TERM PROJECT

Programming Languages/Survey of Programming Languages.
ITCS 4102/5102

Survey of Erlang Programming Language

Part -1

Learning and Experimentation

Team Members

1. Gargi Sarkar - #800822013
2. Naga Bijesh Roy Raya - #800846698
3. Venkata Satya Sai Ram Adusumilli - #800851145

Submitted to:

Dr. Dewan Tanvir Ahmed

dahmed@uncc.edu

Department of Computer Science

University of North Carolina at Charlotte

Charlotte

Contents:

1. Paradigm of Erlang Programming Language:

Erlang is a functional programming language. With functional programming, the functions and operations of the language are designed in a similar way to mathematical calculations, in that the language operates on functions taking input and generating a result.

The functional programming paradigm means that the individual blocks of code can produce consistent output values for the same input values. This makes predicting the output of the function or program much easier and, therefore easier to debug and analyze. The functional programming approach is easy to understand, but can be difficult to apply if users are used to the more procedural and state-focused imperative languages.

2. History of Erlang:

Erlang was initially developed in order to improve the working of telephony applications. The initial version of Erlang was implemented in Prolog and was influenced by the programming language PLEX used in earlier Ericsson exchanges. Erlang was developed at Ericsson and was designed from the ground up for writing scalable, fault-tolerant, distributed, non-stop, soft-realtime applications. Everything in the language, runtime and libraries reflects that purpose, which makes Erlang the best platform for developing this kind of software. The first version was developed by Joe Armstrong in 1986. It was originally a proprietary language within Ericsson, but was released as open source in 1998. In 2006, native symmetric multiprocessing support was added to the runtime system and virtual machine.

Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

3. The elements of Erlang language:

Data types in Erlang Programming language are called as terms. The different Data types are:

- **Constant data types**

  – **Numbers**

  - Integers, for storing natural numbers
    Integers are written as sequences of decimal digits, for example, 12, 12375 and -23427 are integers. Integer arithmetic is exact and only limited by available memory on the machine. (This is called arbitrary-precision arithmetic.)

- Floats, for storing real numbers
  Floating point numbers use the IEEE 754 64-bit representation.

```
1> 42.
42
2> $A.
65
3> $\n.
10
4> 2#101.
5
5> 16#1f.
31
6> 2.3.
2.3
7> 2.3e3.
2.3e3
8> 2.3e-3.
0.0023
```

– **Atoms** :

An atom is a literal, a constant with name. An atom should be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @.

```
hello
phone_number
'Monday'
'phone number'
```

– **Pids** (short for 'process identifiers'), for storing process names

A process identifier, pid, identifies a process. spawn/1,2,3,4, spawn_link/1,2,3,4 and spawn_opt/4, which are used to create processes, return values of this type.

```
1> spawn(m, f, []).
<0.51.0>
```

– **References**

References are globally unique symbols whose only property is that they can be compared for equality. They are created by evaluating the Erlang primitive make_ref ().

-**Compound data types**

– Tuples

Tuples are containers for a fixed number of Erlang data types.

The syntax {D1,D2,...,Dn} denotes a tuple whose arguments are D1, D2, ... Dn. The arguments can be primitive data types or compound data types. Any element of a tuple can be accessed in constant time.

Tuples are used for storing a fixed number of items and are written as sequences of items enclosed in curly brackets. Tuples are similar to records or structures in conventional programming languages. for example: {a, 12, b}, {}, {1, 2, 3}, {a, b, c, d, e}.

– Lists

Lists are containers for a variable number of Erlang data types.

The syntax [Dh|Dt] denotes a list whose first element is Dh, and whose remaining elements are the list Dt.\ The syntax [] denotes an empty list. The syntax [D1,D2,..,Dn] is short for [D1|[D2|..| [Dn|[]]]]. The first element of a list can be accessed in constant time. The first element of a list is called the *head* of the list. The remainder of a list when its head has been removed is called the *tail* of the list.

Lists are used for storing a variable number of items and are written as

For example: [], [a, b, 12], [22], [a, 'hello friend']. Sequences of items enclosed in square brackets.

