listan med 2, `five/1` skall addera 5 till alla tal i listan och `animal/1` skall byta ut alla förekomster av `:dog` mot `:fido`. Implementeringen av dessa funktioner presenteras nedan:

```
def double([]) do [] end
def double([head | tail]) do [head * 2 | double(tail)] end

def five([]) do [] end
def five([head | tail]) do [head + 5 | five(tail)] end

def animal([]) do [] end
def animal([head | tail]) do
```

```
    if head == :dog do
      [:fido | animal(tail)]
    else
      [head | animal(tail)]
    end
end
```

De är inte särskilt komplicerade att implementera. De har samma basfall och operationerna som de utför är nästan identiska, varje funktion manipulerar listan på sitt egna vis.

Istället för att behöva implementera tre identiska funktioner kan vi kombinera de i funktionen `double_five_animal/2`. Denna funktion tar emot en lista och en atom som specificerar hur listan skall manipuleras. Lösningen presenteras i koden nedan:

```
def double_five_animal([], _) do [] end
def double_five_animal([head | tail], f) do
    case f do
      :double ->
        [head * 2 | double_five_animal(tail, f)]
      :five ->
        [head + 5 | double_five_animal(tail, f)]
      :animal ->
        case head == :dog do
          true ->
            [:fido | double_five_animal(tail, f)]
          false ->
            [head | double_five_animal(tail, f)]
        end
    end
  end
```

Vi har kvar samma basfall som i de föregående funktionerna. Det vi istället gör är att vi tar emot en till parameter, en atom som specificerar vilken av operationerna som vi skall utföra. I denna implementering har vi ingen onödig repetition som vi hade tidigare. Nu är vi ett steg närmare en funktion av högre ordningen.

Funktioner som argument

Variabler kan bindas till funktioner. Detta innebär att vi kan behandla funktioner som data. En funktion i Elixir representeras som:

```
f = fn(x) -> x * 2 end
```

I detta fall representerar variabeln f en funktion som tar emot en variabel x och multiplicerar den med 2.

Om vi kan definiera funktioner som variabler och behandla dem som data så kan vi även använda de som argument i andra funktioner.

Vi skall konstruera funktionen `apply_to_all/2` som omvandlar alla element i en lista genom att applicera funktionen på alla element i listan. Koden för fuktionen presenteras i koden nedan:

```
def apply_to_all([head | tail], f) do
    [f.(head) | apply_to_all(tail, f)]
end
```

Basfallet för funktoinen är enkelt, om vi får in en tom lista returnerar vi en tom lista. I det rekursiva fallet delar vi upp listan i det första elementet och resten av listan. Sedan utför vi funktionen på första elementet och anropar `apply_to_all/2` rekursivt på resterande del av listan. Värt att notera är att man måste binda funktionen till en variabel innan man använder den i funktionen. Annars beter sig inte funktionen korrekt.

Vi har implementerat Elixirs egna funktion `map/2` som är otroligt kraftfull. Vi kan enkelt manipulera stora mängder av data med vår funktion.

Reducera en lista

Högerriktad "fold"

Vi kan implementera funktioner som summerar en lista och multiplicerar alla tal i listan. Detta kräver att vi implementerar två funktioner som kommer vara identiska. Den enda skillnaden kommer vara den aritmetiska operatorn, vi vill undvika onödiga upprepningar. Därför skall vi implementera funktionen `fold_right/3` som tar emot en lista, ett grundvärde och en funktion som argument. Funktionen skall returnera grundvärdet för en tom lista och i övriga fall skall den returnera värdet som erhålls när vi applicerar funktionen på alla element. Funktionens basfall kommer endast returnera ett grundvärde. När vi utför multiplikation vill vi ha grundvärdet 1 och när vi adderar vill vi ha grundvärdet 1. Den rekursiva delen av funktionen presenteras i koden nedan:

```
def fold_right([head | tail], base, f) do
    f.(head, fold_right(tail, base, f))
end
```

Vi ser genast att funktionen är en högerriktad "fold", eftersom den börjar med det sista elementet i listan. Funktionen $f(x, y)$ tar emot två argument för att sammanfoga listan. Det rekursiva anropet sker före sammanfogningen av elementen vilket innebär att om vi utför en addition med listan [1, 2, 3, 4] ser det ut som följande: $(1 + (2 + (3 + (4 + 0))))$. Denna funktion är inte svansrekursiv, eftersom vi går igenom hela listan tills vi kommer till det sista elementet där ifrån börjar vi sammanfoga elementen.

