Trey Phillips

Dr. Angela Guercio

CS 33101

3 December 2019

Erlang

Functional programming is one of the oldest paradigms of computer science, and thus has a rich history within the field. Based on lambda calculus, it has shaped the trajectory of modern programming ideologies, most notably through languages such as FORTRAN, LISP, Haskell, and Algol 60 (Turner). During this prominence of functional languages in the mid to late 20th century, a language known as PLEX was being used by the company Ericsson for telephony applications (“Inside Erlang…”). This language had become somewhat obsolete due to some advancements in the field. A team led by the newly hired Joe Armstrong set out to revise the language and revitalize its usefulness in the industry. With this endeavor, the language known as Erlang was born. Along with his peers Robert Virding and Mike Williams at Ellemtel Telecommunication Systems Laboratories, he finished the language relatively quickly and established Erlang as proprietarily active within Ericsson in 1986 (“Erlang Programming Language Profile”). Erlang was used to assist with telecommunication systems, and it became used in various applications within Amazon, Facebook, Nortel, T-Mobile, and the mobile WhatsApp following its open source release in 1998 (“Inside Erlang…”). Besides being used in current day applications, Erlang has also influenced other languages, most notably Elixir (“Scoping Rules…”) and Rust (Thokala). With these pieces of information mind, it is plain to see that Erlang will hold its place in history among functional languages.

The language is an interesting one for a myriad of reasons. Erlang is a functional programming language (“Erlang Programming Language Profile”), with a focus on “concurrent processes, scheduling, memory management, distribution, networking, etc.” (Armstrong et al.). For these reasons, it makes sense that companies would choose Erlang for important projects involving telecommunications: its potential for robust, parallel processes helps with preventing massive failures in important business sectors. What adds to this ideology is that Erlang is strongly and dynamically typed (“Erlang Tutorial”). Through the implementation of the language, type errors are always found without ever needing to specify the type, increasing readability and reliability. Part of this is aided by the fact that assignments, or rather bindings, for variables happen at declaration only; a variable’s value cannot change unless it is deallocated. Along with this, since typing is dynamic and value bindings happen once per variable, coercion is not a part of the language and explicit conversions are required despite the type checking occurring at runtime (Trottier-Hebert). Erlang was developed with the expectation that errors are going happen regardless of the care put into coding a particular application and seeks to intervene in that likely case, saving crucial technologies from catastrophe.

The language itself is apparently functional when the syntax and semantics are looked over. Generally speaking, functions and statements are expressed in this format:

FunctionName (Pattern1… PatternN) ->

Statement1,

Statement2,

…

StatementM.

where each Nth argument is a pattern or function established by the programmer and there are M statements (“Erlang Tutorial”). It is worth noting that because this language is also based on lambda calculus, everything is a function, including numbers. Notice also that each sequential statement is separated by a comma and the very last statement is delimited by a period. This layout is how every section of code is expressed, even the main part of the program, ‘start()’. However, every function must be treated using the embedded ‘export’ function like so:

-export([FunctionName1/N1,…, FunctionNameK/NK]).

This is basically exporting the header for the function so that it may be used within the program. Again, this must happen for ‘start()’. In order to import a function from a separate module, a program must have the following statement:

-import(ModuleName, [FunctionName1/N1,…, FunctionNameK/NK])

This allows for procedural abstraction via separation of program parts.

As far as syntax goes, the most basic operators are pretty straightforward. Comments can be made using the character ’%’ (“Erlang Tutorial”). You specify variable values at declaration with ‘=’. Arithmetic operators include ‘+’, ‘-‘, ‘\*’, and ‘/’; however, for modular expressions with division, ‘rem’ and ‘div’ give the remainder and whole quotient respectively. Unary operators like ‘++’ and ‘—’ are also available, but are right associative. Relational operators include ‘==’, ‘<’, ‘>’, ‘>=’, and ‘/=’, which tests for the difference between two values and will have a truth table similar to XOR. ‘or’, ‘and’, ‘not’, and ‘xor’ constitute the logical operators. To specify bitwise operators, merely add the letter ‘b’ in front of a logical operator. For example, ‘or’ would become ‘bor’. All operators are left associative unless the precedence is altered with parenthesis.

Some constructs are available with the language, and they come with their own syntaxes. For example, the included if-else construct includes the following syntax for N conditions:

if

Cond1->ReturnValue1;

Cond2->ReturnValue2;

…

CondN->ReturnValueN

end

Of course, the delimiter would be after the ‘end’ (“Erlang Tutorial”). A similar construct is case statements, which differ in that the ‘if’ is instead ‘case an\_expression of’ and each result is a statement rather than a return value. Another construct offered with the language, which makes it so powerful, is a guard. These are added after function headers so that functions may be defined for varying circumstances concerning their parameters. It is also responsible for how recursion is properly implemented, which is a staple of functional programming. Multiple guard conditions may be used, separated by commas, and are applicable to if and case statements both. Additionally, because of the intent of the language, try and catch constructs exist in a similar format to case statements. Both constructs are one way in which blocks are used in the language. Lastly, loops and jumps are not included in such a functional language.

Variables are treated in a unique was in Erlang. As stated previously, bindings only happen at declaration; they are dynamic bindings where allocation is accomplished implicitly to a heap (Thokala). For data types, it has a very limited embedded selection. This would include integers, floats, lists, atoms, tuples, Booleans, bit strings, records, and maps, which are Erlang’s version of dictionaries (“Erlang Tutorial”). Additionally, it has support for unions, string literals, and user-defined types (Lindahl and Sagonas). Names for variables are limited in that they must start with a capital or underscore; however, variables whose names begin with underscore, denoting an anonymous variable, have the added property that their values can be ignored (“Erlang Reference Manual”). Atoms are special in that they are “to be enclosed in single quotes (') if (they) does not begin with a lower-case letter or if (they) contain other characters than alphanumeric characters, underscore (\_), or @," giving atoms added flexibility. Pointers are not permitted in Erlang, but you may undermine this restriction using bit strings and interpreting the binary value as any other data type (Scalas).

