Published in: *The Practical Application of Prolog* - 1 – 3 April 1992.
Institute of Electrical Engineers, London

Use of Prolog for developing a new programming language

J. L. Armstrong, S. R. Virding, M. C. Williams
Computer Science Laboratory
Ellemtel Telecommunications Systems Laboratory
Box 1505
S - 125 25 Älvsjö
Sweden
joe@erix.ericsson.se

January 27, 1992

Abstract

This paper describes how Prolog was used for the development of a new concurrent real-time symbolic programming language called Erlang.

Erlang was developed by first building a prototype in Prolog - the prototype was used by a user group to test their reactions to the language. As time passed many features were added (and removed) from the interpreter and eventually the language reached a level of maturity where it was decided to try it out on a significant problem.

About 3 years and some 20,000 lines of Erlang later, performance became an issue - we wrote Prolog cross compilers from Erlang to various concurrent logic programing languages followed by a direct implementation of Erlang itself. The direct implementation of Erlang was loosely based on the WAM and made by writing a Prolog compiler from Erlang to a new abstract machine and an emulator for the abstract machine in 'C'. The instruction set for the abstract machine was first prototyped in Prolog - finally the compiler was re-written in Erlang, thus totally removing any dependency on Prolog.

This paper describes some of the key events which lay between the simple prototype and the current version of the language.

1 Introduction

This paper describes the development of the Erlang programming language [AR90].

Although we did not know it at the time, we were unwittingly following the advice of Brooks [BR86], who argued that software systems should be incrementally *grown* rather than designed in a conventional manner, and that rapid prototyping should be used as the design methodology.

It was never our intention to develop a new programming language. Erlang emerged as a *side effect* of an attempt to find an elegant way to program a PABX (Private Automatic Branch Exchange) for simple telephony applications. The ground work for this is described in [DA86], and some of the conclusions of the initial programming experiments in [AR86].

In writing this paper we have followed the advice of Parnas and Clements [PA86], that is, we present the development of our language *as if* it had been the result of a rational design process - whereas in fact the real design process is virtually impossible to describe.

The success of the development is in large measure due to Prolog (indeed had it not been for Prolog it is doubtful whether a language such as Erlang would have emerged). Prolog provided the kind of interactive environment which could accommodate the rapid changes in direction which occurred during the development process. At a finer level of detail the implementation of Prolog provided the inspiration necessary to make an efficient version of Erlang. The *errors* in Prolog (things which we would have done differently) provided inspiration for how not to do certain things!

Erlang supports a number of mechanisms not found in Prolog. Erlang is a concurrent, real-time, distributed, functional language suitable for building large embedded real-time systems. It has modules to support programming in the large, code can be changed in a running system *on the fly* and it has sophisticated error trapping mechanisms, which allow the construction of highly robust software systems. More details are available in [AR89, AR90].

The language started as a simple meta interpreter which added notions of concurrency and various error handling mechanisms on top of Prolog. It finished as a programming language in its own right. The ideas behind the Erlang interpreter are described in section 2.

Having built a simple interpreter for Erlang we were keen to use it on real problems. A group of application programmers started using it for prototyping real telephony applications. Some of the results are described in [PE91].

The influence of the user group on the development of the language is described in section 3.

As the language developed it obtained its own syntax and its relation to Prolog weakened - the development of the language is described in section 4.

Up to now the development had been highly informal - the interpreter (which was changed almost daily) was the only definition of the language. As the interpreter matured and changes became less frequent, proper documentation became necessary. This change is described in section 5.

After about 3 years, the user group was satisfied with the resultant language. A study was made of approximately 20,000 lines of Erlang which implemented various features in a telecommunications switching system. These features had previously been implemented in conventional languages. Program design using Erlang required between 7 and 22 times less work. Performance of the language now became the primary concern. While satisfied with the expressive power of our language, our users required an implementation which was at least 40 times faster that the original interpreter.

Section 6 describes our attempts to speed up the implementation of the language. The first attempt was to cross compile Erlang to Prolog or STRAND.

