Train Shunting - Report

Adrian Hjert

02/Mar/2023

Introduction

The aim of this report is to explore the methods required to solve the train shunting problem in Elixir. The report will discuss some of the functions that were created and how they work. The report will also briefly discuss some of the results.

Methods

This section of the report will be divided into tree parts to make it easier to read and separate between the files required to solve this assignment.

Train

The first task is to create the trains we will be using for the later parts of the task. This file contains mostly different checks so we can compare the trains and see if a wagon belongs to a train. It also involves the **take/2** and the **drop/2** functions:

```
def take(_, 0) do [] end
   def take([front|rest], wgns) when
    wgns > 0 do [front|take(rest, wgns-1)] end

   def drop(front, 0) do front end
   def drop([_front|rest], wgns) when
    wgns > 0 do drop(rest, wgns-1) end
```

These functions are in a way just a variation of one another where **take/2** returns the first n wagons of the train (including the first wagon), whereas **drop/2** omits the first wagon. The trains are returned as lists and the functions works recursively by first finding the first wagon (if **take/2** first wagon is *front*, if **drop/2** first wagon will be the wagon after *front*. This can be seen by the way we have *front* as the first element in the list followed by the *rest* of the wagons).

Another important funciton is the **split/2** function which essentially breaks up a train into two parts at a specified wagon (second argument) and returns the segment before the specified wagon, and the segment after, not including the specified wagon. The code for this function is as follows.

```
def split([iWgn|rest], iWgn) do {[], rest} end
  def split([front|rest], iWgn) do
    {rest, drop} = split(rest, iWgn)
    {[front|rest], drop}
  end
```

This function is called recursively to cycle through the wagons (if any) up until *iWgn*. It then cycles through the wagons after *iWgn* (if any) until the end is reached, it then returns a tuple of the first part and the second part of the train, without representing *iWgn* in the returned tuple.

The **main/2** function is responsible for telling us how wagons have moved from the front of the train to the rear. The function takes two arguments, the first of which is a list containing the train, and the second argument containing an integer indicating the number of wagons to be moved as stated. The **main/2** function can be seen in the following code.

```
def main([], wgns) do {wgns, [], []} end
  def main([front|rest], wgns) do
      case main(rest, wgns) do
        {0, drop, take} -> {0, [front|drop], take}
        {wgns, drop, take} -> {wgns-1, drop, [front|take]}
      end
  end
```

This function will return a list indicating how many wagons have yet to be moved, the wagons at the front of the train, and the wagons are at the end of the train. The function works recursively and decrementing *wgns* by 1 each time until there are no more wagons to be moved.

Moves

The **Moves** module relies heavily on the **Train** module to work given that this module has to do with keeping track of and actually moving different parts of the train. This module has two main functions, each of course with their separate cases. One of them is the **seq/2** function which can be seen in the following lines of code.

```
def seq([], state) do [state] end
  def seq([move|rest], state) do
    [state|seq(rest, single(move, state))]
  end
```

This function is responsible for returning a list containing the different states a train goes through when a sequence of instructions is performed on it. Essentialy, the function works by taking in a list of instructions to be performed on the train, and the state of the train. The function then uses then **single/2** function (seen later in this subsection) to apply each instruction to the train and updating its state after each instruction. Naturally, recursion is crucial to this function. Once there are no more instructions to be executed, the recursion stops and the train states are returned as a list.

The **single/2** function is responsible for applying instructions to wagons and trains. There are two different kinds of instructions, namely **:one** and **:two** which indicate the direction/train to be operated on. The code for the different directions is rather similar so only one will be shown, along with the base case.

```
def single({_, 0}, state) do state end
def single({:one, dir}, {main, one, two}) when dir > 0 do
  {0, rem, wgns} = Train.main(main, dir)
  {rem, Train.append(wgns, one), two}
end
def single({:one, dir}, {main, one, two}) when dir < 0 do
  wgns = Train.take(one, -dir)
  {Train.append(main, wgns), Train.drop(one, -dir), two}
end
```

The base case is rather obvious, if no wagons are to be moved then the initial state is returned. In other cases, the function uses pattern matching on the direction and the state the train is in at the time that instruction is executed. This function makes use of the **Train** module using the **take/2** and **drop/2** functions from that module for negative values of *dir*, *i.e.* we are removing wagons from the train, and **main/2** to add wagons to the train in case of *dir* being positive, with the state being constantly updated.

Shunt

This is the module responsible for solving the shunting problem. The first function we need to implement is the **find/2** function. This function essentially splits the train wagons into two parts and determines how many wagons need to be shunted to different tracks. This function runs recursively until all tracks have the correct wagons on them. The code for this is as follows.

```
def find(xs, [y|ys]) do
    {hs, ts} = Train.split(xs, y)
    tn = length(ts)
```

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

The base case is omitted as it is simply two empty lists each showing the configurations on the tracks, so, no moves need to be made, thus, an empty list is returned.

Now, a function **few/2** is created to slightly optimize the number of moves we need to perform to get the configuration we want. By "optimize" in this case we mean to reduce the number of moves. This function has one more case than **find/2**, otherwise it is rather similar with only a few very small modifications. The additional case can be seen in the following code.

```
def few([h|hs], [y | ys]) when h == y do
    few(hs, ys)
  end
```

To further optimie the number of moves, we can implement a function **rules/1** which alter the way we consider moves, for example, since the move {_, 0} does not affect any wagons on the tracks, it can be omitted. Furthermore, consecutive moves on the same type can be altered to be one move, for example, {:two, 1} followed by {:two, 2} can be combined as {:two, 3}.

A final function **compress/1** is given in the instructions which applies **rules/1** recursively until no more changes occur in the list of moves outputted by the mdoule.

Results and Conclusions

This section will show some inputs to some of the functions discussed in the **Methods** section and their respective outputs. The **split/2** function of the **Train** module is tested as follows.

```
def test1() do
      front = [:a]
      rest = [:b, :c, :d, :e]
      split([front | rest], :d)
    end
```

This returns {[[:a], :b, :c], [:e]} as expected The **seq/2** function of the **Moves** module is tested using the inputs

```
seq([{:one, 1}, {:two, 2}, {:one, -1}, {:two, -1}],
 {[:a,:b, :c], [:d], [:e, :g]})
```

This gives the output

```
[ {[:a, :b, :c], [:d], [:e, :g]}, {[:a, :b], [:c, :d], [:e, :g]},
  {[], [:c, :d], [:a, :b, :e, :g]}, {[:c], [:d], [:a, :b, :e, :g]},
  {[:c, :a], [:d], [:b, :e, :g]}]
```

Note that some line breaks are removed to conserve some space. The **find/2** function from the **Shunt** mofule is tested like this:

```
def test() do
   start = [:a, :b, :c, :d]
   fin =  [:d, :a, :b, :c]
   find(start, fin)
 end
```

Which returns [one: 1, two: 3, one: -1, two: -3, one: 3, two: 0, one: -3, two: 0, one: 2, two: 0, one: -2, two: 0, one: 1, two: 0, one: -1, two: 0]