Seminar Report

Erlang

Thomas Trenkwalder
csag9584@uibk.ac.at

15 February 2011

Supervisor: Bertram Felgenhauer

Abstract

This work is concerned with the functional programming language Erlang. It was developed at Ericsson beginning in the 1980s, and evolved to a language that is now widely used in the industry. This paper mainly describes the most useful features of the language, but it does not intend to teach programming in Erlang. Basic knowledge about functional programming is assumed.

Contents

1 Introduction

This paper gives a short Introduction to the Erlang programming language. Erlang is one of the few functional programming languages that are used in the industry today. The language provides a number of built-in mechanisms to create concurrent, distributed and fault-tolerant applications.

First, the history of the Erlang programming language is summarized. In the next section some of the language's features are described. At the end, a short overview of existing projects using Erlang is given.

2 History of Erlang

This section summarizes the history of Erlang.

The Erlang homepage (see [1]) provides a short summary of Erlang's history[1] and a FAQ including answers to historical questions[2].

The interested reader can find a detailed history in [4] and [6] and a chronology in [5].

2.1 Experiments

The research leading to the development of Erlang began in the Ericsson Computer Science Laboratory during the 1980s. Previously, PLEX[3] was used to program Ericsson's AXE telephone exchange, which was first produced in 1974. PLEX offered many useful features, but it was specifically designed for the AXE hardware [6].

Experiments with more than 20 different programming languages were conducted to find out which languages were suitable for programming telecommunications applications. The results showed the need for a high level symbolic language. Declarative language programs turned out to be shorter and easier to understand than imperative language programs [4].

The people involved at the start were Joe Armstrong, Robert Virding and Mike Williams.

2.2 Birth of Erlang

During the experiments, no language with all the desired features for the task was found. Prolog was chosen as the best fitting language to continue experimenting with. Joe Armstrong then started an experiment to add concurrent processes to Prolog:

> "At this stage I did not intend to design a new programming language, I was interested in how to program POTS (Plain Old Telephony Service) — at the time the best method for programming POTS appeared to be a variant of Prolog augmented with parallel processes." – Joe Armstrong [5]

[1] `http://www.erlang.org/course/history.html`
[2] `http://www.erlang.org/faq/academic.html`
[3] Programming Language for EXchanges

The development of the emerging language was driven by programming a small telephone exchange in the laboratory and solving problems as they were encountered during this project.

The name “Erlang” was first mentioned in 1987. It was named after the Danish mathematician Agner Krarup Erlang[4], but can also stand for “Ericsson Language” [5].

2.3 Breaking out of the Laboratory

By the end of the 1980s, a group of Ericsson engineers started work on a prototyping project and chose Erlang as programming language. Collaboration with this team produced many ideas for the language. Performance requirements of the project led to development of the JAM (Joe’s Abstract Machine) which was inspired by the Warren Abstract Machine used for Prolog. Erlang was now regarded as a new programming language, rather than just a dialect of Prolog. The JAM was later replaced by a new abstract machine to further improve performance, the BEAM (Bogdan’s Erlang Abstract Machine).

In 1995, the Ericsson AXE-N project collapsed and a decision was made to start the project again, this time with Erlang. The project resulted in the development of the successful AXD301 switch (see 4.1), which was first delivered in 1998. The OTP project (Open Telecoms Platform) was started to consolidate a number of ideas and libraries created for Erlang.

In 1998, Ericsson banned Erlang for new product development. According to Joe Armstrong, Ericsson preferred to be a consumer of software technologies rather than a producer [5].

In 1998, Erlang and the OTP were released subject to an Open Source License, and the language continued to spread.

3 Language Features

The Erlang whitepaper[6] gives an overview of the features in Erlang. After showing the influence of Prolog on this language, this section will describe Erlang’s main features in detail.

Note that this section is not a guide to learn programming in Erlang. Basic knowledge about programming in a functional language is assumed.

3.1 Prolog Influence

As noted in section 2, Erlang started out as a variant of Prolog. The two languages thus have a similar syntax, as shown in the following code examples.

