Asynchronous vs Synchronous

Johan Montelius

KTH

VT23

let's implement a memory

The *memory* should be a zero indexed *mutable data structure* of a given size and provide the following functions:

- `@spec new([any]) :: memory` : creates a memory initialized with values from a list
- `@spec read(memory, integer) :: any` : returns the value stored in the memory at position *n*
- `@spec write(memory, n, any) -> :ok` : writes the value at position *n* in the memory

what do we mean by read and write

a first try

```
defmodule Mem do

  def new(list) do
    {:mem, List.to_tuple(list)}
  end

  def read({:mem, mem}, n) do
     elem(mem, n)
  end

  def write({:mem, mem}, n, val) do
      ???
  end
```

... hmm, not that easy

the functional way

Functional programming rules!

Let `write/3` return a new memory, a copy of the original with the update.

```
def write({:mem, mem}, n, val) ->
   {:mem, put_elem(mem, n, val}}
end
```

I love functional programming

life is easy

```
def test() do
  mem1 = Mem.new([:a, :b, :c, :d, :e, :f])
  mem2 = Mem.write(mem, 3, :foo)
  take_a_look_at_this(mem1)
  and_check_this(mem2)
end
```

What if we always write like this:

```
def test() do
  mem = Mem.new([:a, :b, :c, :d, :e, :f])
  mem = Mem.write(mem, 3, :foo)
  take_a_look_at_this(mem)
  and_check_this(mem)
end
```


why not cheat

Can we cheat, and introduce a mutable data structure?

Can we use processes to implement mutable data structures?

a mutable cell

cell/1

```
def cell(v) do
  receive do
    {:read, pid} ->
       send(pid, {:value, v})
       cell(v)
    {:write, w, pid} ->
       send(pid, :ok)
       cell(w)
    :quit ->
       ok
  end
end
```


a synchronous interface

read/1

```
def read({:cell, cell}) do
  send(cell , {:read, self()})
  receive do
    {:value, v} ->
      v
  end
end
```

write/1

```
def write({:cell, cell}, val) do
  send(cell, {:write, val, self()})
  receive do
    :ok ->
      :ok
  end
end
```

the cell module

```
defmodule Cell do

  def start(val) do       ## things to do in mother process
    {:cell, spawn_link(fn() -> init(val) end)}
  end

  def read({:cell, cell}) do  ... end

  def write({:cell, cell}, V) do ...  end

  def quit({:cell, cell}) do ...  end

  def init(val) do        ## things to do in the child process
     cell(val)
  end
```


the memory

```
defmodule Memory do

def new(list) do
  cells = Enum.map(list, fn(v) -> Cell.start(v) end)
  {:mem, List.to_tuple(cells)}
end

def read({:mem, mem}, n) do
  Cell.read(elem(mem, n))
end

def write({:mem, mem}, n, val) do
  Cell.write(elem(mem, n), val)
end

def delete({:mem, mem}) do
  Enum.each(Tuple.to_list(mem), fn(c) -> Cell.quit(c) end)
end
```


at last

```
test() ->
    array = Memory.new([:a,:b,:c,:d,:e,:f,:g,:h])
    Memory.write(array, 5, :foo)
    Memory.write(array, 2, :bar)
    love_all(array)
    sort_it_for_me(array)
    i_love_imperative_programming(array)
 end
```


functional vs processes

By extending our language to handle processes, we have left the functional world.

We can implement *mutable data structures*, something that we agreed was evil.

Why are mutable data structures evil?

the evil of mutability

```
  :
this = check_this(mem),
%% I hope it did not change anything
that = check_that(mem),
  :
```


truly bad

all_work/2

```
def all_work(cell, 0, jack) do
  send(jack, :run)
end
def all_work(cell, n, jack) do
  x = Cell.read(cell)
  Cell.write(cell, x + 1)
  all_work(cell, n-1, jack)
end
```

no_play/1

```
def no_play(n) do
   cell = Cell.start(0)
   me = self()
   spawn(fn() ->
      all_work(cell, n, me) end)
   spawn(fn() ->
      all_work(cell, n, me) end)
   receive do :run ->
        receive do :run ->
             Cell.read(cell)
        end
   end
end
```


a dull boy

- without mutable data structures, concurrency would be easy
- sharing mutable data structures is the root of all evil
- a process is, in one way, a mutable data structure
- ... It's only a movie.

adding atomic operations

```
def cell(v) do
  receive do
    {:read, pid} ->
       send(pid, {:value, v})
       cell(v)
    {:write, w, pid} ->
       send(pid, :ok)
       cell(w)
    :quit ->
       ok
  end
end
```

```
def cell(v) do
  receive do
    {:read, pid} ->
       send(pid, {:value, v})
       cell(v)
    {:write, w, pid} ->
       send(pid, :ok)
       cell(w)
    {:inc, n} ->
       send(pid, :ok)
       cell(v+n)
    :quit ->
       ok
  end
end
```

a lock

We want to avoid processes interfering with each other when intracting with a process.

Let's implement a lock using our cell.

- take the lock
- relase the lock
- at most one process can hold the lock

using a cell

```
def critical(danger, lock) do
  case Cell.read(lock) of
    :locked ->
        critical(danger, lock)
    :free ->
        Cell.write(lock, :locked)
        do_it(danger)
        Cell.write(lock, :free)
  end
end
```

hmmm, not so good

a better lock

```
defmodule Lock do

def start() do
   {:lock, spawn_link(fun() -> free() end)}
end
```

```
def free() do
  receive do
    {:take, pid} ->
       send(pid, :taken)
       taken()
  end
end
```

```
def taken() do
  receive do
   :release ->
     ...
  end
end
```


asynchronous communication

Messages in Elixir/Erlang is a form of *asynchronous communication*.

```
send(pid, {:take, self()})
```

The sender does not block and wait for the receiver to accept the message.
Asynchronous : not at the same time

synchronous communication

We can implement *synchronous communication*

```
def take(lock) do
  send(lock, {:take, self()})
  receive do
    :taken ->
       :ok
  end
end
```

```
def free() do
  receive do
    {:take, pid} ->
       self(pid, :taken)
       taken()
  end
end
```

The reply is generated by the application layer.

```
:ok = take(lock)
```

The application process sees a *synchronous operations*,

Synchronous : at the same time

pros and cons

What are the pros and cons of asynchronous communication?

synchronous by default

We could provide synchronous communication by default, for example:

```
send(jack, {:take, this})
%% I now know that the message is in the queue of Jack
  :
  ?
```

... or

```
send(jack, {:do, that})
%% I now know that the message has been "received" by Jack
  :
  ?
```

but why not..

synchronous when needed

```
send(jack,  {:hello, self()}
%% I need to know that Jack has fun
receive
  :having_fun ->
     run_as_hell()
end
```

an asynchrounous memory

Synchronous programing is boring.

```
def cell(v) do
  receive do
    {:read, ref, pid} ->
       send(pid, {:value, ref, v})
       cell(v)
    {:write, w, ref, pid} ->
       send(pid, {:ok, ref})
       cell(w)
    :quit ->
       :ok
  end
end
```


an asynchrounous memory

```
def redrum({:cell, cell}) do
  ref = make_ref()
  send(cell, {:read, ref, self()})
  ref
end

def murder(ref) do
  receive do
   {:value, ^ref, value} ->
      value
  end
end
```

```
  :
ref = redrum(cell)
  :
  :
val = murder(ref)
  :
```


Summary

- Processes can be used to implement mutable data structures.
- Same problems needs to be considered.
- Made easier since each mutable data structure is a user defined process.
- Asynchronous vs synchronous message passing - pros and cons.