Годишник на секция "Информатика"
Съюз на учените в България
Том 6, 2013, 103–110

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 6, 2013, 103–110

KEY FEATURES AND SOME APPLICATION AREAS OF ERLANG PROGRAMMING LANGUAGE

Maria Nisheva-Pavlova, Magdalina Todorova, Pavel Pavlov,
Atanas Semerdzhiev, Trifon Trifonov, Georgi Penchev, Petar Armyanov

Faculty of Mathematics and Informatics, Sofia University
5 James Bourchier blvd., Sofia 1164, Bulgaria

Abstract: The paper discusses the design goals and the main characteristics of Erlang – a modern functional programming language with support for concurrency, communication, distribution, and fault-tolerance. The main aspects of programming with Erlang – sequential functional programming, concurrent programming, programming multicore CPUs – are briefly described. Some typical application areas of Erlang have been analyzed.
Keywords: Programming language, Functional programming, Concurrent programming.

1. Introduction: Key Features of Erlang

Erlang is a general-purpose programming language that was created at the Ericsson Computer Science Laboratory by Joe Armstrong and his team. It was released under an open source license in 1998.

Erlang was originally designed for implementing telecom switching systems, i.e. for building large and complex systems which need to be reliable and to perform in a predictable way [1, 2]. It may be characterized as a concurrent functional programming language that has been used to create soft real-time systems with various amount of code. Erlang comes with the Open Telecom Platform (OTP) [3], which provides a rich set of libraries supporting the development of distributed, dynamically evolvable applications.
The design goals of Erlang are reflection of its initial application area [1]:

- *concurrency* – thousands events, such as phone calls, happen and should be processed simultaneously;
- *robustness* – Uptime in telecommunications is a lot more important than domains such as Web/Online services. Thus, an error in one part of the application must be caught and handled so that it does not interrupt other parts of the applications. Preferably, there should be no errors at all.
- *distribution* – the system should be distributed over several computers, either due to the intended features of the application, or for robustness or efficiency.

The following key features of the language follow in the train of the above considerations:

- functional programming style with strict evaluation, single assignment, and dynamic typing;
- message passing paradigm for concurrency;
- error recovery;
- hot code replacement.

2. Sequential Programming in Functional Style

Erlang is a pure functional programming language. Each variable can be given a value only once. Thus Erlang code causes no side effects and this is an important prerequisite for clearer programming, easy debugging of programs and concurrency implementation.
The basic units of code in Erlang are called *modules*. All functions are stored in modules. Modules have to be compiled before their code can be run.

2.1. Erlang Data Types

Erlang is a dynamically typed programming language which supports the following data types:

- Arbitrary-sized integers, e.g. `2345678987654321` and `16#AB10F`;
- Floating point numbers: `123.456789345`;
- Atoms (non-numerical constant values), e.g. `red`, `blue`, `'an atom'`, `'55'`;
- Tuples (group a fixed number of values in a single entity): `{name,michaela}`, `{age,23}`, `{person, {{name,michaela},{age,23}}}`;
- Lists (group variable number of values in a single entity), e.g.:
  - `[]` – the empty list,
  - `[1,2,3,4,5]` – list of five integers,
  - `[{apples, 5},{pears, 4},{plums, 10}]` – list of three tuples,
  - `[1,foo,asd,{aa,bb,3}]` – list with elements of different types,
  - `[H|T]` – list with head (first element) `H` and tail (list of remaining elements) `T`,
  - `[E1,E2,E3|T]` – list with head `E1,E2,E3` and tail `T`;
- Funs (functions): `fun(X) -> X*2 end`;
- Pids (unique process identifiers);
- Refs (guaranteed unique identifiers).