Inspiration for the next stage in development came from the WAM [WA83] - since Erlang had grown out of Prolog it seemed natural that the techniques used to make fast Prolog systems should also be applicable to Erlang. We describe how we made a fast implementation of Erlang.

Section 7 describes the current status of the language.

2 First Interpreter

We start with a discussion of a simple Prolog interpreter for an extremely restricted subset of Erlang.

The first interpreter was a simple Prolog meta interpreter which added the notion of a suspendable process to Prolog. This interpreter was the starting point of a series of experiments to try to find out what features should be included in a language suitable for programming robust concurrent real-time applications.

The first interpreter was rapidly modified (and re-written) to include the functionality identified in a series of detailed studies performed to establish the desirable characteristics of a language for programming telephony applications [DÄ86].

At this stage the entire interpreter could be rewritten in a matter of days and simple extensions could be made in hours.

In the following sections we describe the basic techniques used in the first interpreter.

2.1 Suspending a computation

The first step is to program a *suspendable* computation. We want to write a meta interpreter which can be suspended at any point in time - it should be possible to *resume* the computation later.

Consider the *vanilla* interpreter `solve/1` which is written as follows:

```
solve((A,B)) :-
     solve(A),solve(B).
solve(A) :-
     builtin(A),
     call(A).
solve(A) :-
     rule(A, B),
     solve(B).
```

The intermediate stages of the computation are inaccessible to the programmer. The goal `solve/1` terminates when a solution has been obtained and it is not possible to suspend the computation while it is taking place.

The simplest way to provide a suspendable computation is to keep all unsolved goals in a list, and successively take the goal at the head of the list, solve it, then solve the goals remaining in the tail of the list, together with any new goals which may have been generated in solving the goal at the head of the list. This can be achieved as follows:

```
reduce([]).
reduce([{X}|T]) :-
     call(X),
     !,
     reduce(T).
reduce([Lhs|More]) :-
     eqn(Lhs, Rhs),
     append(Rhs, More, More1),
     !,
     reduce(More1).
```

This reduces a list of items `[Item1, Item2, ....]` where is assumed that items contained in curly braces "`{`" and "`}`" are Prolog goals, and that all other items are user defined rules.

As an example we show how the well known näive reverse and factorial functions can be written in terms of `eqn/2`:

```
eqn(nrev([H|T], Z), [nrev(T, T1), concat(T1, [H], Z)]).
eqn(nrev([], []),   []).

eqn(concat([H|T], T1, [H|T2]), [concat(T, T1, T2)]).
eqn(concat([], T, T), []).

eqn(fact(0, 1), []).
eqn(fact(N, F), [{N1 is N - 1}, fact(N1, F1), {F is N * F1}]).
```

`reduce/1` works as expected:

```
?- reduce([fact(3,F), {write(result(F)), nl}]).
result(6)
F = 6 ?
..
?- reduce([nrev([a,b,c], R), {write(result(R)), nl}]).
result([c,b,a])
R = [c, b, a] ?
...
```

If we were to trace successive iterations of `reduce/1` for the factorial example, the following would be obtained:

```
reduce([fact(3,X1),{write(result(X1)),nl}]).
reduce([{X3 is 3-1},fact(X3,X2),{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([fact(2,X2),{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([{X4 is 2-1},fact(X4,X3),{X2 is 2*X3},{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([fact(1,X3),{X2 is 2*X3},{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([{X5 is 1-1},fact(X5,X4),{X3 is 1*X4},{X2 is 2*X3},{X1 is 3*X2},
{write(result(X1)),nl}]).
reduce([fact(0,X4),{X3 is 1*X4},{X2 is 2*X3},{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([{X3 is 1*1},{X2 is 2*X3},{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([{X2 is 2*1},{X1 is 3*X2},{write(result(X1)),nl}]).
reduce([{X1 is 3*2},{write(result(X1)),nl}]).
reduce([{write(result(6)),nl}]).
reduce([]).
```

The list of goals which remains to be reduced at any particular point in the computation is a normal Prolog data structure which can be manipulated in any way required by the user - in particular we may decide to suspend the computation after a fixed number of reductions.