-**Binaries**

A built-in data type which can be used to store and manipulate an area of untyped memory. A binary is a sequence of bytes. Binaries provide a space-efficient way of storing binary data. Erlang primitives exist for composing and decomposing binaries and for efficient input/output of binaries.

-**Ports**

Ports are used to communicate with the external world. Ports are created with the built-in function (BIF)open_port. Messages can be sent to and received from ports, but these messages must obey the so-called "port protocol."

A port identifier identifies an Erlang port. open_port/2, which is used to create ports, will return a value of this type. They provide a byte-oriented interface to an external program. When a port has been created, Erlang can communicate with it by sending and receiving lists of bytes, including binaries.

The Erlang process which creates a port is said to be the **port owner**, or the **connected process** of the port. All communication to and from the port should go via the port owner. If the port owner terminates, so will the port (and the external program, if it is written correctly).

The external program resides in another OS process. By default, it should read from standard input (file descriptor 0) and write to standard output (file descriptor 1). The external program should terminate when the port is closed.

-**Funs**

Funs are function closures. Funs are created by expressions of the form: fun (...) -> ... end.

-**Strings**

Strings are written as doubly quoted lists of characters. This is syntactic sugar for a list of the integer ASCIIcodes for the characters in the string. Thus, for example, the string "cat" is shorthand for [99,97,116]. It has partial support for Unicode strings.[11]

-Records

Records provide a convenient way for associating a tag with each of the elements in a tuple. This allows one to refer to an element of a tuple by name and not by position. A pre-compiler takes the record definition and replaces it with the appropriate tuple reference.

Erlang has no method of defining classes, although there are external libraries available.

Reserved Keywords:

The following are reserved words in Erlang:

After, and, andalso, band, begin, bnot, bor, bsl, bsr, bxor, case, catch, cond, div, end, fun, if, let, not, of, or, orelse, receive, rem, try, when, xor.

4. Syntax of the Erlang language:

In most languages we would need to write something similar to this:

```
function greet(Gender,Name)
if Gender == male then
print("Hello, Mr. %s!", Name)
else if Gender == female then
print("Hello, Mrs. %s!", Name)
else
print("Hello, %s!", Name)
end
```

With pattern-matching, Erlang saves us a whole lot of unnecessary code. A similar function in Erlang would look like this:

```
greet(male, Name) ->
io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
io:format("Hello, ~s!", [Name]).
```

A simple implementation of Quick Sort program in Erlang:

```
qsort(X) -> qsort(X, []).

%% qsort(A,B)
%% Inputs:
%% A = unsorted List
%% B = sorted list where all elements in B
%% are greater than any element in
%% Returns
%% sort(A) appended to B

qsort([Pivot|Rest], Tail) -> {Smaller,Bigger} = split(Pivot, Rest),

qsort(Smaller, [Pivot|qsort(Bigger,Tail)]);

qsort([], Tail) -> Tail.
```

5. Basic control abstractions of Erlang

Erlang can use the primitives **case** and **if** for the conditional evaluation of the statements

Case: The case expression can allow choice between alternatives within its body clause.

Syntax:

```
case Expr of

    Pattern1 [when Guard1] -> Seq1;
    Pattern2 [when Guard2] -> Seq2;
    ...
    PatternN [when GuardN] -> SeqN
End
```

Control

The Expr is evaluated and the value of Expr is sequentially matched with the patterns [Pattern1, Pattern2, .. Pattern N] until an exact match is found. If a match is found then the guard test will be succeeded and then the corresponding Pattern sequence will be executed.

At least the Expr should match with one pattern else if none of the patterns match it would generate run time error and the error handling mechanism will be activated.

(i) **if**

The **if** statement has a guards and sequences for corresponding gards.

Syntax

```
if

  Guard1 ->

      Sequence 1;

  Guard 2 ->

      Sequence 2;

  …

end
```

The Guards [Guard1, Guard2, ….., Guard N] are evaluated sequentially. If the guard satisfies then the corresponding sequence will be evaluated. The result of this evaluation becomes the value of the if form.

