Book review

Concurrent Programming in Erlang by Joe Armstrong, Robert Virding and Mike Williams, Prentice Hall, 1993, 281 pp.

The following impressive text is written on the back cover of the book *Concurrent Programming in Erlang* by Joe Armstrong, Robert Virding and Mike Williams:

Erlang is a new functional programming language designed for building large-scale industrial applications. The language is especially suited for programming large-scale fault-tolerant real-time applications, and has many features which are essential for non-stop industrial applications. It provides the following:

- Mechanisms for changing software in running systems without stopping the system.
- Lightweight processes for expressing concurrency.
- Advanced error detection mechanisms for building robust and fault-tolerant systems.
- Real-time garbage collection which also eliminates memory fragmentation.

Anyone who is working in a telecom company knows how important the requirements are for which the designers of Erlang claim to provide a solution: on-line replacement, concurrency at lightweight-process granularity, advanced error-handling, real-time garbage collection – it's really everything a telecom software supplier can ever dream of!

I work in a telecom company myself and I also have some knowledge about functional programming. Therefore, when I first saw the book, I was really curious from the practical point of view, to see if all these promises were real and, from the theoretical point of view to see how Erlang deals with the important topics stated above. I have not tried out Erlang on very large programs (yet), but my first impression of Erlang is very positive. The language designers and implementors have, in my opinion, realized an important piece of work. Erlang offers its user simple and elegant concepts which enable her or him to write excellent telecom applications.

It is also clear to me that the Erlang designers have chosen for a pragmatic approach. The language offers a set of extra concepts which make it more useful to program telecom applications. Many of these concepts, especially the ones for dealing with parallelism, have no evident corresponding concepts (yet) in pure (referentially transparent) functional programming languages. Some results have already been obtained in this direction by Hudak and Jones (1993): they propose a monadic parallel programming style, in the same spirit as the monadic input/output programming style proposed by Peyton Jones and Wadler (1993). Anyhow, Erlang can serve as an inspiration for researchers of the functional programming community who want to provide their favourite pure functional programming language with concepts for making the language more useful and therefore, by definition, more popular.

One minor point: Erlang is *not* a strongly typed language. This has the well known disavantages. I have personally had some problems with a program dealing with lists of lists. Inserting a list element instead of a one element list gave me much runtime trouble which could have been avoided at compile time.

About the book

The book is written in a very clear style. It can be used as such for giving a course on some of the more important aspects of concurrent (functional) programming, although it does not contain exercices. Many courses on (functional) programming do not deal with the fascinating

aspects of concurrency at all. This book provides readers with a lot of interesting examples which will hopefully convince them of the usefulness of having multiple threads of control at their disposal.

On the one hand, the book is intended for functional programmers which are not familiar with techniques for concurrency and robustness. On the other hand, the book is intended for imperative programmers which are familiar with the problems related to large-scale programs, but which are not familiar with functional programming. For both audiences the book offers valuable insights in the art of programming.

The book introduces a lot of programming concepts which are perhaps not very common to the average functional programmer, who may wonder if all these concepts have anything to do with functional programming at all. However, all concepts are build around a nice little pure functional programming kernel and, as mentioned above, one should consider it as a challenge to find out to what extent these concepts could be put into a pure functional programming framework.

Chapter 1

The first chapter introduces the reader to Erlang via a short tutorial. Erlang is a fairly straightforward functional language, with features we have all come to know and love, including pattern matching. The language is strict (call-by-value) rather than lazy (call-by-need).

The basic data types are numbers, atoms, lists, and tuples. Atoms begin with a small letter, variables with a capital. As the language is untyped, lists may be heterogenous, e.g. `[1,a,X]` where `1` is a number, `a` is an atom, and `X` is a variable which may be bound to a value of any type. Tuples are like lists, but written with curly brackets, e.g. `{1,a,X}`. The values `[1,a,X]` and `{1,a,X}` differ only in their representation: the list will occupy three cons cells, while the tuple will occupy a single cell with three fields. There are no sum types, but these are conventionally formed from atoms and tuples, as in `{left,X}` and `{right,Y}`.

Erlang is halfway between a first-order and a higher-order language. Functions may be passed as arguments, returned as results, and stored in data structures, but to apply a function bound to a variable one must first use the built-in `apply` function. (This point isn't made clear until Chapter 3, however.)

Erlang is a concurrent programming language. It has primitives to *spawn* a parallel process, to *send* a message to a process, and to *receive* a message from a process. A nice property is that message layout can be descibed in exactly the same way as function parameters: by making use of patterns. This offers a flexible way to unpack messages.

