Tågväxling

Programmering II - Elixir

Kenan Dizdarevic

27 februari 2023

Inledning

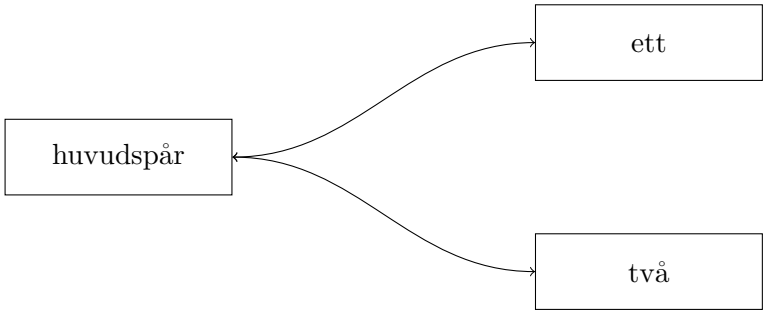
Vi ansvarar för att förflytta och ordna tågvagnar genom att använda de växlar och spår som finns tillgängliga. Vi antar att varje vagn är självkörande samt att tåget inte har något lok. Funktionerna tar emot två argument där det första argumentet är det givna tåget och det andra argumentet är vårt önskade tåg, tågen representeras som listor. Vi skall även skriva ut sekvensen för alla förflyttningar.

Till en början skall vi arbeta med bearbetningsfunktioner. Dessa funktioner skall användas för att exempelvis dela upp tåget på lämpligt vis. Det vi egentligen arbetar med är relativt enkla listoperationer. Sedan skall vi konstruera en funktion som tar en förflyttning och ett tillstånd som argument, den skall returnera det nya tillståndet där förflyttningen är genomförd. Slutligen skall vi lösa problemet med tågväxlingen och förflytta vagnarna i rätt ordning.

Implementering

Modellering

Vi behöver införa tydliga representationer för att ha en möjlighet att applicera en adekvat lösning. En enskild vagn kommer att representeras som en atom. Detta innebär att ett tåg kommer representeras som en lista av atomer. Vi antar att ett tåg inte har flera vagnar som representeras av samma atom. En fullständig beskrivning av tågets tillstånd består således av tre listor. En lista representerar det nuvarande tåget på huvudspåret, den andra listan beskriver tåget som finns på spår ”ett” och den sista listan beskriver tåget som finns på spår ”två”. Stationen för tågväxling konkretiseras i Figur 1.



Figur 1: Station för tågväxling

Vi behöver representera vagnarnas position på lämpligt vis. Vagnarna representeras på det sätt som vi representerar spåren. Detta innebär att den första vagnen på spår ett och två är den vagn som är närmast huvudspåret. Den första vagnen på huvudspåret är den vagn som är längst bort från spår ett och två

Förflyttningar representeras som binära tupler, första elementet är antingen `:one` eller `:two` det andra elementet är en integer som beskriver hur många vagnar som skall flyttas. Om integern är positiv skall de högra vagnarna flyttas till det önskade spåret. Om integern istället är negativ skall de vänstra vagnarna flyttas till det önskade spåret.

Bearbetning

Vi skall konstruera sju olika funktioner som ger oss möjlighet att utföra diverse operationer på tågen som representeras i form av listor. Funktionerna som vi skall implementera är följande:

I. `take(train, n)`: returnerar tåget som består av de `n` första vagnarna.

II. `drop(train, n)`: returnerar tåget utan de `n` första vagnarna.

III. `append(train1, train2)`: returnerar det sammansatta tåget.

IV. `member(train, wagon)`: kollar om vagnen är en del av tåget.

V. `position(train, wagon)`: returnerar första positionen av vagnen i tåget.

VI. `split(train, wagon)`: returnera en tupel med två tåg där `wagon` inte tillhör någon vagn av det delade tåget.

VII. `main(train n)`: returnerar tupeln {`k, remain, take`} där `remain` och `take` är tågets vagnar och `k` är återstående vagnar för att ha `n` vagnar i take.

