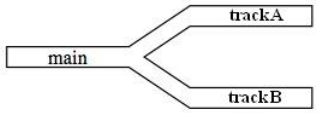
TDT4165 PROGRAMMING LANGUAGES
Fall 2012

Exercise 02
Togvogn-skifting

Problembeskrivelse

Du er sjef for å skifte vognene til et tog. Vi antar at hver vogn selv har en motor og at toget ikke har noe lokomotiv. Beskrivelsen av oppgaven din er gitt ved to sekvenser av vogner: sånn toget er før du begynner og sånn du ønsker det skal bli. Din oppgave er å omordne toget ved hjelp av en byttestasjon sånn at du ender opp med det ønskelige toget. Du skal ikke bare gjøre om på rekkefølgen, men gjøre dette ved hjelp av flytt. Byttestasjonen er vist i Figur 1. Den har et hovedspor "main" og to byttespor "trackA" og "trackB". En situasjon på byttestasjonen er kalt en tilstand. Et flytt beskriver hvordan vogner flytter fra spor til spor.



Figur 1: Togets byttestasjon

Mål

Det endelig målet med denne oppgaven er å finne en kort sekvens av flytt som endrer en sekvens av vogner til på "main" til en annen konfigurasjon på "main". Før vi prøver oss på dette målet skal vi fikse på modelleringen av problemet vårt og utvikle litt hjelp for å behandle lister.

Oppgavens mening

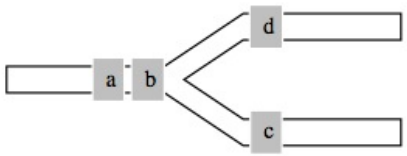
I denne øvingen skal vi se på mange viktige ting: Hvordan blir et problem modellert i datastrukturer som lister og poster (records)? Hvordan blir lister behandlet? Dette strekker seg fra enkle til mer kompliserte mønstre av rekursjon over lister. I tillegg er det meningen av oppgaven skal få deg godt i gang med Oz og Mozart. Sist, men ikke minst, håper vi at du har det litt moro med å

løse denne oppgaven. For å gjøre det litt mer moro har vi lagt med et program du kan bruke for å visualisere hva algoritmen din gjør.

Modellering

Tog, vogner og tilstander

Vogner er framstilt som atomer, og tog på banen er lister av atomer. Et tog har ingen duplikatvogner (det betyr at [a b] er et gyldig tog, mens [a a] ikke er gyldig). En komplett beskrivelse av tilstanden til byttestasjonen består av tre lister: en liste som beskriver "main" og to lister som beskriver sporene "trackA" og "trackB". En full tilstand er da en post (record) med `main`, `trackA` og `trackB` som feltnavn (features) med de tilsvarende listene som felter (fields). I tillegg skal merkelapp (label) for posten være `state`.



Figur 2: Et eksempel på en tilstand

Tilstanden `state(main:[a b] trackA:[d] trackB[c])` er vist i Figure 2.

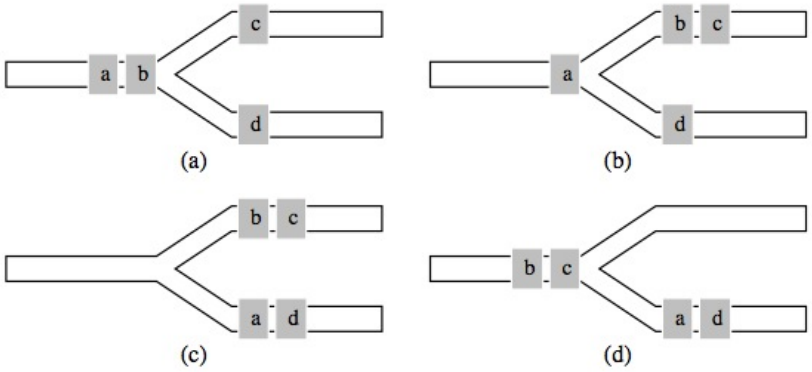
Flytt

Et flytt har en verdi, lagret som et felt i en tuppel (tuple) med merkelappen `trackA` eller `trackB`. (Tupler er det samme som poster, bare uten å spesifisere feltnavn.) Hvert flytt inneholder et tall, som sier hvor mange vogner som blir flyttet. For eksempel: `trackA(2)`, `trackB(2)` og `trackA(~3)` er alle flytt.

Utføre et flytt gitt en tilstand

Flytt beskriver hvordan en tilstand blir endret til en annen tilstand:

- Om flyttet er `trackA(N)` og N er større enn null, så blir de N vognene lengst mot høyre flyttet fra "main" til "trackA". Om det er flere enn N voger på "main" så blir resten igjen.
- Om flyttet er `trackA(N)` og N er mindre enn null, så blir de N vognene lengst mot venstre flyttet fra "trackA" til "main". Om det er flere enn N vogner på "trackA" så blir resten igjen.
- Flyttet `trackA(0)` har ingen effekt. Det samme gjelder for `trackB(0)`.