Some minor remarks on this tutorial: first of all I had a little problem to find out how to load the code of a module (this could have been documented in a better way) and second I had a little problem to let the first 'concurrent program' perform something useful for me (perhaps a slightly more elaborated example would be more appropriate here).

Chapter 2

This chapter introduces the reader to the fundamentals of sequential programming in Erlang. The chapter contains no surprises for the average functional programmer (although its programs may look, like all other functional programs, surprisingly simple to the average imperative functional programmer).

Chapter 3

This chapter introduces the reader to the fundamentals of programming with *lists*. Again the chapter contains no surprises for the average functional programmer. The chapter contains a section on common patterns of recursion on lists, and introduces higher-order functions such as `map` and `filter`, which require the above mentioned `apply` function for their definition.

https://doi.org/10.1017/S0956796800001519 Published online by Cambridge University Press

One can make use of a whole range of built-in functions dealing with lists (and with tuples, the subject of the next chapter). An example is the function `list_to_atom` which converts a list of ASCII characters to an atom. Another example is the function `setelement` which updates the value of the `N`-th field of a tuple `T`. An important family of built-in functions deals with *run-time type information.* An example of this is the guard `list` which checks whether its argument is a list. A complete list of built-in functions is presented at the end of the book.

Chapter 4

This chapter introduces the reader to the fundamentals of programming with *tuples.* Again, the chapter contains no surprises for the average functional programmer. It shows how to use tuples to deal with classical data structures as dictionaries and (balanced) binary trees.

Perhaps one striking feature of this chapter is the ease with which one can use tuples and atoms to deal with exceptional situations. For example: the result of a `lookup(Key,Dict)` which looks for a key in a dictionary is either a tuple `{value, Value}` or an atom `undefined`. This is similar to the approach using the `Maybe` monad which is often used in Haskell.

Chapter 5

This chapter starts exploring the more interesting aspects of Erlang: the possibility to use Erlang as a concurrent programming language. All concurrency aspects of Erlang are explicit. The basic concurrency primitives deal (not very surprisingly) with process creation and process communication.

Process creation is dealt with using the built-in function `spawn(Mod,Func,Args)` which has the same signature as the function `apply(Mod,Func,Args)`. Both apply function `Func` defined in module `Mod` to arguments `Args`. The difference is that `spawn` will create a new (concurrent) process to evaluate the function call and returns immediately with the process identifier `Pid` of the created process. This identifier `Pid` is only known to the creating process, and may be retrieved by the created process using the built-in function `self()`. No other process can retrieve this identifier unless it is explicitly exported, which affords some measure of security. The result of evaluating the function call will be lost. Thus there must exist some communication facility between processes to let created processes perform something useful for their creator.

Process communication is only possible by means of message passing by using the language construct `!` (send) on one side and the language construct `receive` on the other side. The arguments of send can be arbitrary terms. A send is an *asynchronous* operation. The process which evaluates a send will not wait for anything. Every process has a mailbox which contains all messages which have been send to it in the order in which they have arrived. If a process evaluates a receive then it linearly scans the messages in the mailbox to see if they match one of the patterns of the receive primitive (in the order they appear in the receive primitive). If such a message is found, then it is removed from the mailbox and the expression corresponding to the matching pattern is evaluated. If no such message is found, then the process is suspended until such a matching message is found. The same binding convention as the one for function arguments applies for the receive primitive.

The order of the patterns in a receive primitive cannot directly be used as a method to implement priority (this can, however, be done using zero-timers, see later).

If a process wants to receive only messages from a specific process, then this has to be arranged in co-operation with the sender who has to include its own process identifier (say `Pid`) in the message (e.g. as a first field of the message). The receiver can then decide to wait only for messages of this specific sender by using a pattern `{Pid, Msg}`. (Note the slightly unusual pattern matching: variable `Pid`, which is bound in the outer scope, matches only against the value bound to it; while variable `Msg`, which is not bound in the outer scope, matches against any value and binds that value to the variable.)

https://doi.org/10.1017/S0956796800001519 Published online by Cambridge University Press

The private context of a process is simply the arguments `Args` passed to it in the call `spawn(Mod,Func,Args)`. The only way to let a process change its context is through recursive calls of `Func` with different arguments. In my opinion this is one of the most important features of the language! (In particular, as described later, it makes it easy to do on-line code replacement.) Of course recursion may result in run-time overhead, but the authors emphasize how to avoid this by using tail recursion.