Simplest:

```
factorial(0) -> 1;

factorial(N) -> N * factorial(N - 1).
```

Using function guards:

```
factorial(0) -> 1;

factorial(N) when N > 0 -> N * factorial(N - 1).
```

Using if:

```
factorial(N) ->

    if N == 0 -> 1;

    N > 0 -> N * factorial(N - 1)

end.
```

Using case:

```
factorial(N) ->

    case N of

    0 -> 1;

    N when N > 0 ->

    N * factorial(N - 1)

end.
```

6. How Erlang language handles abstraction

Programs of a certain size are complex. As long as the program is written by a single programmer and is fairly small, say under 1000 lines of code, then everything is easy. The programmer can keep the whole program in the head and it is easy to do stuff with that program. If on the other hand the program grows in size or we add more programmers, then we can't rely on the singular knowledge of a programmer.

The only way to solve this problem is to build in abstractions in your programs. We will review two such methods in Erlang. The idea of abstraction, informally, is that we will hide certain details and only provide a clean interface through which to manipulate stuff. Erlang is a "Mutually Consenting Adult Language" (read: dynamically typed with full term introspection - or more violently - unityped stuff with everything in one big union type). So this abstraction is not possible in reality. On the other hand, the dialyzer can provide us with much of the necessary tooling for abstraction.

Erlang has not one, but two kinds of ways to handle large applications: You can use modules, exports of types and opaqueness constraints to hide representations. While you *can* break the abstraction, the dialyzer will warn you when you are doing so. This is a compile-time and program-code abstraction facility. Orthogonally to this, a process is a runtime isolation abstraction. It enforces a given protocol at run time which you must abide. It can hide the internal representation of a process. It provides an abstraction facility as well. It is also the base of fault tolerance. If a process dies, only its internal state can be directly affected. Other processes not logically bound to it can still run. It is my hunch that these two tools together is invaluable when it comes to building large Erlang programs, several hundred thousand lines of code - and get away with it!

So in conclusion: To create modular code-level functional abstractions, rely on the dialyzer to create them for you like in the queue example from above. To create a modular runtime, split your program into processes, where each process handles a concurrent task.

7. An evaluation of the language's writability, readability, and reliability.

Erlang code tends to be concise & readable, which is made possible by the simplicity of the language & powerful abstraction mechanisms available.

When you write your app in Erlang, you get dynamic code upgrade support *for free* when you use OTP. The mechanism itself is very straightforward and easy to understand.
This can save hundreds of hours of time in development.

7. Strengths and Weaknesses of Erlang

Strengths:

Lightweight concurrency:

Erlang's lightweight concurrency model with massive process scalability independent of the underlying operating system is second to none. With its approach that avoids shared data, Erlang is the perfect fit for multicore processors, in effect solving many of the synchronization problems and bottlenecks that arise with many conventional programming languages. Its declarative nature makes Erlang programs short and compact, and its built-in features make it ideal for fault-tolerant, soft real-time systems.

Erlang also comes with very strong integration capabilities, so Erlang systems can be seamlessly incorporated into larger systems. Processes are very lightweight, with only about 500 bytes of overhead per-process. This means that millions of processes can be created, even on older computers. Because Erlang's processes are completely independent of OS processes (and aren't managed by the OS scheduler), your programs will behave in exactly the same way regardless of whether they run on Linux, FreeBSD, Windows or any of the other systems that Erlang runs on.

Free & open-source:

Erlang is distributed under a permissive open-source license, and is free to use for any open-source, freeware, or commercial projects.

Cross-platform:

Erlang runs on Linux, FreeBSD, Windows, Solaris, Mac OS X, and even embedded platforms such as VxWorks.

Plays well with the outside world:

Integration with existing Java, .NET, C, Python, or Ruby code is straightforward. There is an interface to the underlying OS should you need one. Solid libraries to work with XML, JSON, ASN.1, CORBA etc are also available.

HiPE:

The High Performance Erlang Compiler can compile Erlang to native code on Windows, Linux and Mac OS X and comes in the standard Erlang distribution.

Static typing, when required:

You can annotate your code with type information & use Dialyzer, a powerful type checker, to ensure the correctness of your code and gain performance. Dialyzer comes bundled with Erlang, and also supports gradual typing to give you maximum flexibility.

Bit syntax:

Another feature unique to Erlang is it makes working with binary data a breeze. Writing programs such as binary file readers or network protocol parsers is easier in Erlang than in any other language. Erlang code that uses the binary syntax is compiled into very efficient machine code, often beating hand-written C code in performance.

