Task 10 - Train Shunting

Casper Kristiansson

February 18, 2022

Introduction

Task ten of the course Programming II consisted of shunting of different wagons. The main goal of the task is to change the layout of the train wagons with the help of three different tracks. This should be achieved without breaking any physical rules which means that wagons cannot jump over other wagons. The author will be developing the basic operations for moving wagons and the writing algorithms for finding the necessary moves to get a desired wagon pattern.

Result

The author will be dividing the result section into three different parts. The first part consists of the list processing module, the second part involves the functions for applying train shunting and the last one for solving and finding shunting moves.

List processing

The first part of the task was to implement a module for processing and manipulate lists. The module contains six different type of functions; **take, drop, append, member, position,** and lastly **split**. The most basic function is take which returns the n first elements from a list and the function drop which remove the first n elements from a list.

The third important function is **split** which takes a list and an element as input. The method will return two different lists, one for the elements left and one for the elements right of the input element. This function is used by the algorithms for solving and finding different wagon layouts.

```
def take([], _) do [] end
def take(_, 0) do [] end
def take([head | tail], n) do
  [head] ++ take(tail, n - 1)
end

def drop([], _) do [] end
def drop(xs, 0) do xs end
def drop([_ | tail], n) do
  drop(tail, n - 1)
end

def split(xs, x) do
  {take(xs, position(xs, x) - 1), drop(xs, position(xs, x))}
end
```

Figure 1: Three most important functions of list processing

Moves

The next part of the task is to implement the **moves** module which contains the functions for applying different shunting moves. The module has two distinct functions, one for applying a single move on a state and one which receives a list of moves and returns all the different states for those moves.

```
def move([], state) do [state] end
def move([move | moves], state) do
  nextState = single(move, state)
  [state] ++ move(moves, nextState)
end
```

Figure 2: Recursively applying multiple moves on a state

The single move method starts off by checking if the input track is either the top one (**:one**) or the bottom one (**:two**). The function than checks if the steps (wagons) is either positive or negative. If it's positive the track **:one** should receive n wagons from the main track. If its negative the main track will receive n wagons from track :one. To add the correct wagons when the step is positive the list needs to be reversed to extract the last wagons of the list. Figure three consists of the method for moving wagons between the main track and the track **:one**.

```elixir
def single({:one, steps}, {main, one, two}) do
  cond do
    steps > 0 ->
      main = Enum.reverse(main)
      one = Enum.reverse(one)
      {
        Enum.reverse(Processing.drop(main, steps)),
        Enum.reverse(Processing.append(one, Processing.take(main, steps))),
        two
      }
    steps < 0 ->
      {
        Processing.append(main, Processing.take(one, -steps)),
        Processing.drop(one, -steps),
        two
      }
    true ->
      {main, one, two}
  end
end
```

Figure 3: The function for adding or removing wagons from track :one

Shunting

The next part of the task was to implement functions for converting a certain wagon pattern to another one. The different functions that was develop was **find, few, compress, fewer**. The author will be discussing the basic function **find** and the more advanced function **fewer**.

The function find builds on an algorithm where it will constantly move wagons between track :one and :two to the main track. By doing this the program is able to change the order that the wagons are placed in. In a more detail way, the function starts of by checking the first element in the desired wagon order. The list will than split and move the right wagons and the desired wagon to track one. It will than move the remaining wagons (minus the found wagons) to track :two. The wagons from track :one will then be moved back to the main track. Using this algorithm, it will take a total of four moves per wagon ($4n$).

```
def find(_, []) do [] end
def find(xs, [y | ys]) do
  {hs, ts} = Processing.split(xs, y)

  moves = [
    {:one, 1 + Enum.count(ts)},
    {:two, Enum.count(hs)},
    {:one, -(1 + Enum.count(ts))},
    {:two, -(Enum.count(hs))}
  ]

  moves ++ find(ts ++ hs, ys)
end
```

Figure 4: The basic algorithm for finding all of the different moves

The more advanced algorithm which uses the tracks :one and :two to store the other wagons only requires three moves per wagon. The algorithm starts of by checking if the required wagon exists either in the main, :one, :two track. Depending in what track it exists the function will be moving either the most left or right elements to the other tracks.

```
def fewer(_, _, _, []) do [] end
def fewer(ms, hs, ts, [y | ys]) do
  cond do
    Processing.member(ms, y) ->
      {hsSplit, tsSplit} = Processing.split(ms, y)
      moves = [
        {:one, 1 + Enum.count(tsSplit)},
        {:two, Enum.count(hsSplit)},
        {:one, -1}
      ]

      [{ms, hs, ts}] = Enum.take(Moves.move(moves, {ms, hs, ts}), -1)
      moves ++ fewer(ms, hs, ts, ys)
  ...
```

Figure 5: More advanced algorithm for finding fewer moves for a desired wagon pattern

Discussion

The author didn't have enough space to discuss or show the code for the functions few and compress. The important function of those two is the compress function. The method will receive a list of moves and return a more compact list of those moves. The process builds upon four different rules, where moves could either be removed or merged. Those moves are when the algorithms produce a move with zero wagons or if they produce two moves in a row from the same track.

The author thought that this task was extremely fun to solve and to develop an algorithm for finding different moves the train wagons could take for finding a specific pattern. The author had a bit of problem at the start of developing an algorithm which can solve bigger wagons with at least four elements. But the author solved it by debugging the program and checking what the algorithm did at each recursion.