Functions are, as its paradigm suggests, extremely important to Erlang’s usability. Because of this, subprograms are allowed as additional functions. The scope within these functions, and any block of expressions for that matter, is static (“Scoping Rules…”). Actual polymorphism is offered for functions in Erlang through function specifications and can be expanded using guard statements (“Erlang Reference Manual”). Each function will have its own block plus any additional if, case, or try-and-catch blocks within the function body. With the exception of the bit string loophole described earlier, which could be argued to be pass by reference, all parameter passing in Erlang is pass by value, further adding to the language’s robustness (O’Keefe). Functions can of course call other functions, as most everything in Erlang is considered a function, and otherwise recursion could not occur, preventing effective use of the functional language paradigm. For the boasted concurrency of the language, a programmer would use the concept of a process in code by using the ‘spawn()’ method.

Erlang has some other tools that make the mathematical language even more practical. For example, the trigonometric functions, among other mathematical functions, are included in the base language (“Erlang Tutorial”). The language also had embedded methods for file stream. Additionally, anonymous functions and variables are built in and add in just a bit more imperative functionality. Since it was built with telecommunications in mind, it makes sense that the language would include methods for inter-program relaying. Methods for email, databases, and ports allow this extension for distributed programming where concurrently run programs may send each other mid-process messages.

Using this language is as possible as other languages given the tools that Ericsson provides. Erlang is often distributed with the Open Telecom Platform (OTP) as a sort of shell and operating system for application development (*Erlang/OTP…*). This combination is often referred to as Erlang/OTP. This application can be integrated with command terminals or IDEs, but is almost necessary for programming with Erlang (“Erlang Tutorial”). However, OTP offers increased optimization of the programs created with the language, which is an added benefit of using this standard.

The creators of Erlang set out to create a functional language that would seldom fall under intense use. To achieve this goal, they carefully considered each attribute of general programming languages. They borrowed from the running tradition of functional programming languages at the time, especially PLEX. The strong, dynamic type system gave their creation increased usability in the target industry and others. The essentials of the language, such as functionality, tolerance, and concurrency, were prioritized in the implementation, syntax, and semantics. The data types, structures, constructs, and additional language tools add to the utility of Erlang as well. With all of these elements and how they work together in mind, it is no wonder that the language by Ericsson has maintained its place in the industry for so long.

Example Code

-module(main).

-export([factHelp/2,fact/1,start/0]).

factHelp(N,Res) when N==0 ->

Res;

factHelp(N,Res) when N>0 ->

factHelp((N-1),(N\*Res)).

fact(N) ->

factHelp(N,1).

start() ->

{ok,[N]}=io:fread("enter>","~d"),

io:fwrite("~w",[fact(N)]).

References

Armstrong, Joe, Bjarne Däcker, Thomas Lindgren, and Håkan Millroth. “Open-source Erlang – White Paper.” Ericsson product team, Version 2.0, Ericsson, web.archive.org/web/20111025022940/http://ftp.sunet.se/pub/lang/erlang/white\_paper.html

*Erlang/OTP System Documentation 10.5*. Version 10.5, Ericsson AB, 17 Sept. 2019, https://erlang.org/doc/pdf/otp-system-documentation.pdf

“Erlang Programming Language Profile.” *OSS Blog*, www.ossblog.org/erlang-programming-language-profile/

“Erlang Reference Manual User’s Guide.” *Erlang*, Ericsson AB, 17 Sept. 2019, erlang.org/doc/reference\_manual/users\_guide.html

“Erlang Tutorial.” *Tutorialspoint*, Tutorials Point (I) Pvt. Ltd., 2016, www.tutorialspoint.com/erlang/index.htm

“Inside Erlang – creator Joe Armstron tells his story.” *Ericsson*, 4 Dec. 2014, www.ericsson.com/en/news/2014/12/inside-erlang--creator-joe-armstrong-tells-his-story

Lindahl, Tobias and Kostis Sagonas. “EEP 8: Types and function specifications.” *Erlang*, Ericsson AB, 2 Dec. 2007, erlang.org/eeps/eep-0008.html

O’Keefe, Richard A. “[erlang-questions] 回复： Is the function call always pass by value?” *Erlang*,Ericsson AB, 11 Mar. 2013, erlang.org/pipermail/erlang-questions/2013-March/072760.html

Scalas, Alceste. “EEP 7: Foreign Function Interface (FFI).” *Erlang*, Center forAdvanced Studies, Research and Development in Sardinia/Ericsson, 3 Sept. 2007, www.erlang.org/erlang-enhancement-proposals/eep-0007.html

“Scoping Rules in Elixir (and Erlang).” *The Unofficial Elixir Documentation*, 2014, elixir-lang.readthedocs.io/en/latest/technical/scoping.html

Thokala, Krishna Kumar. “A Comparison between Rust and Erlang.” *InfoQ*, Sergio De Simone, 13 Mar. 2018, www.infoq.com/articles/rust-erlang-comparison/.

Trottier-Hebert, Frederic. “Types (or lack thereof) – Dynamite-strong Typing.” *Learn You Some Erlang for Great Good!*, learnyousomeerlang.com/types-or-lack-thereof

Turner, D. A. *Some History of Functional Programming Languages*. University of Kent & Middlesex University, www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf