Jon Bowen

New Language Project Formal Report

CS 3210

Brief History

Erlang grew from the need by Ericcson, a telecommunications equipment manufacturer, for a language best suited to program switches used in telecommunications. Were conducted with 20 different existing languages from 1982 – 1985. Those experiments concluded that the language needed to be symbolic and high level to achieve productivity gains. Further experiments from 1985 – 1986 concluded that the language must contain primitives for concurrency and error recovery, the execution model must not be backtracking, and the granularity of concurrency necessary to represent one asynchronous telephony process by one process in the language. These requirements ruled out Lisp, Prolog, Parlog, and other candidates. The conclusion was that Ericcson must develop its own language ultimately culminating in Erlang. Following this line, the first version was released in 1988 for Ericcson internal use programming Private Automatic Batch Exchanges, a networking hardware component.

From 1989 to 1997 Erlang went from a small project of Joe Armstrong, Robert Virding, and Mike Williams to hundreds of people being involved. Following presentation at a conference in 1989, researchers at Bellcore expressed interest and the first external release was delivered to them by mid-1990. With interest growing, a C emulator, a considerably improved compiler, and the addition of libraries were produced by the end of 1990. Between 1990 and 1993 implementation additions included garbage collection, copying data between processes, records, included files, and macros. The language continued to grow to over 30 sites. In 1993 the first official book was published on Erlang following the decision to commercialize the language. Also in 1993 the addition of the BEAM compiler allowed Erlang to be compiled to C, adding cross-platform use and a large increase in speed at the cost of increased code volume.

The pivotal moment for Erlang came in 1995 when the collapse of a large Ellementel project called AXE-N, a large hardware platform with system software written in C++. When the project was reorganized, Erlang was chosen as the development language, and the project became the first large scales use of the language. This proved the value of Erlang and convinced Ericcson to invest significant resources to support the language.

In 1998, however, Ericcson banned the use of Erlang internally arguing that globally used languages offered more long-term value. Concurrent with this decision Jane Walerud convinced the management team that selling Erlang was not viable given the growing availability of open source alternatives. Management agreed, and the project was released as open-source in 1998. These two events convinced a large portion of the Erlang development team to resign and form their own company. These three events together greatly contributed to growth in Erlang use.

Since the late ‘90s, the growth in multiprocessor architectures has made Erlang an excellent choice for parallel processing. In 2006 the open-source language group released Erlang for symmetric multiprocessing. This was the most significant recent release.

Language Overview

Erlang falls under the functional paradigm of programming languages, disallowing side-effects, mutable data, and general changes in state of variables and data. Programming is done with expressions rather than statements, and the output of functions depends only on the arguments passed in. Related to this fact, variables, once assigned, can never be assigned a different value in the same scope. A local variable with the same name as a variable in another scope is effectively a different variable. Expressions in Erlang syntax are ended with a period character (‘.’).

Erlang uses implicit typing with type bound at assignment time. Most literals are of type Number with the runtime distinguishing between integers and floats as necessary. Characters are treated as integers with values selected using some encoding (e.g. ASCII or UTF-8). Strings are of type list with list members being integers corresponding to the characters composing a given string (again in some encoding).

Atoms are literal constants beginning with a lower case letter composed of alphanumeric characters. They are merely a name having itself as a value. Atoms can be used in boolean expressions, passed as arguments, and returned from functions. As an example “inch” (without quotes in the syntax) may be passed as a parameter in a function that converts units to indicate that another passed in measurement is in inches.

The most significant types to mention are tuples and lists. Tuples are comma separated sequences of any valid Erlang term between open and closed curly braces (e.g. “{ inch, “Steve”, 3.14, 7 + 3, MyVar }”). Tuples have a fixed number of elements in them. Lists are comma separated sequences of valid Erlang terms between open and closed square brackets (e.g.“[ one, two, 3, [ 6, 7, 8 ], “four” ]”). Unlike tuples, lists have varying sized.

There are several more complex and/or obscure types. In Erlang, but it is worth mentioning again that functions are first class in Erlang, meaning they can be assigned to a variable and passed as parameters.

Comments in Erlang begin with the % symbol and proceed to the end of a line. There is no syntax for multiline comments in Erlang.

Functions are defined piecewise as a collection of clauses separated by semicolons. The comma separated arguments of each clause form patterns that are match against until a match indicates the clause to be executed. For example if one clause has funcName( { City, { c, Temp } } ) -> *expressions;* then a call to that function will execute the expressions if the argument matches a tuple composed of a variable and a tuple with the 2nd tuple matching the atom c and a variable. This pattern matching is a powerful feature of the language.

Guards in Erlang are boolean expressions that can control execution path. They may occur as part of a function clause “guarding” which expressions will be evaluated if a pattern may match more than one clause, and can also occur in a variety of “if” constructs to determine which path to take. In the If and Case construct, at least one guard must evaluate to true or else a run time error will be thrown. Some but not all built-in-functions that may be used in guards, and no user defined functions may be used in guards. The latter condition is enforced to ensure that guards have no side-effects.

Finally for this high level summary of language and syntax, iteration is typically done through recursion as in most functional languages. Iteration is also possible over lists using foreach and map constructs.

Interesting Tidbits

* The = operator often used for assignment in programming languages is referred to as the match operator in Erlang. This connects nicely with Erlang’s concept of pattern matching and makes it clearer that data is immutable in the language. If we state that A matches 5 (A = 5) in some scope, then in the same scope it cannot also be that A matches 3.14 (A = 3.15 % bad match error).
* The | operator can be used with pattern matching to indicate “the rest of the list” anywhere in the list. For example [A, B | C] = [a, b, c, d] results in A = a, B = c, and D = [c, d]. Likewise a subsequent statement L = [e, f | C] results in L = [e, f, c, d].
* In an if construct, if none of the conditions evaluate to true, and error is thrown. An else is accomplished with the atom true as the guard (condition).
* Guards (expressions that evaluate to true or false) can only use certain built in functions and cannot use user-defined functions. The reason for this is that guards may not have side effects.
* Though not object oriented, message passing is a central feature of Erlang. Message passing is between processes rather than objects.
* An error in an Erlang process propagates to all linked processes. Unhandled error messages cause linked processes to be killed as well, further propagating the error. Processes that trap an error message remain running and do not further transmit the message.

Sample Code Fragments

See appendix

Standard References

* Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Programmers LLC, 2013.
* Joe Armstrong. “A History of Erlang.” 19 Mar. 2018, http://webcem01.cem.itesm.mx:8005/erlang/cd/downloads/hopl\_erlang.pdf.
* “An Erlang Course.” Erlang/OTP unit at Ericsson, 2 Apr. 2018, https://www.erlang.org/course.
* Erlang Homepage. Erlang/OTP unit at Ericsson, 2 Apr. 2018, https://www.erlang.org.
* “Documentation.” Erlang/OTP unit at Ericsson, 2 Apr. 2018, https://www.erlang.org/docs
* “Erlang (programming language).” Wikipedia, 11 Mar. 2018, https://en.wikipedia.org/wiki/Erlang\_(programming\_language).
* “History of Erlang.” Wikipedia, 12 Mar. 2018, https://www.erlang.org/course/history.

Evaluation Using Sebesta Criteria

Readability

Erlang is deceptively simple language with a small number of basic constructs. The core language has a very small number of familiar data types: numbers, atoms, tuples, maps, and lists, and functions. Booleans are represented by the atoms true and false, and strings and characters are stored as lists of numbers. There are several more exotic types, but those seldom require use. The number of keywords and code structures is pleasantly small. Most operations on built in datatypes are provided by built in modules that are themselves constructed from these datatypes.

This simplicity leads to a good balance of orthogonality. Language features generally are independent of the context of their appearance in a program, and most often have the same semantic meaning in all places they appear. The small number of data types and language constructs is directly behind the high orthogonality of the language. Further, data is immutable, leading to a very clear understanding of the meaning of a variable in any given scope.

The syntax of Erlang is initially foreign to a programmer most familiar with imperative languages such as C, but would be very familiar to functional language programmers. The style is comparable to beginner LISP, but without the accumulation of non-orthogonal and backward compatible features that LISP suffers from. Keywords are few in number and meaningful in name. Well-designed statements imply meaning with their appearance; however Erlang does not impose rigorous formatting. It is easy to write code that implies its meaning, but it is also easy to write code that obfuscates its meaning.

The use of -> is ubiquitous preceding clauses in function definitions and conditional statements. Somewhat opaque at first is the pattern matching implied by the syntax. This pattern matching is used in determining which version of a function to execute, variable binding, and parameter passing. However once a user is familiar with this syntax, it becomes a simple and powerful tool.

One potential drawback to readability is the need to write functions piecewise to allow for pattern matching to select the piece to execute. In addition, clauses in one branch often call another branch. This is related the intrinsic recursive approach in Erlang. In complex functions with many possible branches, may confuse even a careful reader.

Writability

Writability follows closely with readability. The small size of the language and related orthogonality means that the basic syntax, data types, and operations can be grasped quickly and powerful succinct code can be written fairly quickly. Complex problems can be solved knowing only the basic primitives and syntax.

There is good support for abstraction of both process and data. Business logic can be readily encapsulated in modules that define flexible methods that act on many kinds of data. This relates to the small number of high flexible data types. The highest level of abstraction in Erlang is at the process level. Erlang was specifically designed to encapsulate behavior in independent processes running on a machine with processes interacting through message passing. Data abstraction is well supported through combinations of the highly flexible primitive datatypes. Numbers, atoms can be assembled into tuples, lists, and maps that increasingly abstract data to a form that is clear for the problem domain.

Expressivity is gained through compact and convenient syntax that brings a relatively small number of powerful operators to bear on a problem. Examples include list comprehensions, function definitions that inherently accept a variety of types for their parameters, and pattern matching. Of course this compactness may hide a great deal of detail if the language is not well understood by the user. This balances against readability concerns, and the author must be relied upon to aid a target reader in using this expressiveness responsibility.

Reliability

Erlang was designed with reliability specifically in mind. The original developers needed a language to use for programming network switches which have extremely high uptime requirements, and error avoidance and recovery are primary to that goal. The small size of the language encourages readable code, avoiding programmer error. The immutability of data ensures that data will not be modified in unexpected ways. The functional paradigm reduces unpredictable or missed side effects from method calls.

Well-designed Erlang applications take advantage of Erlang’s robust error handling. Behaviors can be isolated into separate modules. Modules can be run as separate processes that can be linked and which communicate through message passing. When an error occurs in a process, the process is terminated, and the error propagates to all linked processes killing those processes as well until the error is trapped and handled. This feature encourages Erlang applications to be built in layers with appropriate error handling done by each layer. Additionally, Erlang systems allow code to be updated while a system continues to run. All of these features together allow extremely robust systems to be built.

One critique related to reliability is the lack of type checking. Erlang is dynamically typed, which can lead to runtime errors which might have been caught at compile time in a language more strongly typed. Additionally, Erlang has a relatively small number of flexible types. For example, strings are represented by number lists. This flexibility can lead to incorrect interpretations of data by a program. Such an error may not throw a runtime error, but will lead to an incorrect result.

The lack of type checking is somewhat ameliorated by the immutability of data. Since a variable can never be changed once it is bound, there is no possibility of mistakenly changing a variable’s type. However, nothing prevents a parameter of an incorrect type being passed to a function. In general it is incumbent upon the programmer to avoid type errors through best practices and good design. Fortunately the language encourages good design through its simplicity.

Cost

For programmers experienced with the functional programming paradigm, the cost of training programmers is small. The syntax would be familiar to any LISP programmer, and the small size of the language means that much can be accomplished after studying the language for a short time.

The cost of writing programs is also small compared to many languages in common use. History provides the best evidence of Erlang’s lost cost. When Ericcson prohibited the use of Erlang internally, most of the core development team left to form their own company. Within several months they completed their first product to take to market. The team credited the use of Erlang in enabling them to complete a large scale commercial product so quickly. The simplicity, writeability, and robustness of the language directly lowered the costs of development.

Erlang can have a higher cost of execution compared to alternative languages. Sequential program execution is generally slower than procedural languages because the language was not designed for this kind of use. The language design makes Erlang a suboptimal choice for applications that require a high amount of sequential data processing. However, for applications that are highly parallelizable, the immutability of data and the direct support for concurrency can overcome this drawback if many CPUs are available.

Finally, Erlang has strong advantages relating to generally the most expensive part of commercial software development: maintenance. The high reliability of Erlang systems directly reduces the amount of maintenance required. The ability to keep a system running while making updates also reduces maintenance costs. This feature limits downtime and allows greater flexibility in update timing and planning. The modularity that the language encourages can also cleanly organize the maintenance process. Modularity combined with Erlang’s readability work together to make applications easy to maintain by subsequent developer teams.

General Thoughts

Erlang is an extremely well designed language for systems that require fault tolerant, distributed, and real-time computing with very high availability. The ability to change code without stopping a system is also a big advantage for special applications. Examples of such systems clearly include the network switches for which the language was specifically designed for. Modern web applications also have requirements very well aligned with Erlang’s strengths, so much so that it is surprising that Erlang is not more widely used to build web servers in particular. The lack of Erlang use for such applications likely stems from less familiarity with the functional paradigm and its origin in equipment manufacturers rather than consumer application development.

In this era of multi-processor systems, the functional paradigm also makes Erlang an excellent candidate for systems that can be highly parallelized. With the growing need to write programs that can correctly take advantage of multiple CPUs or cores, I expect interest in Erlang to grow over time. Apart from niche applications in the 80’s and 90’s, Erlang seemed like a good idea in search of good problems to solve. Now there are expanding fields in popular computer science to which this language can be applied with great success.

Erlang is decidedly not a good choice for large scale sequential processing. If you need to loop over the same code a million times, C will be significantly faster, Java or C# will usually be faster, and Python often finish first. These languages also have the advantage of larger communities of developers, deeper resources of support, well tested libraries, and a programming model that is more intuitive to traditionally educated developers.

What Erlang does have is extremely good design. The language is small. The syntax is clear. The limitations encourage good choices whereas alternative languages invite the accumulation of bad choices. I look forward to finding my own applications for Erlang in the future.

Appendix

Code Samples

Declare a variable:

MyVar = 3.14.

MyString = “hello world.”.

MyList = [1, “two”, ‘3’, 4].

MyTuple = {piVal, 3.14}

Working with lists:

lists:nth(2, MyVar). % returns “two”

[H | T] = MyList. % H = 1, T = [“two”, ‘3’, 4]

lists:append(MyList, [a,b,c]). % returns [1, “two”, ‘3’, 4, a, b, c]

Test = [1,2,3,4].

lists:map( fun(E)-> E \* E end, Test ). % returns [1,4,9,16]

lists:seq(1,5). % returns [1,2,3,4,5]

Loops:

% Print each element of the passed in list

while([]) ->

ok;

while([H|T]) ->

io:fwrite("~w~n", [H]),

while(T).

while([1,2,3]).

% prints:

% 1

% 2

% 3

%---------------------------------------------------

% Print each element of the passed in list

Print = fun(E) -> io:fwrite(“~w~n”, [E]) end.

List = [1,2,3].

lists:foreach(Print, List).

% prints:

% 1

% 2

% 3

Conditionals:

% Function to compare two values

testTwo(A,B) ->

if

A < B ->

io:fwrite("A < B~n");

A == B ->

io:fwrite("A == B~n")

true ->

io:fwrite("A > B~n")

end.

testTwo(3,4). % prints “A < B”

testTwo(4,4). % prints “A == B”

testTwo(5,4). % prints “A > B”

Start a new process:

Pong\_PID = spawn(MyMod, MyFunc, Args) % creates process and returns ID

Message passing:

ProcessId ! “hello...can you hear me?” % send a message to process

%---------------------------------------------------

% block until message received, process, then send response

ping(Pong\_PID) ->

receive

pong ->

io:format("Ping received~n", [])

end,

Pong\_PID ! {ping, self()}.