The authors of the book also pay special attention to the discipline of defining *interface functions* through which a process can be invoked. The body of these functions will then typically contain the concurrency primitives. In such a way the usage of the concurrency primitives is encapsulated. Note that a send or a recieve expression can appear everywhere in a function body. This is very flexible but it has the consequence that a process may be descheduled in the middle of evaluating a function call. In my opinion code is structured in a better way if the usage of the concurrency primitives is somehow limited.

The authors also pay special attention to the fact that the receiver is responsible for its mailbox, e.g., to arrange that the mailbox does not get filled up with unprocessed messages. This can be handled by using a default last receive pattern which matches everything.

It is possible to indicate a time interval after which a receiving process will execute a timeout action if no matching message has been received during the interval. Timeouts (especially the ones with a zero-time interval!) have much power of expression. They can e.g. be used to suspend a process, to flush its mailbox, to implement priorities and to implement timer processes. All time intervals are given in milliseconds, which is a nice indication of the application domains to which Erlang is tuned.

It is possible to register processes (i.e. to give them a system-wide unique name) using `register(Name, Pid)`. A send primitive can make use of registered names. In such a way a client can use the services of any registered server. The authors pay special attention to the way client-server applications can be written using three basic ingredients: the server code, a protocol, and an access library. It is the functions of the access library which should be made public for use by the clients.

High level facilities can be programmed in an easy and flexible way using the Erlang concurrency primitives which we have discussed above. The Erlang programmer does not have to worry about such issues as sheduling and memory management. It is up to the Erlang implementors to guarantee that some criteria are satisfied. Typically scheduling will have to be fair and may not block the system too long. Allocating and reclaiming memory must also be done in such a manner that it does not block the system too long. For all those criteria the Erlang implementors have done a great job.

Chapter 6

Erlang has mechanisms for letting a process monitor its evaluation of an expression, monitor the behaviour of other processes, and trap evaluation of undefined functions.

The primitives used for monitoring the evaluation of expressions are `catch` and `throw`. They can be used for protecting sequential code from errors and arrange for a non-local return from a function. To monitor evaluation of an expression `Expr` one writes `catch Expr`. If nothing goes wrong, this returns the value of the expression; thus `catch 2*11` returns 22. If a failure occurs, this returns `{'EXIT',Reason}`, where `Reason` is an atom indicating the reason for failure; thus `catch 2*atom` returns `{'EXIT',badarith}`. If a call to the built-in function `throw` is evaluated, then the value thrown is returned; thus `catch 2*throw(42)` returns 42. Note that calling `throw({'EXIT',Reason})` simulates the effect of a failure.

One process can monitor another's behaviour via a link. Calling `link(Pid)`, causes the executing process to be linked to the process with identifier `Pid` (and vice versa, since links are bidirectional). There is also a built-in function which both spawns a process and links to it as a single atomic action. If a process terminates, a signal of the form `{'EXIT',Pid,Reason}`

https://doi.org/10.1017/S0956796800001519 Published online by Cambridge University Press

is sent to all processes linked to it, where `Pid` identifies the terminating process, and `Reason` is the atom `normal` if the process terminated normally, and is the reason associated with the failure otherwise. By default, if a process receives such a signal it does nothing if `Reason` is `normal`, and otherwise it terminates and propogates the signal to all other processes linked to it. Calls to built-in functions allow one to change this behaviour so that, e.g. any such signal is turned into a message that the process can receive and handle in the usual way.

Chapter 7

This chapter shows how to use the previously described mechanisms to design robust applications. For instance, it is shown how to use `catch` to guard a server against bad data and how to use the `link` to design reliable server processes.

Good program structure is stressed. Clients should communicate with servers through a well defined set of access functions. They should not interfere with the execution of the server. A server can monitor clients and take appropriate actions when they die by creating links to them and by trapping `EXIT` signals. This technique can be used to guarantee that vital processes of a system remain alive. The chapter also describes how to write a command shell that isolates the effects of commands by evaluating them in a separately spawned and linked process.

Chapter 8

This chapter discusses miscellaneous items like last call optimisation, unique references, code replacement, ports and process dictionaries

Last call optimization allows tail recursive programs to run in constant space. Almost all programs in the book are written in such a way that this optimization is possible. As such the possible runtime-overhead introduced by realizing a context change using recursion can be avoided.

References in Erlang are not at all like references in Standard ML. The call `make_ref()` returns a unique reference. The only thing which can be done with such references is to compare them for equality. They can, for example, be used to implement a method of communication with a server which provides 'end-to-end' confirmation that requests have been processed.