Funktionen `split/2` implementeras på följande vis:

```
def split(train, wagon) do
    {take(train, position(train, wagon) - 1), drop(train, position(train, wagon))}
end
```

Vi returnerar en tupel där det första elementet består av de första vagnarna i tåget. De första vagnarna som vi skall behålla är de som är framför `wagon`. Det andra elementet i tupeln består av alla vagnar som kommer efter `wagon`. För att `split/2` skall fungera använder vi `take/2`, `position/2` och `drop/2`.

Funktionen `main/2` kan implementeras med hjälp av `split/2` vilket förenklar koden. Vi skall dock implementera `main/2` som en rekursiv funktion med hjälp av mönstermatchning. Funktionen implementeras på följande vis:

```
def main([head | tail], n) do
    case main(tail, n) do
      {0, remain, take} ->
        {0, [head | remain], take}
      {k, remain, take} ->
        {k - 1, remain, [head | take]}
    end
end
```

Vi använder oss av ett *case-uttryck* där vi rekursivt anropar `main/2`. Om funktionen returnerar $k = 0$ vilket innebär att vi inte behöver flytta fler vagnar för att ha n vagnar i take skall vi lägga till första vagnen i tåget till remain. Om ett annat värde på k returneras skall vi istället lägga till vagnen till take. Till detta har vi implementerat ytterligare två fall. Basfallet behandlar en tom lista, den returnerar {`n, [], []`}. Det andra fallet behandlar $n = 0$, vi returnerar {`0, train, []`}.

Förflyttningar

Vi skall nu implementera funktionerna `single/2` och `sequence/2`. Funktionen `single/2` tar emot en förflyttning och ett tillstånd. Funktionens syfte är att returnera det nya tillståndet där förflyttningen är genomförd. Vi skall med hjälp av mönstermatchning avgöra vilket spår som är involverat i förflyttningen. När vi förflyttar en vagn måste vi veta om vi flyttar den till eller från ett spår, detta avgörs med hjälp av tecknet på n. Om n är positivt skall vi förflytta en vagn till spåret, om n däremot är negativt skall vi förflytta en vagn från spåret. För att möjliggöra denna lösning kommer vi nyttja funktionen `main/2` som vi tidigare implementerade.

Funktionen `single/2` har fem olika fall. Basfallet behandlar fallet där $n = 0$, ingen förflyttning skall ske.

```
def single({_, 0}, {main, one, two}) do {main, one, two} end
```

Vi returnerar endast det nuvarande tillståndet.

Vi kan flytta vagnar till spår ett eller två. En förflyttning representeras som en tupel, `{:one, n}` vi skall flytta n vagnar till spår ett. De fall vi nu skall behandla är två fall för vardera spår, fallen är $n > 0$ och $n < 0$. Koden för att flytta n vagnar till och från spår ett presenteras nedan:

```
def single({:one, n}, {main, one, two}) when n > 0 do
    {0, remain, take} = Train.main(main, n)
    {remain, Train.append(take, one), two}
end
def single({:one, n}, {main, one, two}) when n < 0 do
    n = abs(n)
    {Train.append(main, Train.take(one, n)), Train.drop(one, n), two}
end
```

I det första fallet är $n > 0$, då skall vi flytta vagnar till spår ett. Vi utför en mönstermatchning där vi använder `main/2`, när det inte finns fler vagnar att flytta skall vi returnera tupeln som består av element. Första elementet är `remain` som vi erhåller från mönstermatchningen. Det andra elementet är en sammankoppling av de vagnar som vi skall flytta samt de vagnar som finns på spår ett. Det sista elementet är de vagnar på spår två, som vi inte har förflyttat.

Det andra fallet behandlar när vi skall flytta vagnar från spår ett till huvudspåret. Vi tar absolutbeloppet av n för att kunna utföra våra förflyttningar korrekt. Det första elementet i tupeln består av sammankopplingen mellan de vagnar som finns på huvudspåret och de vagnar som skall tas bort från spår ett. Det andra elementet består av de vagnar som vi skall ta bort från spår ett. Det sista elementet består av de vagnar på spår två, som är oförändrade.