Figur 3: Flere flytt utført på tilstander

Eksempel

Figur 3 viser eksempler på flytt gjort på tilstander. (a) er starttilstanden, (b) etter `trackA(1)`, (c) etter `trackB(1)` og tilslutt (d) etter `trackA(~2)`.

Task 1 Litt listebehandling

Før du starter skal du lage noen listebehandlingsrutiner som trengs senere.

- {Length Xs} returnerer lengden til listen Xs
- {Take Xs N} returnerer en liste med de N første elementene av listen Xs. Om N er større enn antall elementer i Xs skal hele listen returneres i stedet.
- {Drop Xs N} returnerer Xs uten de første N elementene. Om N er større enn antall elementer i Xs skal `nil` returneres.
- {Append Xs Ys} returnerer en liste med alle elementene i Xs etterfulgt av alle elementene i Ys.
- {Member Xs Y} returnerer sann om Y finnes i Xs og usann ellers.
- {Position Xs Y} returnerer posisjonen til det første tilfellet av Y i Xs. Du kan gå ut i fra at Y finnes i listen.

Om du gjorde øving 1 kan du simpelthen kopiere løsningen derfra. Du kan også bruke biblioteket til å implementere de oppgitte signaturene, men dette kan være litt knotete, så det er kanskje lærerikt det også.

Uansett, legg alle definisjonene i en fil `List.oz`. Du kan inkludere dem i programmet ditt ved å skrive \insert 'List.oz' før koden som bruker dem.

Task 2 Utføre flytt

Den første deloppgaven er å skrive en funksjon `ApplyMoves` som tar inn en starttilstand og en liste av flytt. Den resulterer i en liste med tilstander. Om listen med flytt har N elementer vil den resulterende listen over tilstander ha N+1 elementer. Det første elementet er starttilstanden og det siste er den tilstanden toget endrer opp i. For eksempel:

```
{ApplyMoves state(main:[a b] trackA:nil trackB:nil) [trackA(1) trackB(1) trackA(~1)]}
```

returnerer listen

```
[state(main:[a b] trackA:nil trackB:nil)
state(main:[a] trackA:[b] trackB:nil)
state(main:nil trackA:[b] trackB:[a])
state(main:[b] trackA:nil trackB:[a])]
```

Vise frem tilstander

For å få en forståelse av tilstander kan du bruke det medfølgende visualiseringsprogrammet. For å bruke det må du laste ned `Visualizer.zip` fra hjemmesiden til faget, og pakke det ut i katalogen der filene dine ligger. Deretter kan du inkludere linjen `\insert 'Visualizer.oz'` i koden din, og kalle prosedyren `Visualize` på en liste over tilstander. Du kan prøve å visualisere eksempelet over. "Prev" og "Next" knappene gjør det mulig å gå gjennom tilstandene.

Struktur

Nå kan vi lage `ApplyMoves`. Angrip oppgaven på denne måten:

- Utfør hvert flytt rekursivt i rekkefølge.
- Om det ikke er noen flytt som skal gjøres (tom liste) skal en liste med starttilstanden som eneste element returneres.
- For hvert flytt skal programmet ditt bestemme ved mønstergjenkjenning hvilket spor som er innblandet.
- For hvert spor må du finne ut om vognene blir flyttet fra eller til sporet (om N er negativ eller positiv).
- Å ta vogner fra slutten av toget kan gjøres ved å bruke `Length` og `Drop`; å ta vogner fra starten av toget kan gjøres ved å bruke `Take`; å legge til vogner til et spor kan gjøres ved å bruke `Append`.

- Om S er en tilstand kan man få vognene på "main" ved å ta `S.main`.

Strukturen til `ApplyMoves` kan se slik ut:

```
fun {ApplyMoves S Ms}
   %% S is the state, Ms the list of moves to be applied
   %% Returns list of resulting states
   case Ms of nil then ...
   [] M|Mr then
      %% Compute S1 as new state
      S1 = case M of trackA(N) then
      if ... then ...  else ...  end
   [] trackB(N) then
      if ... then ...  else ... end
   end
   in
      ...
   end
end
```

Lagre programmet ditt i ei fil kalt `ApplyMoves.oz` Ikke glem å bruke visualiseringsprogrammet for å overbevise deg selv om at løsningen din virker.

Task 3 Å finne flytt

Lag en funksjon `Find` some tar to tog Xs og Ys som input og returner en liste med flytt slik at flyttene transformerer tilstanden:

```
state(main:Xs trackA:nil trackB:nil)
```

til

```
state(main:Ys trackA:nil trackB:nil)
```

Xs og Ys må her inneholde de samme elementene(vogner) og hver vogn må være unik. Dette betyr at Xs er en permutasjon av Ys. Angrip oppgaven slik.

- Problem skal løses rekursivt og hvert steg skal flytte en vogn til en posisjon krevd av Ys.
- Grunntilfellet er enkelt. Om det ikke er noen vogner trengs det ingen flytt.
- Ellers tar vi den første vognen Y fra Ys(det ønskede toget). Vårt mål er å finne en liste med flytt som fører vognen Y fremst på "main". Dette er gjort på følgende måte

- Del opp toget Xs i vognene Hs (hode) og Ts (hale) hvor Hs er vognene før Y i Xs og Ts er vognene etter Y i Xs. Skriv en funksjon `SplitTrain` som tar en liste av vognene Xs og vognen Y og returnerer paret Hs#Ts. For eksempel:

  ```
  {SplitTrain [a b c] a}= nil#[b c]
  ```

  Bruk funksjonene `Position, Take` og `Drop`.
- Flytt Y og de følgende vognene(Ts) til spor "trackA".
- Flytt resten av vognene(Hs) til spor "trackB".
- Flytt alle vognene fra "trackA" til "main" (inkludert Y).
- Flytt alle vognene fra "trackB" til "main".

- Etter at du har flyttet en vogn til riktig posisjon trenger vi bare å se på resten av vognene av både Xs og Ys (i den nye rekkefølgene på "main")

Lagre programmet ditt i `Find.oz`

Visualisere flytt

Du kan bruke visualiseringsprogrammet til å se at du gjør det riktig.

Eksempel

Gitt inputtoget [a b] og outputtoget[b a] er listen over flytt som blir laget av `Find`:

```
[trackA(1) trackB(1) trackA(~1) trackB(~1)
trackA(1) trackB(0) trackA(~1) trackB(0)]
```

Task 4 Finn færre flytt

Lag en funksjon `FewFind` som oppfører seg som `Find`, men for hver rekursiv operasjon tar for seg om den neste vognen allerede er i riktig posisjon. Hvis den er i riktig posisjon, er ingen flytt nødvendig. Gjør noen få endringer i ditt program `Find` (bare noen få modifikasjoner trengs). Lagre programmet ditt i `FewFind.oz`.

Eksempel

Gitt inputtoget [c a b] og outputtoget [c b a] er listen over flytt med `FewFind`:

```
[trackA(1) trackB(1) trackA(~1) trackB(~1)]
```

Task 5 Enda færre flytt

Listen over flytt laget av `FewFind` er fremdeles dårlig og kan lett optimiseres ved å innføre følgende regler:

- Erstatt `trackA(N)` direkte etterfulgt av `trackA(M)` med `trackA(N+M)`.
- Erstatt `trackB(N)` direkte etterfulgt av `trackB(M)` med `trackB(N+M)`.
- Fjern `trackA(0)`.
- Fjern `trackB(0)`.

Disse forbedringene er korrekt på den måten at en kortere liste av flytt vil føre til den samme sluttilstanden. Den oppgaven er litt vanskeligere. Tenk for eksempel på:

```
[trackB(~1) trackA(1) trackA(~1) trackB(1)]
```

Om man bruker reglene over flere ganger vil det resultere i ingen flytt i det hele tatt. Ved å bruke den øverste regelen får vi

```
  [trackB(~1) trackA(0) trackB(1)]
```

den tredje gir

```
 [trackB(~1) trackB(1)]
```

den andre gir

```
 [trackB(0)]
```

og til slutt gir regel fire

```
 nil
```

Lag en funksjon `Compress` som tar en liste med flytt og returnerer en komprimert liste med flytt. Angrip denne oppgaven sånn:

- Lag en funksjon `ApplyRules` som kjører gjennom regler rekursivt.
- Deretter kjører du `ApplyRules` om igjen til listen av flytt ikke endrer seg.

`Compress` blir seende sånn ut:

```
fun {Compress Ms}
   Ns={ApplyRules Ms}
in
   if Ns==Ms then Ms else {Compress Ns} end
end
```

Lagre programmet ditt i `Compress.oz`

Task 6 Finn virkelig få flytt

Problemet med både `Find` og `FewFind` er at du alltid putter alle vognene fra "trackA" og "trackB" tilbake til "main" selv om det kan hende det finnes muligheter for å bruke "trackA" og "trackB" til å dytte vognene i riktig posisjon. Vi skal utnytte dette. Lag en funksjon `FewerFind` som tar fire argumenter: Ms for vognene på "main", Os for vognene på "trackA", Ts for vognene på "trackB" og Ys som er ønsket tog. `FewerFind` virker rekursivt som før og hvert rekursive kall vil flytte vognen Y av Ys til den riktige posisjonen. Det som er nytt er at en vogn kan være på hvilket som helst spor.

- Om Y finnes i Ms flytt den til riktig posisjon som gjort tidligere, men la vognene som flyttes unna stå igjen på spor "trackA" og "trackB".
- Om Y er på spor Os: Flytt vognene som er foran først til "main" også til "trackB". Deretter flytt Y til sin posisjon. Hvis ikke så lar vi vognene på "trackA" og "trackB" være uendret. Dette legger til en vogn i riktig posisjon på "main".
- Gjør det samme for "trackB".

Hvert rekursive kall til `FewerFind` må selvsagt fordele vognene riktig mellom alle tre sporene. Når `FewerFind` kalles første gang er "trackA" og "trackB" tomme lister. For eksempel:

```
{FewerFind [a b] nil nil [b a]}
```

returnerer

```
[trackA(1) trackB(1) trackA(~1) trackB(0) trackA(0) trackB(~1)]
```

Lagre programmet ditt i `FewerFind.oz`