Streams

Johan Montelius

KTH

VT24

an infinite list

```
inf = infinity(); [0|inf] = inf.(); [1|inf] = inf.()

def infinity() do
  fn() -> infinity(0) end
end

def infinity(n) do
  [...|...]
end
```


the list of fibonacci

A function that returns an infinite list of Fibonacci numbers.

```
def fib() do
  fn() -> fib(1,1) end
end

def fib(f1, f2) do
  [f1 | fn() -> fib(f2, f1+f2) end]
end
```


a lazy list from 1 to 10

Let's represent a *range* of integers from 1 to 10 as:

```
{:range, 1, 10}
```

Elixir gives us a syntax for this:

```
1..10
```

But we will do our own :-)

This is not how Elixir represents it but it's fine for now

sum/1 of a range

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

```
def sum(range) do  sum(range, 0) end
```

```
def sum({:range, to, to}, acc) do ... end
def sum({:range, from, to}, acc) do sum({:range, ..., to}, ...) end
```


foldl/3 on a range

How do we fold-left on a range:

```
foldl({:range, 1, 5}, 0, fn(x,a) -> x + a end)
```


sum/1 of a range

```
def sum(range) do foldl(range, 0, fn(x,acc) -> x + acc end) end
```


map/2 on a range

How do we map on a range (let's forget the order):

```
map({:range, 1, 5}, fn(x) -> x + 1 end)
```

should we return a list of values or .... a modified range?

filter/2 on a range

How do we filter a range (again, f*ck the order):

```
filter({:range, 1, 5}, fn(x) -> rem(x,2) == 0 end)
```

should we return a list of values or ....

take/2 on a range

How do we take n elements from a range (order ... not):

```
take({:range, 1, 1_000_000}, 5)
```

we don't want to build a list of a million integers

reduce/3: the goto of all

```
def sum(r) do
  reduce(r,  0, fn(x,a) -> x+a end)
end
```

Our `reduce/3` *should work as* `foldl/3` *(left to right, tail recursive).*

```
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
```

... we're not done!

How do we stop in the midle?

Implement take/2 using reduce by .....

We need to control the reduction.

continue

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

The accumulator is both a value and an instruction to continue.

stop in the midle

```
  :
def reduce(_, {:halt, acc}, _fun) do
  {:halted, acc}
end


def take(r, n) do
  reduce(r, {:cont, {:sofar, n, [] }},
    fn(x,{:sofar, n, a}) ->
      if n > 0 do
        {:cont, {:sofar, n-1, [x|a]}}
      else
        {:halt, [x|a]}
      end
    end)
end
```


suspend in the midle: head and tail

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

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


Elixir libraries

- `List` : operatates on lists, returns a list or some value.
- `Enum` : takes an `Enumerable` as argument, returns a list or value.
- `Stream` : takes an `Enumerable` as argument, returns an `Enumerable` or value.

A datastructure is Enumerable *if it implements the* enumerable protocol. *Lists and ranges are* `Enumerable`.

Summary

- range: representation of a range of integers
- streams: lazy evaluation of sequences