2.2. Pattern Matching

The pattern matching operation `LHS = RHS` in Erlang means the following: evaluate the right-hand side (`RHS`) and then match the result against the pattern on the left-hand side (`LHS`). Any unbounded variables are bound in this process.
Examples of pattern matching (identifiers beginning with capital letters denote variables):

Pattern	Term	Result	
{123,abcd}	{123,abcd}	*Succeeds*	
{X,abcd}	{123,abcd}	*Succeeds*	and X → 123
{X,Y,Z}	{123,abc,'aa'}	*Succeeds*	and X → 123, Y → abc, Z → 'aa'
{X,Y}	{123,abc,'aa')	*Fails*	
[H|T]	[1,2,3,4,5]	*Succeeds*	and H → 1, T → [2,3,4,5]
[H1,H2|T]	[1,2,3,4,5]	*Succeeds*	and H1 → 1, H2 → 2, T → [3,4,5]
[H1,H2,H3|T]	[1,2]	*Fails*	

2.3. List Comprehensions

List comprehensions in Erlang are similar to these in Haskell. A list comprehension in Erlang has the form `[Expr || Qualifier_1, Qualifier_2, … ]`, where `Expr` is an expression and `Qualifier_i` is either a generator or a filter. Generator is of the form `Pattern <- ListExpr`, and filter is a predicate that evaluates to true or false.
For example, `[2*X || X <- L]` doubles all elements of the list of numbers `L`.

2.4. Functions

Generally, function definitions are of the form

```
Head_1 -> Body_1; Head_2 -> Body_2; … ; Head_n -> Body_n,
```

where `Head_i` is the name of the function (usually an atom) followed by a set of arguments (patterns) and followed optionally by a guard (see section 2.5). `Body_i` is a comma-separated list of expressions. Named functions are defined inside modules and are called combining the module name with the function name. Erlang maintains tools for definition and use of higher-order functions – functions can be used as arguments to other functions or returned by other functions.
Examples of function definitions:

```
%area function – computes the area of a geometrical figure%
area({rectangle,Width,Height}) -> Width*Height;
area({circle,Radius}) -> Pi*Radius*Radius.
%definition of functions with the same name and different arity%
sum(L) -> sum(L,0).
sum([],N) -> N;
sum([H|T],N) -> sum(T,H+N).
%sequential map function - example of function as argument%
map(_, []) -> [];
map(F,[H|T]) -> [F(H)|map(F,T)].
%anonymous functions%
Double = fun(X) -> (2*X) end.
Even = fun(X) -> (X rem 2) =:= 0 end.
```

Anonymous functions (functions that have no name) in Erlang are called *funs*. Funs may be used as arguments to functions and functions may also return funs.

2.5. Guards

Guards in Erlang have the form `when P`, where `P` is a predicate that can be evaluated to true or false. They are used at the head of a function or at a pattern matching evaluation.
Examples:

```
f(X,Y} when is_integer(X), is_integer(Y), X > Y -> 1;
f(X,Y) -> 0.
max(X,Y) when X>Y -> X;
max(X,Y) -> Y.
```

3. Concurrent programming

Erlang is a functional programming language, especially created for building parallel and distributed computing systems. The concurrency model and its error-handling mechanisms were built into Erlang as one of its design objectives [4]. Erlang is oftentimes called a pure message passing language which means that [3, 6]:

- Processes belong to the programming language and not to the operating system;
- Processes behave in the same way on all operating systems;
- There can be a very large number of processes;
- Processes share no memory and are completely independent;
- The only form of communication between processes is via message passing.

Erlang implements a message exchange model which satisfies the following requirements [7], giving its authors the reason to characterize it as a Concurrency Oriented Programming Language (COPL):

1. COPLs must support processes. A process can be thought of as a self-contained virtual machine.
2. Several processes operating on the same machine must be strongly isolated. A fault in one process should not adversely affect another process, unless such interaction is explicitly programmed.
3. Each process must be identified by a unique identifier. We call this the Pid (shortly for *Process identifier*) of the process.
4. There should be no shared state between processes. Processes interact by sending messages. If you know the Pid of a process then you can send a message to the process.
5. Message passing is assumed to be unreliable with no guarantee of delivery.
6. It should be possible for one process to detect failure in another process. We should also know the reason for failure.

The mathematical model based on this message exchange model is called the *actor model* [8] and has been introduced in [9]. Its main component is the so-called *strong isolation* which requires single-assignment semantics and application of a functional model with non-mutual data. These requirements are satisfied by Erlang. Objects in the actor model are called *actors*. Actors are universal units in parallel computation. Each of them has a

mailbox and a specified behaviour. An actor can send and receive messages, make decisions, and create other actors.

