Streams

Programming II - Elixir Version

Johan Montelius

Spring Term 2021

Introduction

The goal of this assignment is that your should practice working with functions as arguments and return values. It also gives insight into how streams are represented and handled in Elixir.

To begin with we will implement a range in and see how we can represent it in a generic way. We then extend this representation to lazy streams. The interface that we use will be very similar to the `Enumerables` protocol.

1 A range

Let's begin by representing a range of integers. We can of course do this by a list of integers but that wold not be so practical if we are talking about ranges of thousands of integers. Why not simply represent a range as a tuple `{:range, from, to}` where *from* and *to* set the limits of, but are included in, the range. Choosing this representation we continue and implement some functions over ranges.

1.1 sum of all integers

We could then implement a function `sum/1` that could sum all the integers. We don't do a simple calculation but recursively reduce the range to a value.

```
def sum({:range, to, to}) do to end
def sum({:range, from,  to}) do  from + sum({:range, from+1, to}) end
```

We could of course now write arbitrary functions that iterate over the integers in the range but why not write a generic function that works in the same way as `foldl/3`. Let's call it `reduce/3`:

```
  def reduce({:range, from, to}, acc, fun) do
     if from <= to do
       reduce({:range, from+1, to}, fun.(from, acc), fun)
```

```
    else
      acc
    end
  end
```

Summing up all integers is now a one-liner:

```
def sum(range) do reduce(range, 0, fn(x,a) x + a end) end
```

We can even implement a `map/2` function in one line of code. The list of mapped elements might be in the reversed order but we can always reverse the result when we have it.
Implement `map/2`.

1.2 stop half way

The implementation we have gives us the ability to implement arbitrary fold operations over ranges. But what if we want to implement a function that returns the first n elements of a range. We could of course implement is explicitly:

```
def take({:range, _   , _}, 0) do [] end
def take({:range, from, to}, n) do
   [ from | take({:range, from+1, to}, n-1) ]
end
```

This works fine but we might want to implement a more generic function that could do arbitrary things with the first n elements. We can change how `reduce/2` works and use it to stop the reduction when we have found n elements. First we let the second argument signal what should be done. For the regular case we want the reducer to continue the execution but if we tell it to halt it should stop.

```
def reduce({:range, from, to}, {:cont, acc}, fun) do
   if from <= to do
     reduce({:range, from+1, to}, fun.(from, acc), fun)
   else
     {:done, acc}
   end
end
def reduce({:range, from, to}, {:halt, acc}, fun) do
  {:halted, acc}
end
```

We now rewrite the `sum/1` function to use the new protocol.

```
def sum(range) do
   reduce(range, 0, fn (x, a) -> {:cont, x + a} end)
end
```

That did not make things easier but now take a look at this:

```
def take(range, n) do
 reduce(range, {:cont, {:sofar, 0, []}},
    fn (x, {:sofar, s, acc}) ->
      if s == n do
        {:halt, acc}
      else
        {:cont, {:sofar, s+1, [x|acc]}}
      end
    end)
end
```

As long as s is not equal to n we allow the reducer to continue but when we have all the integers that we need we tell it to stop.

Given this implementation the `sum/1` and `take/2` will of course return `{:done, 21}` or `{:halted, [4,3,2,1]}` but this is for you to fix. Now use the extended version of the reducer to implement the following functions.

- A function that returns all integers in the range less than n.
- A function that returns the sum of all integers less than n
- A function that returns the last integer that makes the sum exceed n

It's tricky in the beginning but once you get around thinking about what the function does it becomes easier. Note how much less boiler plate code you have to write. The whole mechanism of how to do the recursion over the range is done by the reducer.

1.3 suspend and continue

Now for the last step to complete the reducer. What if we want to run through the beginning of a range to a certain point as we do in `take/2` but then return not only the result so far but *a continuation*. A continuation would be something that allows us to continue the execution and retrieve for example the following five elements.

We extend the reducer with yet another command `:suspend` that should do the trick. Here we see the power of being able to create a closure. The tuple that we return `{:suspend, acc, closure}` now contains a closure that includes the current range and the function that we have been using.

```
  def reduce(range, {:suspend, acc}, fun) do
    {:suspended, acc, fn (cmd) -> reduce(range, cmd, fun) end}
  end
```

Let's see how we can pick one element at the time from the range.

```
  def head(range) do
     reduce(range, {:cont, :na}, fn (x,_) -> {:suspend, x} end)
  end
```

This looks very strange at first site but it is of course simple once we go through the steps of the evaluation.
Assume we have a range `{:range, 1, 10}` and call our function `head/1`. This will result in a call to:

```
reduce({:range, 1, 10}, {:cont, :na}, fn (x,_) -> {:suspend, x} end)
```