[4] 1878 – 1929, mathematician, statistician and engineer.
[5] `http://www.erlang.org/pipermail/erlang-questions/1999-February/000098.html`
[6] `http://erlang.org/white_paper.html`

```
% Prolog code
:- module(mymodule).
:- export list_length/2.

list_length([], 0).
list_length([_F|R], L) :-
  L is 1 + list_length(R).
```

Listing 1: Prolog code example.

```
% Erlang code
-module(mymodule).
-export([list_length/1]).

list_length([]) -> 0;
list_length([_F|R]) ->
    1 + list_length(R).
```

Listing 2: Erlang code example.

The only major difference in the above example is the way how functions return values. In Prolog, functions can return values via arguments, whereas in Erlang, the return value is the result of the last expression in a function.

Erlang has a dynamic type system which was inherited from Prolog. All types are checked at run-time.

3.2 Variables and Data Types

Variables begin with upper case letters (e.g. `Abc`, `MyVar`). An important restriction in Erlang is the single-assignment of variables. This means that a variable can be assigned (bound) only once, any assignment of an already bound variable will produce an error. Variables are assigned in successful pattern matching operations.

There is no explicit memory allocation or deallocation. Unused memory is managed by a garbage collector.

In Erlang, any piece of data is called a *term*. Some basic data types are:

Numbers: Integers and floats can be used. Character values are also represented as integers.

Atoms: Atoms are unique identifiers. They start with lower case letters or are enclosed within quotes (e.g. `an_atom`, `'Another atom'`).

Tuples: Used to store a fixed number of items of any type (e.g. `{123, an_atom}`, `{person, "Joe", "Armstrong"}`).

Lists: Used to store a variable number of items of any type. (e.g. `[2,3,5,7]`, `[123,some_atom]`).

There is no boolean data type, the atoms `true` and `false` are used instead. Strings are represented as lists of integers (e.g. `"Hello"` is the same as `[72,101,108,108,111]`).

Complex data structures can be created using the above basic types. Erlang also allows the definition of records, which are treated as tuples internally.

Other data types include:

Funs: Funs are functional objects (anonymous functions) which can be used like any other piece of data. Created using the `fun` keyword.

Bit Strings, Binaries: Used to store untyped memory. Erlang offers a *bit syntax* to express and manipulate bit strings (see 4.1).

Pids: Short for Process Identifier. Processes are created using `spawn` (see 3.3.1).

This paper will not go into detail about some other data types (e.g. References, Ports, . . .). A complete overview of all available data types can be found in the Erlang reference manual[7].

3.3 Concurrency

Concurrency has traditionally been made available through threads. Concurrently running tasks are allowed to operate on some shared memory, which introduces the need for locks. Deadlocks and starvation are among the problems caused by this approach.

Erlang however uses a different model based on the Actor Model, which tries to avoid these problems[9].

Actors can be seen as independently running processes. *Message passing* is the only way for actors to communicate with each other. Received messages are buffered in a mailbox. Actors do not have shared memory, therefore locks are unnecessary and problems like deadlocks are effectively avoided.

Erlang processes are not operating system processes or operating system threads. They are lightweight processes which are part of the language itself and are managed by the Erlang runtime system.

3.3.1 Creating an Erlang Process

The `spawn` built-in function can be used to create new Erlang processes. It expects a module name, a function name, and a list of arguments. It will return the pid of the new process and the calling function will continue execution without blocking.

This example shows simple concurrency in Erlang:

[7] `http://www.erlang.org/doc/reference_manual/data_types.html`

```
-module(mymodule).
-export([start/0, hello/2]).

hello(_, 0) -> ok;
hello(Name, N) ->
  io:format("Hello, ~s!~n", [Name]),
  hello(Name, N - 1).

start() ->
  spawn(mymodule, hello, ["world", 5]),
  spawn(mymodule, hello, ["user", 5]),
  ok.
```

Listing 3: Creating a process.

In this program, the `hello` function is used to print a hello message a given number of times. The `start` function is the main function of the program. It creates two new Erlang processes that use the `hello` function to print a different message five times.

There exists another version of the `spawn` function, which takes only a functional object as argument.

3.3.2 Communication between Processes

As noted above, Erlang processes do not have shared memory. They communicate via message passing.

