Functional Programing

Philip Wadler, University of Glasgow[1]

Welcome to the first functional programming column! Functional programmers and formal methoders have much to say to each other, and this column is intended to further such communication.

Functional programming supports formal methods. A large number of proof systems and other formal methods packages have been implemented in functional languages. ML, the doyen of functional languages, takes its name from its first application as a meta language for writing theorem provers. Functional languages may also be suitable for writing executable specifications, for rapidly generating prototypes from specifications, or as the target language of a system that transforms specifications into efficiently executable code.

Conversely, formal methods support functional programming. Functional languages are well known for their amenity to mathematical reasoning, and formal methods can supply tools to support such reasoning. The application of mathematical laws to functional programs is not just a dream for the future – it underpins most of the techniques for generating efficient code from functional languages.

Contributions for future columns are invited – my postal and e-mail addresses appear on this page. I intend to interpret functional programming in the broad sense, ranging from lazy languages with absolutely no side effects, such as Haskell, to strict languages with disciplined use of side effects, such as Standard ML.

The first column deals with Erlang, a language of the latter sort. One impediment to the spread of functional programming is that most of the efforts have been academic in nature, and relatively little effort has gone into matters like programming environments that can make or break the suitability of a system for use in industry. Mike Williams of Ericsson has led the effort to develop Erlang, which is provides a convincing demonstration that functional languages can have industrial relevance. All of us interested in transfering ideas from academia to industry will be keeping an eye on Erlang to see how well it succeeds. Although it doesn't incorporate all the latest academic niceties – for instance, it lacks a type system – if successful it may pave the way for a generation of functional languages and other ivory tower innovations to stroll out of the classroom and into the software houses.

[1]Professor Philip Wadler, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: wadler@dcs.glasgow.ac.uk.

Functional Programming Column

Erlang – A Functional Programming Language for Developing Real-Time Products

Mike Williams, Ericcson Infocom Consultants AB[2]

1 Background

Software development is by far the largest part of the design costs for Ericsson's products. A single AXE-10 telephone exchange, for example, has on average software corresponding to 62 million lines of source code in C, C++, EriPascal and PLEX. Reducing the cost of software design, improving quality (fewer bugs) and cutting time to market are thus of paramount importance.

Using a programming technology which results in smaller and more understandable programs would improve the situation. A functional language is an obvious alternative.

We have designed the functional programming language 'Erlang' for this purpose. Using Erlang as the implementation language in several real-time system products has resulted in considerable reduction both in the size of programs and in the work required for software design.

2 Why do we need a new language?

Why not use ML or Haskell or some similar language? To answer this question we must describe the problems which need to be solved.

Massive
: The sort of system we are considering are *massive*, which means that they contain tens of millions of lines of source code.

Long Lived
: Systems must survive and evolve over a long period of time. For example, the AXE-10 system, which is still a best seller today was designed in the mid 70's. The technology we use now must be able to inter-work both with older software and with new technology which no doubt will turn up in the future.

[2]Mike Williams, Ericsson Infocom Consultants AB, Erlang Systems Division, Box 1153, S-164 22 Kista, Sweden; and Ellemtel Utvecklings AB, Box 1505, S 125 25 Älvsjö, Sweden. E-mail: mike@erix.ericsson.se.

Distributed

Systems are made from a large number of different types of computers running different operating systems and communicating with each other by a variety of mechanisms.

Concurrent

Thousands of different activities must be handled at the same time. These activities (processes) must be able to communicate with each other. We are not primarily interested in parallelism - splitting up a computation onto several different computers to gain speed - we are interested in handling the massive concurrency we already have!

Robust

Failure of a part a system, due to a hardware failure or a software fault, should have as little effect on the whole system as possible. It must be possible to speedily recover from failures and go on running despite the fact that parts of the system may be out of service.

Non Stop

Some systems cannot be stopped to make software changes, to correct bugs, to add new software or to remove unused software. It must thus be possible to do this while the system is running and to ensure the software changes are made in such a way that the services offered are disturbed as little as possible.

Real-time

Many actions must be completed within a fixed time after receiving a stimulus. Language implementations which cause significant pauses in execution, for example for garbage collection, are thus unsuitable.

Very few languages or implementations of languages and operating systems meet the criteria above. As far as we are aware, Erlang is the only functional language designed and implemented with these criteria in mind, criteria which apply to an increasingly large class of real-time systems apart from telecommunication applications.

3 Erlang - The Language

For a detailed description of Erlang see [1]. It has been unkindly said that Erlang is a functional language without proper functions and a logic language without logical variables. More kindly (perhaps) it has been described as a sort of concurrent Lisp.

