Train shunting

Programming II - Elixir Version

Christian Schulte

adapted to Elixir by Johan Montelius

Spring Term 2021

1 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.

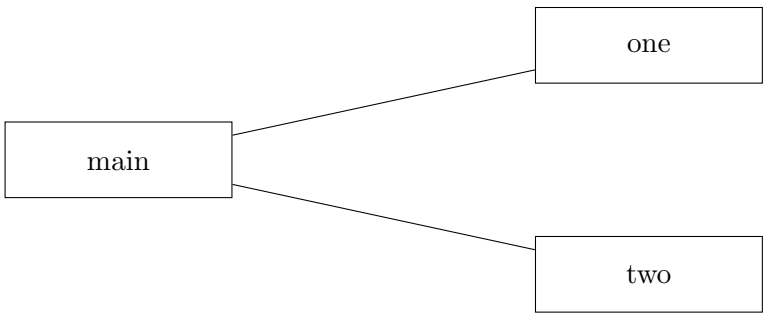

Figure 1: Train shunting station

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.

2 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". An entire state is represented as a tuple with three elements, where the first element represents the train on track "main", the second the train on "one", and the third on track "two".

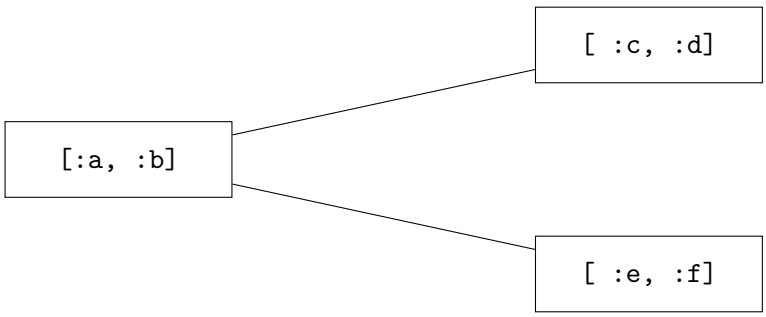

Figure 2: Example state displayed.

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".

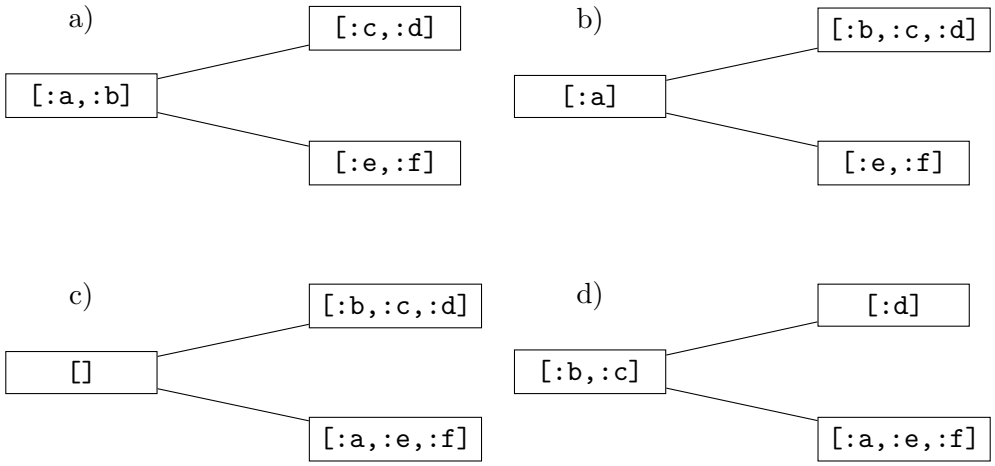

Figure 3: Moves applied to states.

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}`.

3 Some list processing

Before we actually start, we will develop some list-processing routines that you will need later.

1. `take(xs,n)` returns the list containing the first `n` elements of `xs`.

2. `drop(xs,n)` returns the list `xs` without its first `n` elements.

3. `append(xs,ys)` returns the list `xs` with the elements of `ys` appended.

4. `member(xs,y)` tests whether `y` is an element of `xs`.

5. `position(xs,y)` returns the first position of `y` in the list `xs`. You can assume that `y` is an element of `xs`.

   For example, `position([:a,:b,:c],:b)` returns `2`.

Please put all together in one file `list.erl` and add an appropriate module declaration into that file: add a module declaration and export the functions listed above.

4 Applying a Single Move

The first task is writing a binary function `single` 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.

Structure. Now we are in position to develop `single`. 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).
- Taking wagons from the end of a train can be done by using `length` and `drop`; taking wagons from the beginning of a train can be done by using `take`; adding wagons to a track can be done by `append`.
- Take into account that moves are allowed where no wagons are moved at all!

Please store your function in a module called `moves`.

5 Applying Several Moves

The next task is writing a function `move` that takes an input state and a list of moves. It results in a list of states. If the list of moves has n elements, the resulting list of states has $n + 1$ elements. Its first element is the initial state and its last element is the final resulting state.

For example,

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

returns the list of states

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

Structure. Now we are in a position to develop `move`. Approach the task as follows:

- Recursively apply each move in turn.
- When no moves are to be applied, our specification says that the initial state is to be returned (base-case).
- For each move, use `single`.

Please put the function `move` also in the module `moves` (and hence in a file `moves.ex`). Do not forget to convince yourself that your solution works!

6 Finding Moves

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 first wagon `y` from `ys` (the desired train). Our goal is to find a list of moves that takes the wagon `y` in front position on the main track. This is done by the following moves:

1. Split the train `xs` into the wagons `hs` (for head) and `ts` (for tail), where `hs` are the wagons before `y` in `xs`, and `ts` are the wagons after `y` in `xs`.

   Write a procedure `split` that takes a list of wagons `xs` and the wagon `y` and returns the pair `{hs,ts}`.

   For example,

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

   Use the functions `position`, `take`, and `drop`.

2. Move `y` and the following wagons (that is `ts`) on track "one".

3. Move the remaining wagons (that is, `hs`) on track "two".

4. Move all wagons on "one" to "main" (this includes `y`, which goes as needed to the front of "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 both `xs` and `ys` (of course in the new order as they are now on the main track!).

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}]
```

7 Finding Less 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`.

Please store `few` 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}]
```

8 Move Compression

The list of moves computed by `few` 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` 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` that applies rules recursively. Then repeat application of `rules` until the list of moves does not change. Thus, `compress` is implemented as follows:

```
def compress(ms) do
    ns = rules(ms)
    if ns == Ms -> Ms
       true     -> compress(ns)
    end
end
```

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

9 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` and `few` 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}]
```

10 Acknowledgments

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