2.2 Suspending the computation after a fixed number of reductions

The predicate `reduce/3` is similar to `reduce/1` with the difference that it will always terminate before a fixed number of reductions has been performed. The goal:

```
reduce(GoalList, Reductions, Result)
```

reduces the list of goals in `GoalList` binding `Result` to either `terminated(N)` or `continuation(More_Goals)`. `terminated(N)` means that the computation completed with `N` reductions, `continuation(More_Goals)` is returned after 20 reductions have been performed if there are still outstanding goals in the goal list, `Reductions` is used to count how many reductions have occurred:

```
reduce([], N, terminated(N)) :- !.
reduce(Goals, 20, continuation(Goals)) :- !.
reduce([{X}|T], Reds, Result) :-
     call(X),!,
     Reds1 is Reds + 1,
     reduce(T, Reds1, Result).
reduce([Lhs|More], Reds, Result) :-
     eqn(Lhs, Rhs),
     append(Rhs, More, More1),
     Reds1 is Reds + 1,
     reduce(More1, Reds1, Result).
```

So for example:

```
?- reduce([nrev([a,b,c,d], R), {write(result(R)),nl}], 0,Result),
   write(reduce(Result)).
result([d,c,b,a])
reduce(terminated(16))
R = [d,c,b,a],
Result = terminated(16) ?
...
?- reduce([nrev([a,b,c,d,e], R), {write(result(R)),nl}], 0, Result),
   write(reduce(Result)).
reduce(continuation([concat([],[a],X1),
                {write(result([e,d,c,b|_1225])),nl}
               ]))
R = [e,d,c,b|X1],
Result = continuation([concat([],[a],X1),
                 {write(result([e,d,c,b|X1])),nl}]) ?
...
```

We can, of course, restart the computation represented by `continuation(New_Goals)` by solving `reduce(New_Goals, 0, Result1)` later.

2.3 A multi task scheduler

The reduction mechanism in the previous section can be used as the basis of a simple "round robin" scheduler (`multi_reduce/1`) - this takes a list of tasks of the form `job(N, Goals)`. The scheduler allows each job to run for at most 20 reductions, it then goes on to the next task etc. until no more tasks remain. `N` is the name of the task, `Goals` is a goal list representing the task.

```
multi_reduce([]).
multi_reduce([job(N, Goals)|T]) :-
     write(starting(N)),nl,
     reduce(Goals, 0, Result),
     multi_reduce(Result, N, T).

multi_reduce(terminated(_), N, T) :-
     write(terminating(N)),nl,
     multi_reduce(T).
multi_reduce(continuation(Goals), N, Job_queue) :-
     write(suspending(N)),nl,
     append(Job_queue, [job(N, Goals)], New_job_queue),
     multi_reduce(New_job_queue).
```

We have added a few write statements so we can observe what happens:

```
? multi_reduce([
     job(1,[nrev([a,b,c,d,e,f,g,h],R), {write(job1(R)),nl}]),
     job(2,[nrev([1,2,3,4,5], R1), {write(job2(R1)),nl}]),
     job(3,[fact(10, Fact), {write(job3(Fact)),nl}])
     ]).

starting(1)
suspending(1)
starting(2)
suspending(2)
starting(3)
suspending(3)
starting(1)
suspending(1)
starting(2)
job2([5,4,3,2,1])
terminating(2)
starting(3)
job3(3628800)
terminating(3)
starting(1)
job1([h,g,f,e,d,c,b,a])
terminating(1)
```

Here we see that the three goals `nrev([a,b,c,d,e,f,g,h], R)`, `nrev([1,2,3,4,5], R1)` and `fact(10, Fact)` were executed concurrently.

This is a form of pseudo concurrency since we only have one processor. The maximum number of reductions which are allowed before a task is suspended is analogous to a time slice, and the process of swapping tasks, to a context switch. Each single *thread* of computation is analogous to a *process*.

