An Introduction to the Elixir Programming Language

Tyler M. Masthay
Department of Computer Science
University of Dayton
Dayton, Ohio 45469–2160 USA
tmasthay1@udayton.edu

ABSTRACT

We provide a brief introduction to the Elixir programming language and its use of the *Actor* model of concurrency.

KEYWORDS

Actor model of concurrency, Actors, asynchronous communication, concurrent programming, functional programming, mailbox, message passing, pattern matching, synchronization.

Tyler M. Masthay. 2017. An Introduction to the Elixir Programming Language. *CPS 499-03: Emerging Languages, University of Dayton, Ohio 45469–2160 USA, Spring 2017,* 3 pages.

1 INTRODUCTION

Elixir is a concurrent, functional programming language that was designed for implementing distributed, fault-tolerant systems. It was built by José Valim who created the language to address the omission of critical features in Erlang. First, Erlang lacks much support for ease of compilation management. In Elixir, the built-in command `mix` auto-generates a directory that creates a library, test, and configuration directory to build projects easily. Second, Elixir uses a construct called protocols to facilitate polymorphism and metaprogramming. However, Elixir is still mainly designed for concurrent programming like Erlang. Elixir programs compile into bytecode for the Erlang Virtual Machine and, thus, the language closely resembles Erlang. Its functional nature promotes concurrent programming that has no shared state, thus lowering the overhead of critical sections and possibility of race conditions.

2 ELIXIR ESSENTIALS

The concept of pattern-matching is used extensively in Elixir. For example, if we set `i = 1`, then we are just binding 1 to said variable. However, the statement `1 = i` is legal because both sides of the equal sign are 1. Furthermore, `2 = i` will result in a `no match of right hand side value: 1` error because the two patterns do not match. Also, the language mostly prevents the programmer from producing programs with side-effect. For instance, given the following two lines of code, Elixir creates another name for `i` internally so that the identifier is only bound once [4]:

```
i = 1
i = 2
```

Pattern matching can be used in ways similar to Haskell, ML, and LISP, with familiar list operators ++ (append), `--` (list difference), and `|` (cons). As with other functional programming languages, functions in Elixir are first-class. Closures are supported in Elixir through modules. The keyword `defmodule` is used to define a module. The syntax for a module is `defmodule myModule do (...) end`. Table 1 provides an overview of some features of Elixir vis-à-vis Haskell and LISP. For more information regarding Elixir syntax, see [1].

Table 1: Overview of some Elixir concepts and syntax.

Feature		Resemblance
Pattern matching	√	Haskell (syntax)
Heterogeneous lists	√	LISP, Haskell (syntax)
Lambda expressions	√	Haskell (arrow syntax)
Type inference	√	Haskell

3 THE ACTOR MODEL OF CONCURRENCY

Elixir uses the *Actor* model of concurrency. Models of concurrency vary in their level of abstraction. However, many concurrency models "can be shown to be special cases of actors" [3]. In the Actor model, an actor has a mailbox in which to receive messages. The primitive actions of an actor are to (a) create more actors, (b) send messages to other actors, and (c) decide what to do with the next message that it receives from another actor. In Elixir, action (a) is achieved through the function `spawn`. Action (b) is achieved with the function `send`. Action (c) is achieved with the `receive` construct. This model promotes thinking of one message being passed and/or processed at a time. Figure 1 illustrates the uses of these functions and construct.

4 A COUNTER EXAMPLE

The following is an example of message passing within a `Counter` module.

```
defmodule Counter do
  def loop(count) do
    receive do
      {:next} ->
        IO.puts("Current␣count:␣#{count}")
        loop(count + 1)
    end
  end
```

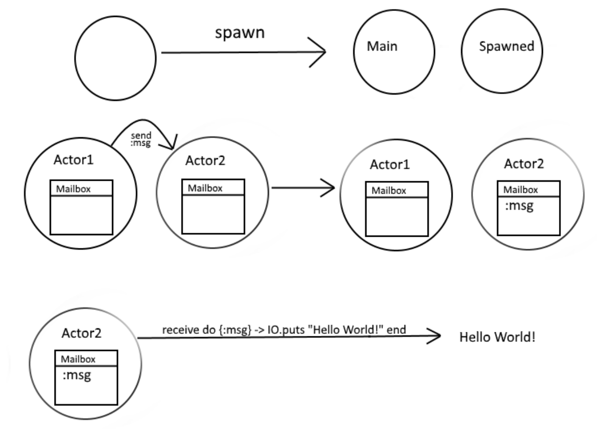

Figure 1: The three primitive actions of an actor.

```
end
```

Note that although we are using recursion, Elixir automatically implements tail recursion so that the stack remains a constant size. Let us make two different counter variables and run the following:

```
$ iex Counter.ex
counter1 = spawn(Counter, :loop, [1])
counter2 = spawn(Counter, :loop, [100])

send(counter1, {:next})
send(counter1, {:next})
send(counter2, {:next})
send(counter2, {:next})
send(counter1, {:next})
send(counter2, {:next})
send(counter1, {:next})

Current count: 1
Current count: 100
Current count: 101
Current count: 2
Current count: 102
Current count: 3
Current count: 4
```