The following points give a very concise summary of Erlang:

- Dynamically typed. Single assignment variables
- All choice / selection done by pattern matching with shallow guards
- Recursion equations
- Explicit light weight processes part of the language
- Relatively free from side effects
- Asynchronous message passing between processes. Message reception is selective, a process can wait for messages which match a number of patterns and cause other messages to be queued.
- Primitives for detecting run-time errors, including errors in other processes
- Transparent cross-platform distribution. The message passing primitives are the same for messages sent between processes on the same processor and between processors. The same applies to the error detection primitives.
- Real-time garbage collection
- Modules, export and import declarations
- Dynamic code replacement - you can replace code in running systems.
- Foreign language interface

The language is eager. It is also very compact and easy to learn. Everything which was found to impair efficiency or which was difficult to implement in a distributed non homogeneous environment was excluded from the language. The language thus does not support Currying, higher order functions, lazy evaluation, ZF comprehension or deep guards.

The lack of higher order functions is alleviated by the built-in function apply, a function supplied as an argument is applied to a list of arguments.

4 Implementations

4.1 JAM Implementation of Erlang

In the JAM (Joe's Abstract Machine) implementation. Erlang [2] is compiled to code for a virtual machine, the JAM. The Erlang compiler is itself written in Erlang. The JAM is implemented by an emulator written in "C". The emulator also implements code loading, memory management, concurrency, garbage collection, distribution (at present with TCP/IP) and built-in functions.

Each Erlang module is compiled to a separate file of instructions which can be executed by the emulator. The emulator loads these files as they are needed. Modules can also be pre-loaded. Modules which are already loaded can be replaced by modified modules - even if these modules are being used.

Garbage collection is done on a per internal Erlang process basis. This means that pauses for garbage collection will be very short provided that there are no very large processes.

Supported versions of this implementation are available (at present) on

- SUN Workstations (both SunOS4 and Solaris 2)
- RS 6000 / AIX
- Interactive Systems UNIX on Intel 486
- VXWorks on Force 68040 and SPARC
- HP 9000 HP-UX

Unsupported versions are available for Intel 486 on MSDos.

This implementation is the most widespread implementation of Erlang and the one which is most suitable for developing and testing software. The code emulated by the JAM (i.e. the output of the compiler) is very compact.

4.2 VEE Implementation of Erlang

The VEE (Virding's Erlang Engine) implementation is similar to the JAM, but is based on a different virtual machine and compiler. A real-time, two space, copying garbage collector using a modified Baker scheme with Brook's optimisation is used. This gives better real-time performance than the JAM.

The VEE implementation is about twice as fast as the JAM implementation.

Supported versions of this implementation are available on

- SUN Workstations (both SunOS4 and Solaris 2)

The code emulated by the VEE (i.e. the output of the compiler) is still compact, but not as compact as in the JAM implementation.

4.3 BEAM Implementation of Erlang

In the BEAM [3] (Bogdan's Erlang Abstract Machine) implementation, Erlang is compiled (by a compiler written in Erlang) into C code. The C code is in turn compiled by the GCC, C-compiler and linked with a run-time kernel. The linker has been specially written for Erlang and allows incremental loading of code into running systems in the same way as the other implementations. Garbage collection is done in the same way as in the JAM implementation.

Supported versions of this implementation are available on:

- SUN Workstations (both SunOS4 and Solaris 2)
- VXWorks on Force 68040 and SPARC

This is the fastest implementation of Erlang, its speed is comparable with non optimised "C" code when executing sequential code. The code size is about five times larger than that of the JAM implementation.

5 Tools to Aid Erlang Programming

Apart from the compilers and run time system needed for the implementations described above, there are many tools available to the Erlang programmer. These include

Libraries:
: Trees, lists, strings, sets, code handling, error handler, error logger, expression evaluator, file system interface, global registration of processes, formatted io, mathematical functions, networking, expression parsers, process groups, load distribution, random numbers, and socket interfaces.

Interface to Foreign Languages:
: At present only an interface to C is supported. A library written in Erlang transforms Erlang terms to (and from) a binary representation. A library written in C can be used to transform *structs* in C to (and from) this binary representation.

Tools:
: Code coverage and profiling, windows based debuggers, make, pretty printer, shell, X-windows based point and shoot interface, parser generator, lexical analyser generator and Erlang mode for emacs.

Special Tools:
: Compiler for ASN.1. SDL to Erlang Compiler Interface to X-Windows.

6 Development of Erlang

Erlang was developed at Ellemtel, a jointly owned subsidiary of Ericsson and Telia (Swedish Telecom). Work with Erlang started with a series of experiments with the goal of determining how software design of large real-time systems could be made easier [4]. The main conclusion of this study was that so called declarative[3] languages resulted in the shortest and clearest programs. At that time we were very involved with Prolog and Parlog. In fact one of the earliest references to

[3]The term declarative is deliberately not defined here!

Erlang is the STRAND book [5]. Erlang has inherited syntax and data structures from Prolog.

It soon became clear that many things in Prolog were unsuitable for our applications. Real-time systems, if they are to do anything useful, have side effects, for example in controlling hardware. Backtracking in Prolog must thus be severely restricted when changes have been made. As a simple example, in a telephone system we may cause a phone to ring. We cannot backtrack and cause the phone to stop ringing, someone may have heard it ringing!

Controlling hardware means that operations must be done in a predecided order. When Parlog is used, extra effort was required to ensure sequentiallity. Languages with unconstrained parallelism are too parallel.

