Train shunting

Programming II - Elixir Version

Christian Schulte

adapted to Elixir by Johan Montelius

Spring Term 2023

Introduction

You are in charge of shunting wagons of a train. In the following we assume that each wagon is self driving and that the train has no explicit engine.

The description for your shunting task is given by two sequences of wagons: the given train and the desired train. Your task is to rearrange the given train with help of your shunting station such that the desired train is obtained. You are not only supposed to rearrange the train but also to compute a sequence of shunting moves (which are called just "moves" from now on).

The shunting station is shown in Figure 1. It has a "main" track and two shunting tracks "one" and "two". A situation in the shunting station is called *state*. A *move* describes how wagons move from one track to another.

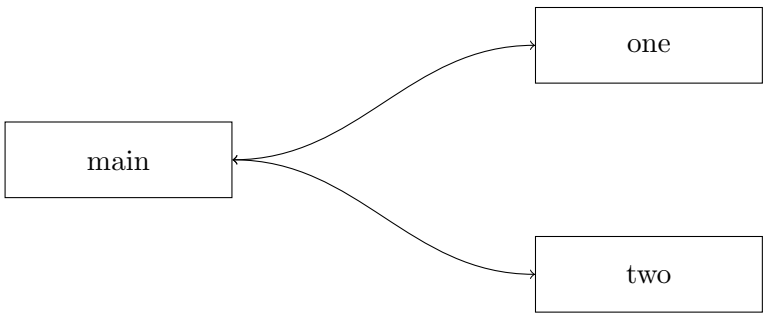

Figure 1: Train shunting station

Goal Our ultimate goal in this lab is to find a short sequence of moves that turn a train on the track "main" into another configuration of the train on "main".

Before we attempt this goal we will fix the modeling of our problem and develop some list processing support.

Lab purpose This lab exercises several important issues. How are problems modeled by data structures such as lists and tuples. How are lists processed. This ranges from simple to more complicated patterns of recursion over lists. This lab is of course also geared at getting you started with Erlang and functional programming in general.

And last, but not least, we hope that you have *some fun* solving this little puzzle.

Modeling

Trains, wagons, and states Wagons are modeled as atoms and trains on tracks as lists of atoms. A train has no duplicate wagons (that is, `[:a,:b]` is a train, whereas `[:a,:a]` is not).

A complete description of the state of a shunting station consists of three lists: a list describing the train on track "main", and two lists describing tracks "one" and "two".

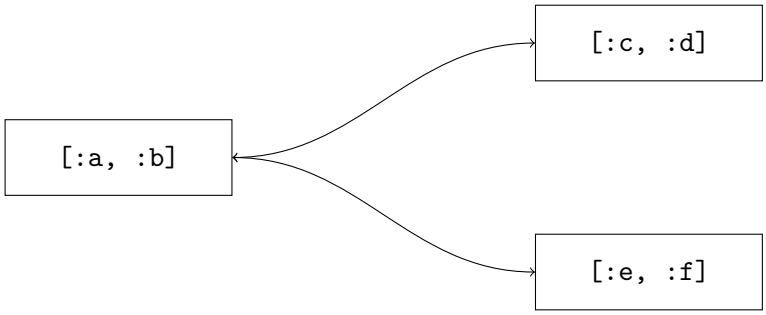

Figure 2: Example state displayed.

right-most, left-most One important question is of course which wagons are the *right-most* and *left-most*. We can of course choose how we represent our tracks but will chose a way that follows how we write our tracks.

For track one and two the first element in the list will be the wagon closes to the path to the main track. The first element in the list that represents the main track is the left-most wagon on the track i.e. furthest from track one and two.

The state `{[:a,:b],[:c, :d],[:e, :f]}` is visualized in Figure 2.

Moves A move is a binary tuple. The first element of a move is either `:one` or `:two`. The second element of a move is an integer. For example, `{:one,2}`, `{:two,2}`, and `{:one,-3}` are all moves.

Applying a move to a state Moves describe how one state is transformed into another:

- If the move is `{:one,n}` and `n` is greater than zero, then the `n` right-most wagons are moved from track "main" to track "one".

  If there are more than `n` wagons on track ":main", the other wagons remain.

- If the move is `{:one,n}` and `n` is less than zero, then the `n` left-most wagons are moved from track "one" to track "main".

  If there are more than `n` wagons on track "one", the other wagons remain.

- The move `{:one,0}` has no effect.

The same holds true for moves with first element `:two` concerning track "two".