The communication between actors is asynchronous and has been realized by message passing. All actors of a program work in parallel and may be considered as independent processes.

In order to control a set of parallel activities Erlang has three basic primitives for multiprocessing [5]: `spawn` starts a parallel computation (called a process), `send` sends a message to a process, and `receive` receives a message from a process.

`spawn/3` starts the execution of a parallel process and returns a Pid which may be used to send messages to and receive messages from the process:

`Pid = spawn(Module, Function, ArgumentList)` creates a new concurrent process that evaluates `Function`. The call to `spawn` returns immediately when the new process has been created and does not wait for the evaluation of the given function. The new process runs in parallel with the calling one.

Pids are used for all forms of communication with a process. A message is sent to another process by the primitive `'!'` (`send`):

`Pid ! Message`

Here `Pid` is the identifier of the process to which `Message` is sent. A message can be any valid Erlang term. `send` is a primitive which evaluates its arguments. Its return value is the message sent.

The primitive `receive` is used to receive a message that has been sent to a process. It has the following syntax [5]:

```
receive
  Pattern1 [when Guard1] ->
    Expressions1;
  Pattern2 [when Guard2] ->
    Expressions2;
  ...
end
```

When a message arrives at the process, the system tries to match it with `Pattern1` (possibly using an additional guard, `Guard1`). If this succeeds, it evaluates `Expressions1`. If the first pattern does not match, the system tries `Pattern2`, etc. If no pattern matching succeeds, the message is saved for later processing and the process stays waiting for the next message.

The patterns and guards here have the same syntactic form and semantics as the patterns and guards in a function definition.

Generally, sending a message is an asynchronous operation – the send call does not wait for the message either to arrive at the destination or to be received. Messages are always delivered to the recipient in the same order they were sent.

Each process has a mailbox and all messages that are sent to a process are stored in its mailbox in the same order as they arrive. In the syntactic form above, `PatternN` are patterns that are matched with the messages in the process's mailbox. When a matching message is found and the corresponding guard succeeds, the message is selected, removed from the mailbox and then the corresponding `ExpressionsN` are evaluated. `receive` returns the value

of the last evaluated expression. No message arriving unexpectedly at a process can block any other message to this process.

It is possible to send/receive messages to/from a specific process. The sender can do it using the following form of `send`:

`Pid ! {self(),Message}`

which sends a message containing the identifier of the sender's process, returned by the built-in function `self()`.

Such messages may be received by

```
    receive
  {Pid,Pattern} ->
  ...
end
```

If `Pid` is bound to the sender's process identifier, then the evaluation of calls to `receive` in the above form would allow to receive messages only from this process.

4. Programming Multicore CPUs

Contemporary functional programming has gone out of the academic community and finds a wide range of applications in the implementation of complex real-time software systems. Modern functional programming languages support tools which ensure significant performance gains when running on multicore systems. Erlang is an industry-oriented functional language, appropriate for programming multicore processors. The latter is one of the main reasons for the growing interest in it. The concurrency model and the support of symmetric multiprocessing in Erlang are integral parts of the language standard.

While a great amount of efforts have been directed to the search/development of methods facilitating the transition of software applications to multicore CPUs, Erlang applications can be transited practically unchanged. Moreover, since Erlang is a concurrency oriented programming language, Erlang programs are expected to be vastly better when run on multicore systems – possibly n times faster on an n-core processor. The comparative analysis presented in [7] shows that this is not always true and that performance gains are limited by how parallel the program is and by the current Erlang running environment implementation even on adequately parallel programs. Joe Armstrong gives in [3] the following good advices how to make Erlang programs run efficiently on multicore CPUs:

- use lots of processes;
- avoid side effects;
- avoid sequential bottlenecks;
- write "small messages, big computations" code.

Reference [3] contains some comments regarding these advices as well as a discussion of proper techniques for parallelizing sequential code and an example demonstrating that using the basic primitives for multiprocessing: `spawn`, `send`, and `receive`, one can build a considerable family of abstractions.

