Train Shunting

Axel Månson Lokrantz

Spring Term 2023

Introduction

Train shunting is a problem where you have to to rearrange a sequence of wagons to a desired order in the least amount of moves, The function should return the sequence of moves needed to rearrange the train to its desired state. The shunting station has a ”main” track and two shunting tracks ”one” and ”two”. The following program was written in Elixir together with Daniel Dahlberg at KTH.

Github link

A link to the program in its entirety can be seen here.

Modeling

Wagons are modeled as atoms and trains as list of atoms. A train cannot have a duplicate wagons. Each track, consists of a list which holds the wagons that are currently located at that particular track. The tracks are ordered in a way so that we can only move the right most wagons from the main track and the left most wagons from the ”one” and ”two” tracks. If the move is `:one, n` and n is greater than zero then we move n number of wagons from the ”main” track to ”one”. If n is less than zero we move n number of wagons from ”one” to main. The same goes for ”two”. If n is equal to zero no wagons are moved.

Train Processing

To get started we implement a number of functions that will be needed in a later stage of the program.

- `take(train,n)` Returns the train containing the first n wagons of the train.

```
def take(_, 0) do [] end
def take([h | tail], n) do
```

```
    [h | take(tail, n-1)]
end
```

- `drop(train,n)` Drops n number of wagons from the front of the train and returns the remaining wagons.

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

- `append(train1, train2)` appends and returns a combination of the two trains.

```
def append(train1, train2) do train1 ++ train2 end
```

- `member(train, y)` Test whether or not y is a wagon of the train.

```
def member([y | _], y) do true end
def member([], _) do false end
def member([_ | tail], y) do
    member(tail, y)
end
```

- `split(train, y)` Return a tuple with two trains, all the wagons before y and all wagons after y. Y will not be part of either.

```
def split([], _, _) do false end
def split([y|tail], y, train1) do
    {Enum.reverse(train1), tail}
end
def split([h|tail], y, train1) do
    split(tail, y, [h | train1])
end
def split(train, y) do
    split(train, y, [])
end
```

- `main(train, n)` Returns a tuple {`k, remain, take`}. The function will take a sequence of wagons (train) and place n number of wagons in "take", the remaining wagons, if any, will be placed in "remain". K acts as the remaining moves. If n was set to 8, but the train only contained 4 wagons k would be equal to 4, "take" would equal the 4 wagons and "remain" would equal an empty list.

```
def main([_|tail], n, acc, remain, take) do
    main(tail, n-1, acc + 1, remain, take)
```

```
    end
    def main([], _, acc, _, _) do acc end
    def main(train, n) do
        acc = main(train, n, 0, [], [])
        if acc-n < 0 do
            {-(acc - n), [], train}
        else
            {0, take(train, acc-n), drop(train, acc-n)}
        end
    end
```

Applying moves

To apply the moves we start off by creating a binary function `single` that takes a move and an input state and returns a new state computed from the state with the move applied. To move a wagon we will use the previously implemented `main` function.

```
def single(move, tracks) do
    main = elem(tracks, 0)
    track1 = elem(tracks, 1)
    track2 = elem(tracks, 2)
    case move do
      {:one, n} ->
        if n > 0 do
          moved_wagons = main(main, n)
          remain = elem(moved_wagons, 1)
          take = elem(moved_wagons, 2)
          {remain, append(take, track1), track2}
        else
          remain = drop(track1, -n)
          take = take(track1, -n)
          {append(main, take), remain, track2}
        end
      {:two, n} ->
        if n > 0 do
          moved_wagons = main(main, n)
          remain = elem(moved_wagons, 1)
          take = elem(moved_wagons, 2)
          {remain, track1, append(take, track2)}
        else
          remain = drop(track2, -n)
          take = take(track2, -n)
          {append(main, take), track1, remain}
```

```
        end
        {_, 0} -> tracks
    end
end
```

Lastly we create a function sequence which takes a list of moves and a state and returns a list of states that represent the transitions when the moves are performed.

```
def sequence([], _) do [] end
 def sequence([h|tail], tracks) do
   move = single(h, tracks)
   [move | sequence(tail, move)]
 end
```

The Shunting Problem

Now, for the actual problem we create function `single` that takes two trains `xs` and `ys` and returns a list of the moves needed for `xs` to transform into `ys`.

The problem is solved recursively and the goal is to take wagon `y` from its current position in `xs` to being the left-most wagon in the main track train.

1. Split the train `xs` into the wagons `hs` and `ts`. `hs` are all wagons before `y` and `ts` are all wagons after `y`.

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

2. Calculate the length of `hs` and `ts`.

   ```
   tn = Enum.reduce([y|ts], 0, fn(_, acc) -> acc + 1 end)
   hn = Enum.reduce(hs, 0, fn(_, acc) -> acc + 1 end)
   ```

   We chose to include `y` in `tn` for easier indexation.

3. Move `ts` to `:one` and `hs` to `:two` from the main track. Move all wagons on `:one` to main and all wagons on `:two` to main. Reapet recursively.

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

   With this code we get a functioning program, that returns a sequence of moves required to transition between the two states. Problem is that we also get redundant moves. A better solution would be to

check each recursive iteration if the next wagon is already in its right position, if so no moves are needed. This is done by one more line of code.

```
def find([y | xs], [y | ys]) do find(xs, ys) end
```

Move compression

To further trim the sequence list we can remove moves that do nothing such as {:one, 0} or {:two, 0}. We could also try to shorten the sequence by eliminating adjacent moves that are self-contradictory for example, {:one, -1} followed by {:one, 1} since two contradictory moves might not be adjacent the first round of eliminating moves. This has to be done recursively through the `compress` function which was given in the assignment.

```
def rules([]) do [] end
def rules([{:one, 0}|tail]) do rules(tail) end
def rules([{:two, 0}|tail]) do rules(tail) end
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([move|tail]) do
    [move|rules(tail)]
end
```