Using logical variables would be extremely difficult to implement efficiently in a distributed system and would imply all sorts of hidden information passing between processes. Logical variables were thus excluded.

All the implementations we studied had some form of stop and copy garbage collection which were unsuitable in real-time systems.

It was also important to us that concurrency is explicit. In our applications we control a very large number of activities at the same time. For programs to be clear, it is important that each real life asynchronous activity be represented by a process in our system. It is also important that processes are light weigh. The computational effort in creating and scheduling processes and sending messages between processes should be small. The memory required by each process should be as small as possible and vary dynamically as the process expands and contacts.

Erlang was being developed and used at the same time as the application described in [6]. This application was a prototype of a new PABX (Private Automatic Branch Exchange). The designers of this prototype were mainly interested in the software architecture of their system and needed a new language which made it easy for them to perform experiments. The users working in this project developed several thousands of lines of code despite the fact that we were changing the language at the same time. These users contributed much to the design of Erlang. When we found that constructs in the language were not being used, we removed them, when we found that users were writing unclear or convoluted code we added constructs which alleviated their problems and made the code clear. This has resulted in a very small and compact language which we have found easy to use and to teach to new users.

7 Use of Erlang in Ericsson

For the first few years Erlang was mainly used to build prototypes. Ericsson has been involved in six projects in the European RACE programme where Erlang has been used to build demonstrators and prototypes. An example is BIPED [7].

Erlang has been used to build many prototypes internally in Ericsson. Notable among these are a private telephone exchange[6], an office cordless telephone system [8] and a prototype to demonstrate user mobility in telephone networks [9]. Erlang has been used to build countless other prototypes ranging from optical cross connect controllers to paging systems.

The success of these prototypes has resulted in Erlang being used as the implementation language for several products which are at present being developed. For commercial reasons the details of these products cannot be mentioned here.

The largest of these products, a complex and very specialised switching system, is being developed by a team of about 25 software engineers. So far about 300 000 lines of Erlang source code have been developed for this product, which is probably the largest and most complex functional program in the world today. It has been estimated that this would have required about three million lines of source code if this had been programmed in a conventional language - and would have required a much larger team.

It is important to note that using a language such as Erlang effects far more than the coding phase of systems development. The semantic gap between specifications and programs is much smaller than if languages such as C, C++ or PLEX are used. Testing also becomes easier, there is less code to test, the code is easy to understand and testing can be done on the symbolic "Erlang" level. The total work required to produce software, from first specifications to testing has, in some projects, been reduced by as much as an order of magnitude.

Erlang is thus spreading slowly into Ericsson products, starting with the newer less establish products. It has proved difficult to get Erlang used in older established products for which there already is a large volume of code written by an organisation which reflects a well defined methodology.

8 Use of Erlang Outside Ericsson

The JAM implementation is available free of charge, on a non commercial licensed basis to universities and similar organisations. About 80 systems have been delivered on this basis. Erlang is now being taught at several universities as far afield as Bejing, Melbourne and Stockholm.

Ericsson has decided to sell Erlang on a commercial basis. This task has been assigned to Ericsson Infocom Consultants AB. The Erlang Systems Division has been set up to support, maintain, develop and sell Erlang both within Ericsson and to external customers. A team of consultants is available to help customers with the use of Erlang.

Courses are held regularly in Sweden and abroad. These have been attended by both a large number of Ericsson employees and by people from other companies.

Enquires about obtaining Erlang, both for commercial and noncommercial use, should be address to Ericsson Infocom Consultants. AB, Erlang Systems division.

9 References

[1] Concurrent Programming in Erlang, Joe Armstrong, Mike Williams and Robert Virding, Prentice Hall, 1993.
[2] Implementing a functional language for highly parallel real time applications, Joe Armstrong, Bjarne Dcker, Mike Williams and Robert Virding, Software Engineering for Telecommunication Switching Systems and Services, March 30 - April 1, 1992, Florence.
[3] Turbo Erlang, Bogumil Hausman, International Logic Programming Symposium, October 26-29, 1993 Vancouver.
[4] Experiments with Programming Languages and Techniques for Telecommunication Applications, Bjarne Dcker, Nabiel Elshiewy, Per Hedeland, Carl Wilhelm Welin and Mike Williams, Software Engineering for Telecommunication Switching Systems, April 14-16 1986, Eindhoven.
[5] Chapter 3 Programming Telephony (Armstrong and Virding) from Strand - New Concepts in Parallel Programming Ian Forster and Stephen Taylor, Prentice Hall, 1990.
[6] New Technology for Prototyping New Services, Kerstin dling, Ericsson Review no. 2 1993
[7] Control Switching Implementation of the BIPED Demonstrator, F. Monfort Second Australian Conference on Telecommunications Software, Sydney 1993
[8] Prototyping Cordless Using Declarative Programming, Ingemar Ahlberg, John-Olof Bauner and Anders Danne, Ericsson Review no. 2 1993
[9] A Prototype Demonstrating User Mobility and Flexible Service Profiles, Jan van der Meer, Ericsson Review no 1 1994.