# Erlang and Elixir

The Bread and Butter of Concurrent Software

Jarred Parr

December 2018

## 1 Abstract

The purpose of this paper is to discuss the power of reliability, concurrency, and fault tolerance that Erlang and Elixir provide. This paper will primarily focus on the language features of Erlang which forms the basis of Elixir's core functionality, but it will also provide a breif exposé into the features of the more modern language. The primary focus is to outline to data abstractions, control abstractions, and support for other paradigms.

## 2 Introduction

Erlang is a distributed, fault-tolerant, and highly available programming languages used in all manner of applications in which reliable and fast messaging is needed. Rising to fame as being the backbone of much of the telecom industry, erlang also has found home in a variety of high-profile applications and companies. It was designed with the goal of improving existing telephone infrastructure. The initial version was developed in Prolog, but was found to be too slow and in 1992 the BEAM virtual machine was developed which compiled Erlang to C code [1]. Erlang supports the functional programming format and follows it very closely. This is in an effort to prioritize reliability over all else. Joe Armstrong, the creator of the language, said in a 2013 interview Ïf Java is write once , run anywhere, then Erlang is write once run forever.̈ This is because of the highly functional backing that is so strongly enforced by the language. It makes reliability its biggest concern. Well written Erlang code should be able to run indefinitely. Erlang has been in use since the 90's, but in recent years companies like WhatsApp, Facebook, Instagram, and Discord all rely on erlang to run their chat services and other real time concurrent features of the platform.

Each of which has been able to scale to over a billion concurrent users on their platform with relative ease and extremely high reliability.

## 3 Data Abstractions

Erlang features many of the datatypes that are commonly seen in other C-style imperative programming languages, however it also features a lot more that help it be so good for concurrency. Erlang introduces a lot of new nomenclature to support these new types it introduces. The first of which are terms. Terns are defined as being any piece of any data type. We also habe standard number types like literals, integers, and floats. Erlang also has two specific notations: $\$char$ which is the ASCII value of a character char and $base\#value$ which is an integer with base $base$. It must be an integer in range 2 - 36. Erlang also features Atoms which are literals. These value types are enclosed in single quotes like $'name'$ if they do not begin with a lower-case letter or contains characters that are not _ or @. The last two are pretty specific to erlang, port identifiers and pid's. Port identifiers specify erlang ports and pid's are used to identify processes that are spawned by the erlang runtime.

Erlang is dynamic through and through. It deatures dynamic type typing and dynamic type checking. The original authors came from dynamically typed languages and, as a result, Erlang models that same sentiment. Erlang is not an explicitly typed language and can infer the typings on runtime instead of needing them to be defined beforehand. As a result, a lot of type coercion occurs as the result of casting into new types

## 4 Control Abstractions

Erlang has a lot of ways to do control abstraction and these are in large part because of its non imperative design. Common control structures that can be expected in C-style languages are not seen in the same way in Erlang. This is because of two reasons: one, it's (almost) purely functional, two, it's based off of prolog, not C.

Expressions are evaluated in a bottom-up approach. All subexpressions are evaluated before the expression itself is evaluated, This is always the case unless explicitly stated otherwise. For example, a simple addition expression of the following form:

$$expr1 + expr2$$

*expr*1 and *expr*2 are expressions as well as the whole expression adding them together. Both expressions, *expr*1 and *expr*2 are evaluated first - in any order - before the addition is performed [1]. Precedence is determined by how the expression is structured. In most cases individual statements are handled randomly like they are in standard imperative languages, however, in cases where compound statments exist, the individual operations are executed before the compond operation. In general, scoping is handled almost the exact same as C and other imperative programming languages. However, since Erlang doesn't really use class structures, the encapsulation of data is preserved in the design of functions and scoping of variables inside of those functions.