On-line code replacement is a feature which is particularly useful for telecom software: one does not want to stop a running system in order to replace old (probably buggy or slow) code by new (hopefully correct or faster) code.

In Erlang, a call across module boundaries is always made by listing the module and the function name together, thus `Mod:Func(Args)`. Such calls are always dynamically linked. This has the great advantage that Erlang code is quick to run, as there is no need for a separate linking phase. This is especially invigorating for the Erlang beginner. Dynamic linking has the disadvantage that code management becomes more difficult, although Erlang has some code management features to help with this problem.

Typically, a process is written as a function that receives a message, takes appropriate action, and then loops by making a tail recursive call to itself. If this tail recursive call explicitly mentions the module, then it will be dynamically linked: reloading the module will cause the tail call to be made to the old code rather than the new code. This depends crucially on two features of Erlang. The first is that the Erlang system always maintains two versions of the code, old and new. Thus, code can be reloaded at any time, but the switch to executing the new code happens at the tail recursive call. Second is that, as mentioned previously, all context is made explicit as arguments to tail recursive calls.

The resulting model of on-line code replacement is surprisingly simple to explain and to use. One can definitely say this is one of the points at which use of a functional language has provided great leverage!

Ports provide the basic mechanism for communicating with the external world. A port provides a byte-oriented communication channel between Erlang and the outside world. From the programmers point of view there is no basic difference between using those ports to communicate with processes of the outside world or with Erlang processes. A port can be opened and can be given a portname. It behaves like an Erlang process but can recognise only three kinds of messages: one for sending a list of bytes to an external object, one for closing the port and one for changing the internal process which is linked to the port. The process which is connected to the port can receive messages from the outside world by using a receive pattern which has a port name as its first component.

Ports can be opened for executing an external program and for connecting to an external resource such as a file. The kind of data which can be send to a port are output in packets of fixed length (1, 2 or 4 bytes) or output in packets of variable length. For spawned external processes one can also decide to use standard input/output for communicating with Erlang. Typically an external process will be a compiled C program which, besides executing the code which does the job the program is written for, makes use of simple routines to read and write buffers.

Dictionaries provide a restricted form of global state. Every process has an associated local dictionary, which is accessed by the built-in calls `put` and `get`. A dictionary allow access to the same global information in different functions of a process, and acts as a private context in which is destructively updated (via `put`). The usage of dictionaries must be done with much care and is normally discouraged.

By the end of Chapter 8 the reader has been introduced to all the features of Erlang. The remaining chapters provide extended programming examples.

Chapter 9

This chapter shows how to write a client which makes use of a database. The interface is through a number of access functions which encapsulate the underlying complexity of the actual database which does the job. Thus the database may simply be a process which maintains a dictionary or a two level database in which the processing of requests is forwarded by a first level server to second level auxiliary servers. It is also shown how to write a database which supports atomic transactions and roll-back. Furthermore it is shown how to introduce fault-tolerance. Again this is done by trapping `EXIT` signals. Finally, it is shown how to interface to an external database and to make use of an internal cache in order realise a safe way to update the external database.

Chapter 10

This chapter deals with system related functions. It explains how the standard Erlang operating system is structured. It also contains some very useful information about the primitives which allow the Erlang programmer to build systems which permit on-line code replacement. Code management is limited to two versions of the code of modules. Finally some standard issues as input/output and the standard shell are discussed. Processes access externel files through a file server process which is connected through a port to the underlying operating system.

Chapter 11

In this chapter the power of concurrent programming is shown through two real time applications: a lift control system and a sattelite control system. During the design of both systems much emphasis is put on the mapping of each concurrent activity onto a process of the system. In such a way one obtains, in a natural way, a mapping of a concurrent system to a concurrent functional program. The first example deals with lifts (or elevators, if you

are American). I had no problem to understand this example at first reading. It is also a very natural example: everybody has an idea about how lifts (ought to) behave. The second example deals with satellites. This example was more difficult for me to understand. Perhaps because it is not such a common known kind of system. Satellite systems are certainly known to one of the authors who has written a satellite control system for the Swedish Viking satellite in Fortran. The example shows how it *could* have been written in Erlang. Apart from the fact that it is a nice example of how to use Erlang, I personally think that the example is perhaps too much involved for an introductory book.