Using the send operator (`!`), any Erlang term can be sent to another process. Writing `A ! B` sends the message `B` to the process `A`.

A `receive` block is used to retrieve a message from the mailbox of the current process, different patterns can be specified to selectively retrieve messages.

This example shows message passing:

```
-module(mymodule).
-export([start/0, pingpong/0]).

pingpong() ->
  receive
    {From, ping} ->
      io:format("Received ping~n"),
      From ! pong
  end,
  pingpong().

start() ->
  Process = spawn(mymodule, pingpong, []),
  Process ! {self(), ping},
  receive
    pong ->
      io:format("Received pong~n")
  end,
  ok.
```

Listing 4: Passing messages between processes.

The `pingpong` function simply waits for a tuple containing a pid and the `ping` atom, and sends the `pong` atom back to the received pid. The built-in function `self` is used to get the pid of the current process.

Messages are sent asynchronously, sending a message with the `!` operator will not block the current process. In contrast, `receive` will block, until there is a message in the mailbox that matches a pattern in the receive block. This behaviour can be changed with the `after` keyword, which can be used to specify a timeout value:

```erlang
receive
    ...
    after 5000 -> % Timeout in milliseconds
        true
end,
...
```

Listing 5: Retrieving messages from the mailbox.

3.4 Distribution

A running Erlang runtime system is called a *node*. A distributed Erlang system has a number of nodes communicating with each other.

Nodes have a name, which is an atom consisting of the node name and the host where it runs, separated by `@` (e.g. `mynode@homepc`). The Erlang Port Mapper Daemon (epmd) maps the node names to machine addresses. It is started automatically on every host where an Erlang node is started.

To create processes on remote nodes, another version of the `spawn` function can be used, which takes the name of a remote node as an extra argument at the beginning:

```erlang
Pid = spawn(mynode@homepc, mymodule, pingpong, []),
Pid ! {self(), ping},
...
```

Listing 6: Creating a process on a remote node.

The pid data type contains information about which node a process is running on. Communication between processes is therefore transparent, as shown in the above example.

3.5 Hot Code Upgrade

A requirement for telecommunications applications is being able to run virtually forever, without interruption. Erlang provides a simple mechanism to upgrade the code of a system while it is running.

The Erlang runtime system always keeps two versions of compiled code for each module. A simple function call (e.g. `myfunction()`) always refers to the *current* version of the code. Adding the module name to the function call (e.g. `mymodule:myfunction()`) always refers to the *newest* version of the code.

Consider the following piece of code:

```
-module(echo).
-export([loop/0]).

loop() ->
  receive
    {From, Message} ->
      From ! {self(), Message},
      loop();
    code_switch ->
      echo:loop()
end.
```

Listing 7: An echo process.

This piece of code shows a simple echo process. A running echo process can be upgraded by compiling the module `echo` and then sending the atom `code_switch` to the process.

There exists another solution to upgrade the code of running systems, because functions are first-class objects and can also be sent as messages. Consider the following piece of code:

```
-module(server).
-export([loop/1]).

loop(F) ->
  receive
    {newFunction, F2} ->
      loop(F2);
    X ->
      F(X),
      loop(F)
end.
```

Listing 8: A generic processing server.

This piece of code shows a simple generic server, which has a processing function. The processing function can be changed by sending a tuple containing the `newFunction` atom and a new function to the process. Any other message will be passed as argument to the current processing function.

3.6 Error Handling

Handling errors in Erlang is different than in most other languages. In single-threaded applications, not correcting an error might cause the entire application to fail. Erlang however follows a “let it fail” philosophy: Because Erlang programs generally consist of many processes, the failure of a single process is not so important. To build reliable systems, it is more important to detect the failure of processes and let other processes correct these problems.

Erlang processes can be linked together: `link(Pid)` creates a link between the current process and the process `Pid`. The links are bidirectional and are used to propagate errors. If a process dies, an error signal is sent to all processes linked to it. By default, a process that receives an error signal also dies and sends out error signals. This can be changed by calling `process_flag(trap_exit, true)`, which allows the current process to receive error signals as ordinary messages instead of exiting.