Vänsterriktad ”fold”

Vi skall nu implementera en svansrekursiv funktion som sammanfogar elementen direkt, en vänsterriktad ”fold”. Funktionen `fold_left/3` tar emot en lista, en ackumulator och en funktion. Funktionen skall returnera ackumulatorn i basfallet. I det andra fallet skall den returnera samma sammanfogning som den högerriktade ”folden”. Detta är koden för implementeringen:

```
def fold_left([head | tail], acc, f) do
    fold_left(tail, f.(head, acc), f)
end
```

Vi anropar `fold_left/3` rekursivt direkt, men det vi gör är att vi delar upp listan i det första elementet och resterande del av listan. När vi gör det rekursiva anropet är vår nya lista den resterande delen av listan, vi sammanfogar det första elementet med ackumulatorn och funktionen. Ackumulatorn är det som möjliggör att funktionen är svansrekursiv, vi sammanfogar alla element samtidigt som vi förflyttar oss i listan. De rekursiva anropen kommer att upphöra när vi når basfallet, det enda som returneras är ackumulatorn.

Dessa två funktioner som vi har implementerat finns i Elixirs modul `List`, de heter `foldr/3` och `foldl/3`. Det finns även en implementering av den vänsterriktade ”folden” i modulen `Enum`, den heter `reduce/3`. Detta förstärker vårt antagande om att dessa funktioner är viktiga och kraftfulla.

Filter

Den tredje funktionen av högre ordning som vi skall studera är filter. Denna konstruktion har förmågan att filtrera bort element i en lista som uppfyller vissa krav. Vi har implementerat funktionen `odd/1` som tar emot en lista som argument och returnerar en lista av alla element som är udda. Detta är inte smidigt och det kommer leda till onödiga repetitioner av kod. Vi skall därför implementera funktionen `filter/2` som tar emot en lista och en funktion. Funktionen som tas som argument returnerar sant eller falskt för varje element i listan, den avgör om elementet skall filtreras eller ej. `filter/2` returnerar en lista med alla element som uppfyller kriteriet definierat av funktionen som tas som argument. Implementeringen ser ut som följande:

```
def filter([], _) do [] end
def filter([head | tail], f) do
    case f.(head) do
      true ->
        [head | filter(tail, f)]
      false ->
         filter(tail, f)
```

```
      end
end
```

Basfallet är trivialt, om vi tar in en tom lista returnerar vi en tom lista. I det andra fallet delar vi återigen upp listan i första elementet och resten av listan. Vi evaluerar funktionen som tar emot det första elementet ur vår ursprungliga lista. Om denna funktion returnerar sant så skall vi inte filtrera bort det specifika elementet. Vi anropar istället `filter/2` rekursivt på resterande del av listan. Om evalueringen returnerar falskt så skall vi ta bort det specifika elementet ur listan. Återigen anropar vi `filter/2` rekursivt på resterande del av listan.

Funktionen `odd/1` kan enkelt implementeras med hjälp av `filter/2`. Det enda vi behöver är en funktion som ställer det kraven som `odd/1` gör. Funktionen som gör detta är:

```
f = fn(x) -> rem(x, 2) == 1 end
```

Vi kan även filtrera bort alla element som är mindre än 5 med funktionen:

```
f = fn(x) -> x > 5 end
```

Denna funktion returnerar sant för alla x som är större än 5, vilket är det vi vill uppnå.

Slutsats

Slutligen ser vi att högre ordningens funktioner är ett mycket kraftfullt verktyg i programmering. Det gör det möjligt att skapa generella, flexibla och återanvändbara funktioner. Vi slipper även göra koduppprepningar vilket förbättrar läsbarheten för vår kod.

När vi har lärt oss att använda dessa funktioner korrekt kommer effektiviteten att öka avsevärt. Funktionerna var inte komplicerade att implementera men det krävdes eftertanke. Vi behöver även ha god förståelse för hur högre ordningens funktioner fungerar och är konstruerade.