Chapter 12

This chapter deals with a POTS (Plain Ordinary Telephone Service) system. The example is based on a large Erlang program which is used to control an Ericsson MD10 PABX. This example is written in a very clear style. It uses uses the standard technique of modelling the states of a state machine as functions whose body starts with a receive expression so as to wait for messages to arrive. In other words, the design is *state* based: a system is designed in terms of its possible states and, in any state, a switch on the possible incoming messages is done.

Just for your information: a typical C++ program which has the same concurrent behaviour would probably be *message* or *action* based. Perhaps *method* based sounds more familiar to C++ programmers, though I prefer to use *action* based from the moment that concurrency issues are involved. In C++ one would also have to make use of some base class functionality (such as that of the task library) to deal with concurrency issues. There is, at the moment, absolutely no consensus in the telecom community about which approach (state or action based) is the best. By the way: it is, of course, also possible to write action based programs in Erlang. Chapter 15 elaborates on this style of programming.

Chapter 13

This is a separate chapter (written by Claes Wikström) which deals with ASN.1: a type description language which is standardized by the CCITT in order to facilitate the writing of OSI computer communication protocols. The chapter describes an ASN.1 to Erlang cross-compiler written in Erlang. It also shows how to make use of the compiler by means of an application which is developed with the aid of the compiler.

Chapter 14

This chapter shows how to use a graphics module, called `pxw`, which allows to access the X Window system. Again the port mechanism is used in order to communicate with the underlying X Window functionality. It is shown how to write a graphical pocket calculator and a small simulation of a TV camera and monitor.

Chapter 15

The last chapter is devoted to a very popular subject: OOP (Object Oriented Programming). Some of the concepts of OOP have natural corresponding concepts in Erlang (for example: modules). Other concepts can be implemented in Erlang in a number of different ways depending on the context in which they are used (for example: objects, abstraction and encapsulation).

Abstraction is implemented by using modules. Modules can be used to define interface functions which operate on an unspecified type.

Objects can be represented as data structures or as processes. Erlang data structures are *passive* objects which are *non-destructive*, and can be passed freely around for inspection (they cannot be modified). One can encapsulate data structures using the module system. Processes are *active* objects in the sense that they have their own thread of control. The only way to change their state is by letting them evaluate recursive functions calls with changed parameters. Encapsulation is provided through a message interface (which can itself, if one wishes, again be encapsulated using interface functions).

In the chapter it is also shown how to define *class* modules and how to implement inheritance using a `method` function which for each method which it can handle calls a `dispatch` function which tries to apply the method in the class and, if this fails, applies the method in the superclass (using the `superclass` function). Multiple inheritance can also be handled.

The authors claim that one advantage of *not* having a built-in OO mechanism with classes and inheritance is the possibility to modify the OO system to suit the application which is using it. Anyhow, a suitable combination of object orientation and functional programming is not realized yet and, in my opinion, there is still a lot of research to be done before one can come up with a satisfactory solution. Some results have already been obtained in this direction by Hudak (1992): he develops a framework to work with objects which are instances of mutable abstract data types. Also Jones and Hudak (1993) can be seen as a step towards using these results to introduce active objects.

In conclusion

I warmly recommend this book. Erlang boldly takes functional programming where it has not gone before: into the realms of concurrent, real-time, and robust programming. The designers have admirably succeeded in providing a practical language that achieves these ambitious aims. The book itself is a clear and thorough introduction to this work. Let us hope that Erlang will contribute to the widespread usage of functional programming languages for industrial applications and, as such, bridge the gap between the academic and the industrial world.

To reach their goal, the designers of Erlang have taken a pragmatic approach, sacrificing some useful features of functional languages, such as types, referential transparency, and lazy evaluation. How much of the same advantages can be achieved in a typed language? In a pure functional language? In a lazy language? Erlang poses these as intriguing challenges for the future.

Acknowledgement

I would like to thank Philip Wadler for giving valuable comments on this review.

References

Hudak, P. (1992) *Mutable Abstract Datatypes.* Yale Report, YALEU/DCS/RR-914, December.

Jones, M. and Hudak, P. (1993) *Implicit and Explicit Parallel Programming in Haskell.* Yale Report, YALEU/DCS/RR-982, August.

Peyton Jones, S. and Wadler, P. (1993) Imperative functional programming. In: *20th ACM Symposium on Principles of Programming Languages,* Charlotte, NC, January.

LUC DUPONCHEEL
Computing Science
Utrecht University
Utrecht
The Netherlands

J. Functional Programming 5 (4): 661, October 1995

Author Index to Volume 5

https://doi.org/10.1017/S0956796800001519 Published online by Cambridge University Press