Using these mechanisms, a system can be organized in worker processes doing the actual computation, and supervisor processes which monitor the system and restore it to a safe state upon detecting errors.

3.7 Bit Syntax

Erlang provides a special syntax to manipulate *bit strings*[8]. A bit string is a sequence of bits, its length does not need to be divisible by 8. If it is divisible by 8, it is also called a *binary*.

This simple example shows construction of a binary and pattern matching with binaries:

```
A = 1, B = 2, C = 3,
Bin = <<A, B, C:16>>,      % construct a binary
                           % A and B are 8 bits long
                           % C is 16 bits long
...
<<D:16, E, F/binary>>,     % pattern matching
                           % D is 16 bits long, E is 8 bits long
                           % F is the rest of the binary (8 bits)
...
```

Listing 9: Constructing and matching binaries.

A more elaborate example shows how pattern matching can be used to extract the fields of a IPv4 datagram[9]:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

DgramSize = size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32, DestIP:32, RestDgram/binary>>
        when HLen>=5, 4*HLen=<DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
    ...
end.
```

Listing 10: Matching an IPv4 datagram.

4 Uses of Erlang

This section gives a short overview about a number of different projects that use Erlang.

[8] `http://www.erlang.org/documentation/doc-5.6/doc/programming_examples/bit_syntax.html`

[9] taken from `http://www.erlang.org/doc/programming_examples/bit_syntax.html`

4.1 AXD301 ATM Switch

Ericsson's AXD301 ATM Switch was created by a large programming team and has more than 1.6 million lines of Erlang code. It exceeded its original reliability requirements and has a downtime of only 31 ms per year, which makes it one of the most reliable switches ever made[3].

4.2 YAWS Webserver

The YAWS Webserver[10] is entirely written in Erlang and uses one Erlang process to handle each client.

An experiment measuring the performance of YAWS compared to the Apache web server[11] shows that while Apache dies at about 4000 parallel sessions, YAWS is still working at over 80000 parallel connections. Joe Armstrong speculates that the poor performance of Apache is a result of the way the underlying operating system implements concurrency, and the problem is not related to the code of Apache itself. For details about the experiment, see [2].

4.3 Other Uses

Erlang is used for many other projects in different areas, mostly in communications and reliable data storage[7].

Instant Messaging is an area where Erlang has found success, because these systems and telephone exchanges have similar requirements. The Facebook chat system uses the XMPP protocol, using a customized version of *ejabberd*[8].

Erlang is also well suited to implement schema-free databases. There exist a number of such databases, e.g. *Apache CouchDB*[12] and *Amazon SimpleDB*[13], among others.

Erlang has also been used in other areas it was not specifically designed for. Examples include *Wings3D*[14], a 3D graphics modelling program and *Nitrogen*[15], a web development framework.

5 Conclusion

This paper has summarized how Erlang developed and became a practical language that is successfully used in the industry. An overview about the most important features of the language was given. A number of projects using the language were mentioned, showing that Erlang is not only useful for the areas it was specifically designed for, but has also been successfully used as a general-purpose language.

[10] `http://yaws.hyber.org/`
[11] `http://httpd.apache.org/`
[12] `http://couchdb.apache.org/`
[13] `http://aws.amazon.com/de/simpledb/`
[14] `http://www.wings3d.com/`
[15] `http://nitrogenproject.com/`

References

[1] Erlang programming language, official website. http://www.erlang.org/.

[2] J. Armstrong. Apache vs. yaws. http://www.sics.se/~joe/apachevsyaws.html.

[3] J. Armstrong. Concurrency oriented programming in erlang. http://ll2.ai.mit.edu/talks/armstrong.pdf.

[4] J. Armstrong. The development of erlang. *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, 1997.

[5] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[6] J. Armstrong. A history of erlang. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[7] J. Armstrong. Erlang. *Communications of the ACM*, 53, September 2010.

[8] D. Reiss. Using facebook chat via jabber. http://developers.facebook.com/blog/post/110.

[9] R. Vermeersch. Concurrency in erlang & scala: The actor model. http://ruben.savanne.be/articles/concurrency-in-erlang-scala, January 2009.