Lab 1-1: Train Shunting

Mattias Cederlund

01-11-2012

1 Uppgiften

Uppgiften går ut på att bygga en modell för att simulera växling av tågvagnar mellan ett huvudspår och två stickspår. Vi får en given sekvens av vagnar, samt önskad sekvens och ska hitta vilka vagnförflyttningar mellan de tre spåren som ska göras för att nå önskad ordning på vagnarna.

Modellen bygger på tåg, vagnar och dess tillstånd, fördelningen av vagnarna mellan de olika spåren. Vagnar är modellerade som atomer och tåg som listor av atomer. Tillstånden modelleras med en tuple med tre listor som symboliserar tåget på huvudspåret och tågen på de två stickspåren. En förflyttning av vagnarna modelleras med en tuple som innehåller en atom, one eller two som berättar mellan vilka spår förflyttningen ska ske, samt en integer som bestämmer hur många vagnar som ska förflyttas.

För att lösa våra problem med modelleringen behöver vi konstruera funktionerr som modifierar tåg och dess fördelningar över spåren, samt funktioner som bestämmer vilka förflyttningar som behöver göras för att gå från en vagnföljd till en annan. För att göra detta möjligt behöver vi ett antal stödfunktioner för att modifiera listor.

2 Ansatts

Uppgift 1 innebar att utvecka funktioner för att modifiera listor. Vi skapar följande funktioner.

Funktionen take(Xs,N) ska returnera en lista innehållande de första N elementen i listan Xs. Basfallen här var när listan var tom, eller vid N = 0. Då returneras tomma listan. Annars går man rekursivt genom listan och returnerar första elementet följt av anrop på take igen med resten av listan och N-1 som argument.

Funktionen drop(Xs,N) ska returnera listan Xs utan de N första elementet. Detta får vi genom följande basfall: När listan är tom returneras tomma listan. När N = 0 returneras Xs. Däremot om listan har minst ett element anropas drop igen på listan utan första elementet och N-1.

Funktionen append(Xs,Ys) slår ihop Xs och Ys till en gemensam lista. Det får vi genom basfallet att om Xs är tomma listan returneras Ys. Det induktiva fallet fås genom att dela upp Xs i head och tail, samt att returnera listan med första elementet i Xs följt av append på tail och listan Ys.

Funktionen member(Xs,Y) returnerar yes om elementet Y finns i listan Xs, annars no. Som basfall används att om Xs är tomma listan returneras no och om Y återfinns i första positionen i listan Xs returneras yes. Om detta inte uppfylls kallas member med listan Xs utan första elementet och Y som argument.

Funktionen position(Xs,Y) ska returnera på vilken position i listan Xs elementet Y finns på. Vi antar att elementet Y finns i listan för att det ska fungera. Basfallet är att när Y finns i första positionen i listan Xs returneras 1, annars returneras 1+ position(T,Y) där T är listan Xs utan det första elementet.

Uppgift 2 innebar att utveckla en funktion single(Move,State) som modellerar en förflyttning mellan två spår. Funktionen tar en förflyttning och ett tillstånd, ska utföra förflyttningen och sen returnera det nya tillståndet.

I min lösning matchar jag inkommande State med Main, One, Two. Därefter ställer jag upp fall där jag matchar Move med t.ex. one, N för förflyttningar mellan huvudspåret och första stickspåret. Totalt finns fyra fall med olika kombinationer av spår och positivt/negativt N.

Om N är större än 0, vilket kollas med when N ¿ 0 i samband med matchningen, ska vagnar förflyttas från huvudspåret till stickspåret. Är så fallet returneras en tuple innehållande ny state där tåget på huvudspåret ges av de vagnar som blir kvar efter att N vagnar förflyttats till stickspår ett. Det möjliggörs genom att använda funktionen take med det ursprungliga tåget på huvudspåret och dess längd minus N som argument. Vagnarna på stickspåret ges då genom att appenda de vagnar som kom från huvudspåret med de vagnar som redan fanns där. De vagnar som kom från huvudspåret ges av drop med samma argument som givet i föregående take eftersom samma mängd vagnar som ska tas bort från huvudspåret ska läggas till stickspåret.

Om N är mindre än noll ska vagnar flyttas från stickspåret till huvudspåret. Det får vi genom att vagnarna på huvudspåret appendas med de absolutbelopp av N första vagnarna på stickspåret. Första argumentet är vagnarna på huvudspåret och andra argumentet är en take från stickspåret med -N eftersom N är negativt. Vagnarna på stickspåret ges av en drop med argumenten One och -N.

