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

References

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

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

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

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