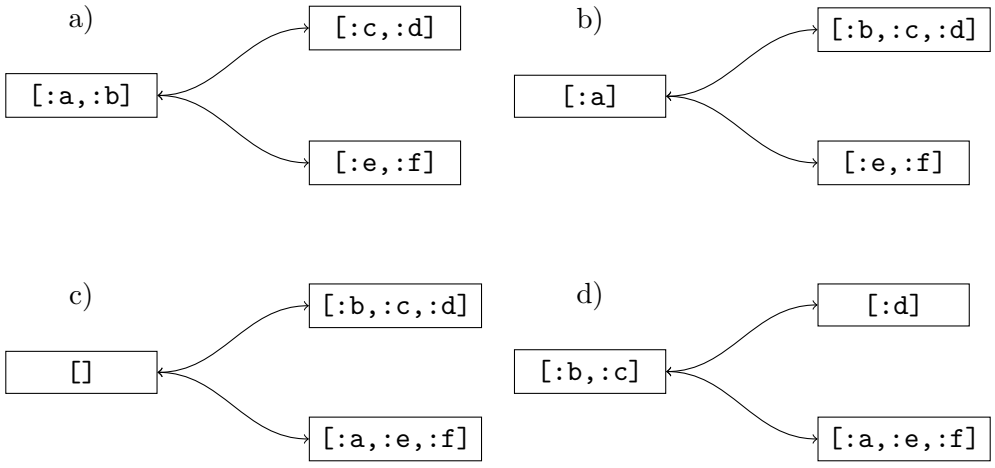

Figure 3: Moves applied to states.

Example Figure 3 shows examples of moves applied to states, where (a) is the initial state, (b) after application of `{:one,1}`, (c) after application of `{:two,1}`, and finally (d) after application of `{:one,-2}` (note the order).

Some train processing

Before we actually start, we will develop some train-processing routines that you will need later. When implementing these functions you should implement them from scratch i.e. not use library modules such as Lists, Enum etc. Why - because you will practice writing recursive functions.

1. `take(train,n)` returns the train containing the first `n` wagons of `train`.
2. `drop(train,n)` returns the train `train` without its first `n` wagon.
3. `append(train2,train2)` returns the train that is the combinations of the two trains.

   For example,`append([:a,:b],[:c])` returns `[:a, :b, :c]`.
4. `member(train,y)` tests whether `y` is a wagon of `train`.
5. `position(train,y)` returns the first position (1 indexed) of `y` in the train `train`. You can assume that `y` is a wagon in `train`.

   For example, `position([:a,:b,:c],:b)` returns `2`.
6. `split(train, y)` return a tuple with two trains, all the wagons before `y` and all wagons after `y` (i.e. `y` is not part in either).

   For example:

   ```
   split([:a,:b,:c],:a) = {[],[:b,:c]}

   split([:a,:b,:c],:b) = {[:a],[:c]}
   ```
7. `main(train, n)` returns the tuple `{k, remain, take}` where `remain` and `take` are the wagons of `train` and `k` are the numbers of wagons remaining to have `n` wagons in the taken part.

   For example:

   ```
   main([:a, :b, :c, :d], 3) = {0, [:a], [:b, :c, .d]}

   main([:a, :b, :c, :d], 5) = {1, [], [:a, :b, :c, .d]}
   ```

The last function requires some explanation; the wagons on the main track are in reverse order i.e. the first wagon in the list is in the leftmost position on the track. When you're asked to move two wagons to another track you should dive the train into two segments; the segment that should remain and the two wagons (to the right i.e. in the end of the list) that should be moved.

You could of course implement this by first counting the number of wagons on the track and then decide how many to take and drop, but why not do this in one recursive function? Implement the function `main/2` as one recursive function without using the functions that you have used before. In the report, describe how you have implemented the function.

Please put all definitions together in one module `Train` in a file `train.ex`

Applying moves

The first task is writing a binary function `single/2` that takes a move and an input state. It returns a new state computed from the state with the move applied.

For example, `single({:one,1},{[:a,:b],[],[]})` returns `{[:a],[:b],[]}`.

`single` should be used later in this assignment whenever a move is to be performed on a state.

Approach the task as follows:

- Your program should decide by pattern-matching which track is involved and what the different elements of a state are.
- For a track, you have to decide whether wagons are moved *on* or *from* the track (that is, is `n` positive or negative).
- Take into account that moves are allowed where no wagons are moved at all!
- Use the `main/2` function when moving wagons from the main track.

You should also implement a function `sequence/2` that takes a list of moves and a state an returns a list of states that that represents the transitions when the moves are performed. For example,

```
sequence([{:one, 1}, {:two, 1}, {:one, -1}, {:two, -1}], {[:a,:b], [], []})
```

should return the lists:

```
[
  {[:a, :b], [], []},
  {[:a], [:b], []},
  {[], [:b], [:a]},
  {[:b], [], [:a]},
  {[:b, :a], [], []}
]
```

You can use this function to verify that the sequences of moves that you generate actually do solve the problem given.

Please store your function in a module called `Moves` in the file `moves.ex`.

1 The shunting problem

So now for the actual problem, finding a sequence of moves that changes the order of the wagons on the main track.

Develop a procedure `find` that takes two trains `xs` and `ys` as input and returns a list of moves, such that the moves transform the state `{xs,[],[]}` into `{ys,[],[]}`.

In the following, we require that `xs` and `ys` contain the same elements (wagons) and that each wagon is unique (in other words, `xs` and `ys` are permutations of each other).

Approach the problem as follows. The problem is solved recursively and each recursive step will move one wagon in the position as required by `ys`.