5. Applications of Erlang

Erlang was designed as a parallel functional programming language, appropriate for programming real-time control systems. Its authors wanted "to make a language which addressed many of the problems which are handled in an operating system while maintaining the advantages of a declarative programming language" [6]. Nowadays many companies are using Erlang in their production systems, for example [4]:

- Amazon uses Erlang to implement its SimpleDB which provides database services as a part of the Amazon Elastic Compute Cloud;
- Yahoo! utilizes Erlang in its social bookmarking service;
- Facebook uses Erlang to power the backend of its chat service;
- T-Mobile uses Erlang in its SMS and authentication systems;
- Ericsson applies Erlang in its support nodes, used in GPRS and 3G mobile networks worldwide.

There are a number of popular open-source Erlang applications, for example: the CouchDB document-oriented database that provides scalability across multicore and multiserver clusters; the Ejabberd system, providing an eXtensible Messaging and Presence Protocol (XMPP) based instant messaging application server; the MochiWeb library that provides support for building lightweight HTTP servers, etc.

As far as Erlang is oriented around concurrency, it is naturally good at utilizing modern multicore systems.

It is shown in [3] that Erlang can be used as a rapid and effective solution for building SHOUTcast servers to stream MP3- and AAC-encoded audio data using the HTTP protocol. It is also a proper platform for building full-text search engines that can index and search large volumes of data. In this sense Erlang could be considered for the implementation of digital library systems intended to keep and give up adequate access to rich digital content.

As skilled users summarize, it is advisable to use Erlang if the application is wanted to: handle large number of concurrent activities; be easily distributable over a network of computers; scale with the number of machines on the network; be fault-tolerant to both software & hardware errors; stay in continuous operation for many years.

6. Conclusion

The transition to multicore CPUs is presenting the IT industry with a number of new challenges that require a radical change of the programming paradigms instead of searching new methods for optimization of software applications. An attempt to resolve these challenges is the Erlang language, which integrates three programming paradigms: the functional, parallel, and strong ones. In the introduction to [4], Mike Williams – Director of Traffic and Feature Software Product Development Unit WCDMA, Ericsson – summarizes that "Erlang is our solution to three problems regarding the development of highly concurrent, distributed soft real-time systems":

- to be able to develop the software quickly and efficiently;
- to have systems that are tolerant of software errors and hardware failures;
- to be able to update the software on the fly, that is, without stopping execution.

This paper describes some of the main features of Erlang which substantiate the achievement of its design goals.

Acknowledgments

The presented study has been supported by the Bulgarian National Science Fund within the project titled "Contemporary programming languages, environments and technologies and their application in building up software developers", Grant No. DFNI-I01/12.

References

[1] Agarwal, S., P. Lakhina, *Erlang – Programming the Parallel World*. `http://cs.ucsb.edu/~puneet/reports/erlang.pdf` (date of last visit: April 27, 2013).

[2] Athinaiou, E., G. Yeniceri, *Erlang and Concurrency*. `http://cs.nyu.edu/~lerner/spring10/projects/Erlang.pdf` (date of last visit: April 27, 2013).

[3] Armstrong, J., Programming *Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2009.

[4] Cesarini, F., S. Thompson, *Erlang Programming*. O'Reilly Media, 2009, ISBN 978-0-596-51818-9.

[5] Primi, M., *The Erlang Programming Language* [Programming Languages 2007 Report]. `http://www.mpri.me/projects.php#erlang` (date of last visit: April 27, 2013).

[6] Armstrong, J., Erlang – A Survey of the Language and its Industrial Applications. *Proceedings of the 9th Exhibitions and Symposium on Industrial Applications of Prolog* INAP96 (16–18 October 1996, Hino, Tokyo, Japan).

[7] Armstrong, J., *Making Reliable Distributed Systems in the Presence of Software Errors*, Ph.D. Dissertation, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[8] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*, Doctoral Dissertation, MIT Press, 1986.

[9] Hewitt, C., P. Bishop, R. Steiger, *A Universal Modular Actor Formalism for Artificial Intelligence*, 1973, IJCAI, Stanford.