Förflyttningarna mellan huvudspåret och det andra stickspåret bygger på exakt samma pricip. Även ett fall där N = 0 finns där inkommande State returneras.

```
case Move of
     {one, N} when N > 0  -> %From main to one
```

```
        {list:take(Main,length(Main)-N),
         list:append(list:drop(Main,length(Main)-N),One), Two};
    {one, N} when N < 0  -> %From one to main
        {list:append(Main,list:take(One,-N)),list: drop(One,-N), Two};
```

I uppgift 3 skulle en funktion move(Moves,State) som utför flera förflyttningar och returnerar en lista med states för varje förflyttning implementeras. Funktionen tar en lista med förflyttningar och ett tillstånd. Den har som basfall att om listan med förflyttningar är tomma listan, returneras starttillståndet State. Om listan med förflyttningar innehåller minst ett element görs en append mellan nuvarande tillstånd och nästa tillstånd. Nästa tillstånd fås rekursivt genom att move anropas med svansen av Moves, samt det nya tillståndet som beräknats med hjälp av single på första förflyttningen och State.

Uppgift 4 går ut på att skriva en funktion find(Xs,Ys) som tar två tillstånd och ska returnera en lista av förflyttningar som transformerar det ena tillståndet till det andra. För att göra detta behöver vi en funktion Split(Xs,Y) som returnerar en tuple innehållande en lista med elementen före Y och en lista med elementen efter Y. Vid basfallet när Xs är tomma listan returneras en tuple innehållande två tomma listor. Annars returneras en tuple innehållande listan av de element före Y i Xs, som ges av take(Xs,position(Xs,Y)-1) samt listan av de element efter Y i Xs, vilket ges av drop(Xs,position(Xs,Y). Första steget i given algoritm är att dela upp listan Xs i två listor. Vi matchar Hs med elementen före Y och Ts med elementen efter Y, där Y är första elementen i sluttillståndet Ys. Resterande steg i algoritmen är följande: Flytta Y samt Ts till första sidospåret. Flytta resterande vagnar från huvudspåret till det andra sidospåret. Flytta alla vagnar från det första sidospåret till huvudspåret. Flytta alla vagnar från andra sidospåret till huvudspåret. Detta resulterar i att en vagn nu står på rätt plats. Dessa fyra förflyttningar appendas till listan med förflyttningar som ska returneras tillsammans med ett rekursivt anrop på find med det nya tillståndet, alltså med Ts och Hs appendat samt listan Ys förrutom det första elementet. Basfallet till find är när antingen Xs eller Ys inte har några vagnar. Då returneras tomma listan.

```
find({Main,_,_},{MainY,_,_}) ->
[H|_] = MainY,
{Hs,Ts} = split(Main,H),
 list:append([{one,1+length(Ts)},{two,length(Hs)},{one,-1-length(Ts)},
{two,-length(Hs)}], find({list:append(Ts,Hs),[],[]},{list:drop(MainY,1),[],[]})).
```

Uppgift 5 gick ut på att man skulle göra en funktion few som hittade färre moves för att göra samma förflyttning som i find. Few består i en modifierad version av find där även man undersöker om nästa vagn redan är på rätt plats. Det vi gör är att lägga till ett nytt fall där vi matchar de två

tillstånden så att det första elementet i båda tillstånden är samma. Isåfall kallar vi på few med de två tillståndens tails.

```
few({[H|T],[],[]},{[H|Y],[],[]}) ->
     few({T,[],[]},{Y,[],[]});
```

I uppgift 6 ska en lista med förflyttningar komprimeras till en mindre lista. Det görs via följande regler: Om två förflyttningar till/från samma spår är efter varandra, lägg ihop dem till en gemensam förflyttning med summan av antal vagnar som ska förflyttas. Förflyttningar av 0 vagnar tas bort. Dessa regler ska implementeras tills att inga förändringar görs i listan med förflyttningar efter implementation av reglerna, vilket given funktion compress(Ms) ser till. Reglerna implementeras genom en funktion rules(Ms). Basfallet är då argumentet är tomma listan. Då returneras tomma listan. Därefter matchas listan med förflyttningar mot en förflyttning där 0 vagnar flyttas finnes först i listan, följt av en svans T. Är så fallet så returneras rules(T). I nästa fall matchas de två första förflyttningarna ifall de båda gäller samma spår. Är så fallet läggs en ny förflyttning til först i listan med summan av antalet förflyttade vagnar, följt av svansen som är ett anrop på rules med ursprunglig listas svans som argument. Matchar ingen av de tre fallen så delas inkommande lista upp i huvud och svans. Därefter returneras listan med huvudet först och som svans anropas rules med indatans svans som argument.

```
rules([]) ->
     [];
rules([{_,0}|T]) ->
     rules(T);
rules([{Tr,N},{Tr,M}|T]) ->
     [{Tr,N+M}|rules(T)];
rules([H|T]) ->
     [H|rules(T)].
```

3 Sammanfattning

Alla uppgifter i labben bygger på samma princip. Mönstermatching och rekursivitet i listmodifiering. När man väl kommit över tröskeln och förstått hur matchningen fungerade och hur man kunde iterera över en lista rekursivt var resten tämligen enkelt.

Det största problemet jag stötte på var att man inte kan sätta en lista som huvud till en ny lista, vilket ledde till att rekursiva anrop behövdes där man hela tiden lade första elementet som huvud och sedan hade det rekursiva anropet som svans. Hjälpfunktionerna och rules är de mest klara exemplen på detta men de andra funktionerna använde rekursion för att bygga upp listor, men då oftast med hjälp av append.