Thus, the output is unpredictable (i.e., a race condition exists), showing us that the passing is done *asynchronously*. This is what is meant by Hewitt's term *decoupled* regarding the relationship between actors. The actors execute independently, and the only way to get them to cooperate according to some synchronization protocol is to pass messages between them. Messages in the mailbox of an actor are processed sequentially (i.e., in FIFO order). To get the two counters to print in the proper order, we need to pass messages *between* the two actors. Thus, we redefine our counter module to have two different counters:

```
defmodule Counter do
  def firstCounter() do
    pid = spawn_link(__MODULE__, :list1, [1])
    Process.register(pid, :first)
    pid
  end
  def secondCounter() do
    pid = spawn_link(__MODULE__, :list2, [10])
    Process.register(pid, :second)
    pid
  end
end
```

The code above says "when we create the first counter, call the function `list1` with an argument list of 1 and register a process identifier of `:first` with the process we just spawned." The analogous statement is true for the second actor. This must mean that we need to define `list1` and `list2`, leaving us with the following final version of the program:

```
defmodule Counter do
  def firstCounter() do
    pid = spawn_link(__MODULE__, :list1, [1])
    Process.register(pid, :first)
    pid
  end
  def secondCounter() do
    pid = spawn_link(__MODULE__, :list2, [1000])
    Process.register(pid, :second)
    pid
  end

  def list1(currVal) do
    receive do
    {:go} ->
      IO.puts currVal
      :timer.sleep(1000)
      send(:second, {:go})
    end
    list1(currVal + 1)
  end
  def list2(currVal) do
    receive do
    {:go} ->
      IO.puts currVal
      :timer.sleep(1000)
      send(:first, {:go})
    end
    list2(currVal + 1)
  end
end
```

The `list1` function blocks until it receives a `:go` message, prints its current value, sleeps for one second, sends a message back to the second counter actor, recursively calls itself with an incremented argument, and waits. Meanwhile, when the second counter receives the `:go` message, it performs the same actions, producing the desired behavior. This is a rudimentary example of message passing in Elixir. Many other examples can be found in [2, Chapter 5] and [5].

5 OTHER MODELS OF CONCURRENCY

In comparison to other concurrency models, consider the code within `receive` clauses for the `list1` and `list2` functions above. How can we justify Hewitt's statement that other concurrency models such as semaphores are special cases of actors? Consider the `list1` function. Its `receive` clause says 'block until a `:go` message is received.' This is similar to (e.g. in Java) declaring a condition variable named go and using `wait` and `signal` methods to communicate between threads. However, condition variable objects must be declared. Message-passing is a high-level synchronization primitive. Semaphores could also be demonstrated by modifying the above module to have a shared variable (e.g., `i`). We could protect this shared variable by a message that indicates which of the two counters can access it. Finally, again with a shared variable `i`, a Hoare monitor will have been implemented in this code. The message `:go` acts like a locking mechanism so that `list1` and `list2` are not executing their main code concurrently. Thus, the Actor model generalizes the three aforementioned approaches to concurrency. Thus, we can simulate condition variables, semaphores, and monitors with actors. Note, however, that this counter program does not involve shared state—Elixir's use of the Actor model provides an elegant means of producing concurrent programs with no shared state. Since most race conditions occur due to mutation of shared state, Elixir is an attractive language for concurrent programming.

6 EXERCISES

The following are some programming exercises that incorporate some essential Elixir concepts:

(1) Modify the program `Counter2.ex` so that a private member variable `currCount` is printed and modified.
(2) Implement a module for a binary tree closure that is purely functional. Use the following representation: a leaf node is a tuple of the form `{:leaf, val}` where `val` is the value associated with this tree and an internal node is of the form `{:internal, left, right}` where `left` and `right` are the associated subtrees. Ensure that polymorphic trees are supported. For example, the tree `{:internal, {:leaf, 1}, {:leaf, fn(x) -> x}}` should be supported by the module.
(3) Use the binary tree from exercise 2 to implement the `foldl` and `foldr` functions from the Haskell language.

REFERENCES

[1] Elixir. Available: http://elixir-lang.org [Last accessed: 29 March 2017]. (????).
[2] P. Butcher. 2014. *Seven Concurrency Models in Seven Weeks: When Threads Unravel.* The Pragmatic Bookshelf, Dallas, TX.
[3] C. Hewitt, P. Bishop, and R. Steiger. 1973. Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Advance Papers of the Conference*, Vol. 3. Stanford Research Institute, 235.
[4] S.L. Salas and E. Hille. 2016. *Erlang and Elixir for Imperative Programmers.* APRESS, New York, NY.
[5] B.A. Tate. 2014. Elixir. In *Seven More Languages in Seven Weeks: Languages That Are Shaping the Future*, B.A. Tate, F. Daoud, I. Dees, and J. Moffitt (Eds.). The Pragmatic Bookshelf, Dallas, TX, Chapter 4, 125–169.