The base-case is simple. If there are no wagons, no moves are needed.

Otherwise, we take the left-most wagon `y` from `ys` (the desired train). Our goal is to find a list of moves that takes the wagon `y` from its current position in `xs` to being the left-most wagon in a train on the main track. This is done by the following moves:

1. Split the train `xs` into the wagons `hs` and `ts`, where `hs` are the wagons before `y` in `xs`, and `ts` are the wagons after `y` in `xs`.
2. Move `verb+y+ and the following wagons (that is` verb+ts+) to track "one".
3. Move the remaining wagons (that is, `hs`) to track "two".
4. Move all wagons on "one" to "main" (this includes `y`, which goes as needed to the left-most position on "main").
5. Move all wagons on "two" to "main".

After having moved one wagon in the right position, we only need to consider the remaining wagons of `ys`. We should of course update the current state on the main track but the state is simply `ts` appended to `hs` (we moved `ts` to the main track before `hs`).

Note that the function `find/2` does not need to "perform" the moves using `single/2` or `sequence/2`, we know what the final state should look like. You can, or should, use `sequence/2` to verify that `find/2` actually takes us from the initial state to the desired.

Please store your functions in the module `Shunt` (and file `shunt.ex`).

Example. Given the input train `[:a,:b]` and the output train `[:b,:a]`, the list of moves computed by `find` is:

```
[{:one,1},{:two,1},{:one,-1},{:two,-1}
 {:one,1},{:two,0},{:one,-1},{:two,0}]
```

Finding less moves

As you probably noticed, a generated sequence of moves contains a lot of redundant moves. Develop a function `few` that behaves as `find` but that takes for each recursive application into account whether the next wagon is already in the right position. If so, no moves are needed.

Proceed by modifying (only very few modifications are needed) your program for `find/2`.

Please store `few/2` also in the module `Shunt`.

Example. Given the input train `[:c,:a,:b]` and the output train `[:c,:b,:a]`, the list of moves computed by `few` is:

```
[{:one,1},{:two,1},{:one,-1},{:two,-1}]
```

2 Move compression

The list of moves computed by `few/2` is still awkward and can be easily optimized according to the following rules:

1. Replace `{:one,n}` directly followed by `{:one,m}` with `{:one,n+m}`.
2. Replace `{:two,n}` directly followed by `{:two,m}` with `{:two,n+m}`.
3. Remove `{:one,0}`.
4. Remove `{:two,0}`.

These optimizations are *correct* in the sense that the shorter list of moves will compute the same final state.

This task is actually tricky: think for example of

```
[{:two,-1},{:one,1},{:one,-1},{:two,1}]
```

Repeatedly applying the rules from above actually results in no moves at all. By application of Rule 1 we obtain `[{:two,-1},{:one,0},{:two,1}]`; by Rule 3 `[{:two,-1},{:two,1}]`; by Rule 2 `[{:two,0}]`; and finally by Rule 4 `[]`.

Develop a function `compress/1` that takes a list of moves and returns a compressed list of moves. Compression must be complete, that is, none of the above rules should be applicable to the returned list of moves.

Approach this task as follows. Develop a procedure `rules/1` that applies rules recursively. Then repeat application of `rules/1` until the list of moves does not change. Thus, `compress` is implemented as follows:

```
def compress(ms) do
    ns = rules(ms)
    if ns == ms do
       ms
    else
       compress(ns)
    end
end
```

Please store your program also in the module `Shunt`.

3 Finding really few moves

This assignment is voluntary. This means you don't have to do it, however we strongly encourage you to do it. And actually it is fun!

The problem with both `find/2` and `few/2` is that they always push back the wagons from "one" and "two" to "main", even though there might be some opportunity to actively use track "two" to push wagons from track "one" into position and vice versa. In the following, we are going to take advantage of this.

Develop a procedure `fewer`, that takes four arguments: `ms` as the wagons on "main", `os` as the wagons on "one", `ts` as the wagons on "two", and `ys` as the desired train.

`fewer` works recursively and as before, each recursive invocation will bring the first wagon `y` of `ys` into the right position. What is new, is that this wagon might be on either track:

- If `y` is a member of `ms`, bring it in the right position as done previously. Leave the other wagons on track "one" and "two".
- If `y` is a member of `os` (it is on track "one"), move the wagons in front first to "main" and then to "two". Then move `y` into position. Otherwise, leave the wagons on "one" and "two" unchanged.

  This adds one more wagon in the right position on "main".
- Do the same for "two".

Each recursive application of `fewer` has of course to supply the wagons on all three tracks correctly!

Initially, `fewer` is applied such that the tracks "one" and "two" are the empty list. For example,

```
fewer([:a,:b],[],[],[:b, :a])
```

returns

```
[{:one,1},{:two,1},{:one,-1},{:two,0},{:one,0},{:two,-1}]
```

4 Acknowledgments

The idea and the initial problem formulation is taken from an assignment at the 8'th Prolog Programming Competition organized by Bart Demoen and Phuong-Lan Nguyen.