If the target system is a high-level, concurrent, robust, soft real-time system that will scale in line with demand, make full use of multicore processors, and integrate with components written in other languages, Erlang should be your choice.

Weaknesses:

- Erlang draws its strength from being a functional language with no shared memory. Hence IMO, Erlang won't be suitable for applications that require in place memory manipulations. Image editing for example.
- Processing big blocks of data is difficult.
- Single variable strength will fetch problems when handling complex data operations.
- Debugging becomes difficult when handling large blocks of Data.

8. Advantages of using Erlang

- Handles very large number of concurrent activities
- Easily distributable over a network of computers
- Fault-tolerant to both software & hardware errors
- Scalable with the number of machines on the network
- Upgradable & reconfigurable without having to stop & restart
- Responsive to users within certain strict timeframes
- Stay in continuous operation for many years
- Integration: Erlang can easily call or make use of programs written in other programming languages. These can be interfaced to the system in such a way that they appear to the programmer as if they were written in Erlang.

9. Sample Programs of Erlang Programming Language:

Program to compute the factorial of an integer.

```
-module(factorials).

-export([factorial/1]).

factorial(0) -> 1;

factorial(N) -> N * factorial(N-1).
```

Output:

> factorials:factorial(6).

720

Program implementing the built in functions of Erlang:

For example atom_to_list/1 converts an atom to a list of (ASCII) integers which represents the atom and date/0 returns the current date:

> atom_to_list(abc).

[97,98,99]

> date()

{93,1,10}

Program to create an echo process which echoes any message sent to it:

```
-module(echo).

-export([start/0, loop/0]).

start() ->
        spawn(echo, loop, []).

loop() ->
        receive {From, Message} ->

                From ! Message,

                loop()

        end.
```

echo:start() creates a simple echo process which returns any message sent to it.

spawn(echo, loop, []) causes the function represented by echo:loop() to be evaluated in parallel with the calling function.

Output:

```
Id = echo:start(),

Id ! {self(), hello}
```

The above input causes a parallel process to be started and the message {self(), hello} to be

sent to the process – self() is a BIF which returns the process identifier of the current process.

Program to show how the Expressions in Erlang language are evaluated:

The Erlang expression evaluation mechanism works as follows.

Terms evaluate to themselves:

> 584.

584

> abc.

abc

> 3.14159874.

3.14159

> {b,c,18,[b,c|d]}.

{b,c,18,[b,c|d]}

> {{},[{}],{a,85,'hello world'}}.

{{},[{}],{a,85,'hello world'}}

Floating point numbers might not be printed out in exactly the same format as they were input.

Program to show how Lists and its functions work :

> length([a,b,c]).

3

> lists:append([a,b], [1,2,3]).

[a,b,1,2,3]

> math:pi().

3.14159

Program to show the Order of Evaluation:

The BIFs apply(Mod, Func, ArgList) and apply({Mod, Func}, ArgList) are functions which apply the function Func in the module Mod to the argument list

ArgList.

> apply(dates, classify_day, [monday]).

weekDay

> apply(math, sqrt, [4]).

2.0

> apply({erlang, atom_to_list}, [abc]).

[97,98,99]

BIFs can be evaluated with apply by using the module name erlang.

Program to represent Module System in Erlang:

Erlang has a module system which allows us to divide a large program into a set of modules. Each module has its own name space;

```
-module(lists1).

-export([reverse/1]).

reverse(L) -> reverse(L, []).

reverse([H|T], L) -> reverse(T, [H|L]);

reverse([], L) -> L.
```

The above Program defines a function reverse/1 which reverses the order of the elements of a list. reverse/1 is the only function which can be called from outside the module. The only functions which can be called from outside a module must be contained in the export declarations for the module.

- In Erlang two functions with the same name but different numbers of arguments are totally different functions.

Erlang program showing the Precedence and associativity of Expressions:

Precedence:

The order of evaluation depends upon the priority of the operator: all priority 1 operators are evaluated, then priority 2, etc. Any bracketed expressions are evaluated first.

Associativity:

Operators with the same priority are evaluated left to right.