The reducer is asked to make one step and prepared the call to:

```
reduce({:range, 2, 10}, fun.(1,:na), fun)
```

Since we have provided the function this evaluates to:

```
reduce({:range, 2, 10}, {:suspend, 1}, fun)
```

The function reduce will now return what we were looking for:

```
{:suspended, 1,  fn(cmd) -> reduce({:range, 2, 10}, cmd, fun) end}
```

Let's see if this works in practice:

```
> {:suspended, n, cont} = High.head({:range, 1, 10})
{:suspended, 1, #Function<7.2894770/1 in High.reduce/3>}

> {:suspended, n, cont} = cont.({:cont, :na})
{:suspended, 2, #Function<7.2894770/1 in High.reduce/3>}

> {:suspended, n, cont} = cont.({:cont, :na})
{:suspended, 3, #Function<7.2894770/1 in High.reduce/3>}
```

As an exercise you can rewrite `take/2` to return a tuple `{:suspended, elements, continuation}` to be able to take more elements from the range.

1.4 Reduce a list

If you take a look at your functions `sum/1`, `map/2`, `take/2` etc, you will notice that they do not know anything about what a range looks like. The only function that knows what a range look like is `reduce/3`. What if we extend this function so that it could also operate over a lists. If we do this we could use the same code to take element of a list as we use to take elements of a range. It might not be the most efficient solution but we do not have to keep track of if we have a list or a range in our hand.

If you do this right the only thing you need to add are two clauses:

```
def reduce([from|rest], {:cont, acc}, fun) do
    ...
end
def reduce([], {:cont, acc}, _) do
    ...
end
```

Given this we can compose functions that operate over ranges but when we do we need to be careful. Take a look at the two examples below, is there a difference in execution?

```
def take_n_map(range, n, f) do map( take(range, n), f) end

def map_n_take(range, n, f) do taken( map(range, f), n) end
```

The end result should be the same. Taking five elements from a range and apply a function to each of the elements can not be different from applying the function to the elements of the range and then take five elements from the resulting list. The only difference is execution time but this will be quite significant of the range is huge.

We would rather be lazy in our evaluation and not map the function to all elements of the range unless it not required. How do we do this?

The answer is streams.

2 Streams

So now we now what pour task is, we should implement lazy ranges. If you map a function to a range, nothing should happen unless someone requests the first element. Only then should we select the first element from the range and apply the function.

2.1 give me the next element

Take a look at `reduce/3`, where does it actually know what the range look like? It is only in the first clause where i does the pattern matching, checks if

`from` is less than `to` and then increments `from`. What the clause is actually doing is checking if there is another element in the range.

What if we provided another representation of ranges, {`:stream, fun`}, where `fun` is a function that will, when applied to no arguments, return either {`:ok, from, cont`} or `:nil`. We could then extend `reduce/3` to also handle this type of range.

```
def reduce(...    , {:cont, acc}, fun) do
  case ...   do
    {:ok, from, cont} ->
      reduce(... , fun.(from,acc), fun)
    :nil ->
      {:done, acc}
  end
end
```

To see the magic we should now define a stream of fibonacci numbers. Since we define what the next element should look like we are not constrained to sequential ranges. The stream is potentially infinite so we have to care full when we work with it.

```
def fib() do {:steam, fn () -> fib(1,1) end} end

def fib(f1,f2) do
   {:ok, f1, fn () -> fib(f2, f1+f2) end}
end
```

Let's try this an see if it works:

```
>  High.take(High.fib(), 5)
```

Magic?

2.2 we're not done yet

Let's implement a range as a lazy stream. We can now use it do do some experiments.

```
def range(from, to) do
  {:stream, fn() -> next(from, to) end}
end

def next(from, to) do
  if from <= to do
    {:ok, from, fn() -> next(from+1, to) end}
```

```
    else
      :nil
    end
  end
```

This call below works but it might not do exactly what we want it to do. The reason is that our `map/2` function will happily generate a list of ten integers that is then passes to `take/2` that only picks the first five. We want `map/2` to return a lazy stream.

```
take( map( range(1,10), fn(x) -> x + 1 end), 5)
```

Let's change out map function so it becomes aware of the difference between ranges and lazy streams. We do not have to do a lot to make it work. Add this clause as the first clause of `map/2`:

```
  def map({:stream, next}, fun) do
    {:stream,
     fn () -> case ...   do
                  {:ok, from, cont} ->
                    {:ok, ... , cont}
                  :nil ->
                    :nil
              end
      end
    }
  end
```

There is of course a lot more to be said about the *Enum* and *Stream* libraries in Elixir and of course the implementation of ranges and streams are more complex than in this tutorial but I hope that you now have a better understanding of how they work.

I also hope that you see how higher order programming can be used to implement some rather complex data structures in a not so complicated way.