Växlingsproblemet

Vi skall nu lösa växlingsproblemet, där vi hittar en sekvens av förflyttningar som ändrar tåget på önskat vis. Vi skall skapa en funktion `find/2` som har två tåg som argument, det nuvarande tåget och det önskade. Problemet skall lösas rekursivt och varje steg kommer att förflytta en vagn till sin korrekta plats. Basfallet behandlar ett tåg utan utan vagnar, vi gör inga förflyttningar. Funktionen som implementeras ser ut som följande:

```
def find(xs, [y | ys]) do
    {hs, ts} = Train.split(xs, y)
    [
      {:one, length(ts) + 1},
      {:two, length(hs)},
```

```
      {:one, -(length(ts) + 1)},
      {:two, -length(hs)} |
      find(Train.append(hs, ts), ys)
    ]
end
```

Vi använder vagnen längst till vänster i det önskade tåget för att dela upp vårt nuvarande tåg. Vi erhåller två listor, en lista med alla element före och en lista med alla element efter vagnen längst till vänster. Sedan skapar vi en lista av tupler där vi presenterar våra förflyttningar. Listan beskriver hur många vagnar vi har flyttat till respektive spår. Det första elementet i listan beskriver hur många vagnar som flyttats till spår ett, det andra elementet beskriver hur många vagnar som flyttats till spår två. De tredje och fjärde elementet beskriver hur många vagnar som flyttas från vardera spår.

Funktionen medför flertal redundanta förflyttningar. Därav skall vi implementera funktionen `few/2` som har samma beteende som `find/2` men som har i åtanke om en specifik vagn redan är på korrekt plats. Funktionen ser ut som följande:

```
def few([h | hs], [y | ys]) when h == y do
    few(hs, ys)
end
def few(hs, [y | ys]) do
    {hs, ts} = Train.split(hs, y)

    [
      {:one, length(ts) + 1},
      {:two, length(hs)},
      {:one, -(length(ts) + 1)},
      {:two, -length(hs)} |
      few(Train.append(hs, ts), ys)
    ]
end
```

Vi har utökat funktionen med ett fall där vagnen är på korrekt plats. Om detta stämmer skall vi låta vagnen vara kvar och utföra ett nytt rekursivt anrop. Resterande del av `few/2` är identisk med `find/2`.

Komprimering

Förflyttningarna som `few/2` genomför kan optimeras och förenklas. Detta skall vi göra med funktionen `compress/1` som är given samt regler som vi implementerar. Vi skall implementera reglerna i funktionen `rules/1` som tar en lista av förflyttningar som argument. Implementeringen av några regler presenteras nedan:

```
def rules([{:one, n}, {:one, m} | tail]) do rules([{:one, n + m} | tail]) end
def rules([{:two, n}, {:two, m} | tail]) do rules([{:two, n + m} | tail]) end
def rules([head | tail]) do [head | rules(tail)] end
```

Om vi har har förflyttat vagnar till spår ett och vi flyttar fler vagnar till spår ett kan vi skriva samman det. Detta gör vi i det översta fallet av `rules/1`. Vi skall utföra samma operation om vi gör en liknande förflyttning till spår två. Det sista fallet hanterar de situationer vi inte kan förenkla. Vi skall låta det första elementet vara kvar i listan och utför sedan ett rekursivt anrop på resterande del av listan.

Slutsats

Slutligen ser vi att problemet med tågväxling är relativt trivialt att lösa, om vi delar upp problemet i mindre beståndsdelar. Det viktiga är att vi har en adekvat beskrivning av tåget och dess vagnar.

Till en början implementerade vi olika operationer som vi kan utföra på ett tåg, exempelvis dela det på önskat vis.

Sedan implementerade vi funktioner som har möjlighet att uppdatera det nuvarande tillståndet baserat på en förflyttning.

Slutligen löste vi växelproblemet, där vi även optimerade den sista funktionen.