Erlang also features full module support but no idea of namespaces. All declared modules exist in the global scope and it is up to the developer to properly handle the encapsulation of important data objects. Thankfully, the functional nature of the code prevents much of the associated problems that could be surmised from hearing this. This is because the functions work heavily off of returned values, it's one of the reasons that every if statement requires an else, something MUST always be returned from the function execution. This provides the assurance of the functionality and the integrity of the data as it moves between modules. In the event that code execution is unpredictable, erlang features a robust exception system with about 20 built in exception types. It also has suppport for standard try-catch blocks as well. This allows for code to be even more reliable as Erlang can throw errors without breaking the code at runtime. Erlang can also be dynamically restarted allowing for the same codebase to continuously run forever even though changes are actively being made to that same code. This ties back to the r̈un foreverïdea. Erlang code is made to start running and never stop unless there is a severe error. This makes things great for handling the new updates that code changes and bring without needing to take a web service down for maintainence.

## 5 Onward To Elixir

So, if elixir exists, does erlang even matter? The short answer is yes, the longer answer is that it depends on the use case when you need one or the other. Elixir brings a modern, ruby-like syntax to the erlang ecosystem and serves as a superset of Erlang. It brings robust new features but still maintains 100% backward compatibility with the erlang stack. The cases where elixir can't be used is when interfacing with existing code. You can

still communicate with it through elixir, but knowledge of erlang must also exist first to allow for an understanding of what is going on with the erlang code on the deeper level. That's really it. Erlang and Elixir are both useful and beautiful solutions to whatever problem needs to be solved, it really just depends on what the user needs to get done and how old the codebase is.

## 6 Code Sample

The code provided is a representation of elixir. It made the most sense to use this language since it is not only significantly easier to read at first glance, but also compiles fully down to barebones erlang code. It would be a travesty to not show off the beautiful syntax and modenization attempts that have been made with the design of this new language format.

---

```
defmodule Calc do
  def add(a, b) do
    a + b
  end

  def subtract(a, b) do
    a - b
  end

  def multiply(a, b) do
    a * b
  end

  def divide(a, b) do
    cond do
      b == 0 -> 0
      b != 0 -> a / b
    end
  end

  def show_total(lst) do
    Enum.sum(lst)
  end
```

```
  def calculate(line, prev) do
    if line[0] == "quit" do
      show_total(prev)
    else
      output = 0
      output =
        case line[3] do
          {:ok, "+"} -> add(line[0], line[1])
          {:ok, "-"} -> subtract(line[0], line[1])
          {:ok, "*"} -> multiply(line[0], line[1])
          {:ok, "/"} -> divide(line[0], line[1])
        end
      Enum.concat(prev, output)
    end
  end

  def parse(line) do
    vals = String.split(line, " ", trim: true)
    vals
  end

  def sub_main(prev) do
    input = IO.gets("> ")
    input
    |> parse
    |> calculate(prev)
    |> IO.puts
    sub_main(prev)
  end


  def main() do
    prev = []
    sub_main(prev)
  end

end
```

## 7 Conclusion

Overall, erlang and elixir are very powerful for concurrent systems programming. The actor pattern and fully functional design make it a joy to solve problems in a unique way. The languages offer robust reliability and readability in a modern package, and have also been shown in several situations to be the no-brainer choice for handling any highly distributed systems engineering and messaging. The sheer reliability of the languages shows that they will without a doubt be here to stay. I had an absolutte pleasure digging into the language and all of its quirks, and it think it helped me understand concepts of recursion and truly stateless programming like I have never seen before. I often times find myself applying such ideologies to my imperative programming because of the ease of reading and incredible reliability that it introduces with such predictable interactions.

## References


[1] The Erlang Creators Documentation *http://erlang.org/doc/index.html*. 1988

[2] Erlang Tutorial on Tutorialspoint *https://www.tutorialspoint.com/erlang/*. Tutorialspoint Inc, Madhapur, Hyderabad, India, 500081

[3] Fred Hebert *http://learnyousomeerlang.com*. No Starch Press, San Francisco, CA, 2011.

[4] Joe Armstrong *The History of Erlang*. Proceedings of the third ACM SIGPLAN conference on History of programming languages, 2007

[5] The Wikimedia Foundation *Erlang (programming language)*. https://en.wikipedia.org/wiki/Erlang_(programming_language)