For example:

A-B-C-D

is evaluated as if it had been written:

(((A - B) - C) - D)

Sort program that shows the functionality of Lists:

The following program is a variant of the well-known quicksort algorithm. sort(X) returns

a sorted list of the elements of the list X.

```
-module(sort).

-export([sort/1]).

sort([]) -> [];

sort([Pivot|Rest]) ->

{Smaller, Bigger} = split(Pivot, Rest),

lists:append(sort(Smaller), [Pivot|sort(Bigger)]).

split(Pivot, L) ->

split(Pivot, L, [], []).

split(Pivot, [], Smaller, Bigger) ->
```

```
{Smaller,Bigger};

split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->

split(Pivot, T, [H|Smaller], Bigger);

split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->

split(Pivot, T, Smaller, [H|Bigger]).
```

Program Execution Process:

The first element of the list to be sorted is used as a pivot. The original list is partitioned into two lists Smaller and Bigger: all the elements in Smaller are less than Pivot and all the elements in Bigger are greater than or equal to Pivot.

The lists Smaller and Bigger are then sorted and the results combined. The function split(Pivot, L) returns the tuple {Smaller, Bigger}, where all the elements in Bigger are greater than or equal to Pivot and all the elements in Smaller are less than Pivot. split (Pivot, L) works by calling the auxiliary function split(Pivot, L, Smaller, Bigger). Two accumulators, Smaller and

Bigger, are used to store the elements in L which are smaller than and greater than or equal to Pivot, respectively. The code in split/4 is very similar to that in reverse/2 except that two accumulators are used instead of one. For example:

```
> lists:split(7,[2,1,4,23,6,8,43,9,3]).

{[3,6,4,1,2],[9,43,8,23]}
```

Program to show the Tail Recursion in Erlang:

```
-module(tailrecursion).

-export(length1/1).

-export(length1/2).

length1(L) -> length1(L, 0).

length1([_|T], N) -> length1(T, 1 + N);

length1([], N) -> N.
```

To evaluate length1([a, b, c]) we first evaluate length1([a, b, c], 0). This reduces to the evaluation of length1([b, c], 1 + 0). The + operation can now be performed immediately (because both its arguments are known). Successive function evaluations in the calculation of

length1([a, b, c]) are thus:

length1([a, b, c])

length1([a, b, c], 0)

length1([b, c], 1 + 0)

length1([b, c], 1)

length1([c], 1 + 1)

length1([c], 2)

length1([], 1 + 2)

length1([], 3)

3

10. Overview of the programs and Features of the language:

Erlang is a functional Programming Language that supports Sequential Processing as well as concurrent Execution of Programs. As this is the extensively used programming Language for Tele communication operations, errors in the programs can be rectified when the other programs are being executing without disturbing the execution flow of them.

Also, Erlang supports all the Basic Data types and even Advanced Data types like Lists, Tuples, binaries etc., like other functional programming languages that makes Erlang easy to learn. Moreover, Erlang programming allows programmer to utilize all the efficient features as discussed in the above modules in order to develop the sophisticated applications.

In addition to that, Erlang programming Language can be integrated to other Scripting, imperative Languages like Java, Ruby, Python etc., which makes Erlang to stand at the top of all the Functional Programming Languages.

The programs mentioned above define the ease of usage of Lists, Tuples, Sort using Lists etc., which in turn support various list operations using the predefined functions. Also the parallel processing can be obtained efficiently with the use of Pid, Binaries etc., Erlang Supports Inter

process Communication which is the key feature in Network communication mechanisms. One more appealing feature of Erlang Programming Language can be its Efficient and fast Execution Speed.

To conclude, Erlang is one of the Efficient Functional Programming Language that gives ease of usage to the Software Developers as it supports Socket programming which further can be integrated with JSon, HTML5 etc., and can be used for web development. Hence, the more users explore the features of Erlang Programming Language, the Smart applications they can built using this Language.

11. References:

1. http://veldstra.org/whyerlang/#more

2. https://www.safaribooksonline.com/library/view/Erlang-programming/9780596803940/ch01.html

3. https://www.safaribooksonline.com/library/view/building-web-applications/9781449320621/ch02.html

4. https://www.safaribooksonline.com/library/view/building-web-applications/9781449320621/