2.4 An interpreter for a functional style of programming

If we ignore for a moment context switching and return to the basic reduction cycle we can define `reduce1/1` which supports a more functional style of programming:

```
reduce1([]).
reduce1([Var = Rhs|T]) :- !,
     reduce1([Rhs, '$bind'(Var)|T]).
reduce1([return(Value), '$bind'(Var)|T]) :- !,
     Var = Value,
     reduce1(T).
reduce1([{X}|T]) :-
     call(X),!,
     reduce1(T).
reduce1([write(X)|T]) :- !,
     write(X),
     reduce1(T).
reduce1([nl|T]) :- !,
     nl,
     reduce1(T).
reduce1([Lhs|More]) :-
     eqn1(Lhs, Rhs), !,
     append(Rhs, More, More1),
     reduce1(More1).
```

Note that in this version of our interpreter `write/1` and `nl/0` are now primitive functions *in* our meta language, i.e. they can be written directly (not enclosed in curly braces) - adding new primitives to the meta language can be easily achieved as for `write/1` and `nl/0`.

Returning to our factorial and näive reverse example these can now be written:

```
eqn1(fact1(0), [
     return(1)
     ]).
eqn1(fact1(N), [
     {N1 is N - 1},
     F1 = fact1(N1),
     {Result is N * F1},
     return(Result)
     ]).
```

and:

```
eqn1(nrev([H|T]) , [
     T1 = nrev(T),
     Result = concat(T1, [H]),
     return(Result)
     ]).
eqn1(nrev([]), [
     return([])
     ]).

eqn1(concat([H|T], T1), [
     T2 = concat(T, T1),
     return([H|T2])
     ]).
eqn1(concat([], X), [
     return(X)
     ]).
```

Which behaves as expected:

```
?- reduce1([ X = fact1(4), write(X), nl]).
24

X = 24 ?
...
?- reduce1([ X = nrev([a,b,c,d]), write(X), nl]).
[d,c,b,a]

X = [d,c,b,a] ?
```

2.5 A functional language with processes and message passing

In our final and most complex example we combine the ideas of the previous two sections to form an interpreter `reduce2/4` which supports a functional style of programming, allows process creation and simple message passing between processes.

Each process is represented by a data structure of the form: `job(Id, Goals, MailBox)` where `Id` is the name of the process, `Goals` is a list of unsolved goals representing the computation and `MailBox` is a list of messages which have been sent to, but not yet received by the process.

The goal `reduce2(Goals, Id, MailBox, Jobs)` represents the current state of a simple multi tasking operating system. `Goals, Id` and `mailBox` represents the currently executing process and `Jobs` a list of all suspended processes.

The list of such processes we call an *environment*

```
reduce2([], _, _, []) :- !,
     write(stopped),nl.
reduce2([], MyId, _, [job(Id, Goals, Msgs)|T]) :- !,
     write(terminating(MyId)),nl,
     write(resuming(Id)),nl,
     reduce2(Goals, Id, Msgs, T).
reduce2([spawn(Id, Goals)|T], MyId, MyMsgs, Env0) :- !,
     write(spawning(Id)),nl,
     append(Env0, [job(Id, Goals, [])], Env1),
     reduce2(T, MyId, MyMsgs, Env1).
reduce2([send(Id, Msg)|T], MyId, MyMsgs, Env0) :-
     send(Id, Msg, Env0, Env1),
     !,
     reduce2(T, MyId, MyMsgs, Env1).
reduce2([receive|T], MyId, [Message|More], Env) :- !,
     reduce2([return(Message)|T], MyId, More, Env).
reduce2([receive|T], MyId, [], Env) :- !,
     write(suspending(MyId)),nl,
     append(Env, [job(MyId, [receive|T], [])], Env1),
     reduce2([], none, [], Env1).
reduce2([Var = Rhs|T], MyId, MyMsgs, Env) :-
     !,
     reduce2([Rhs, '$bind'(Var)|T], MyId, MyMsgs, Env).
reduce2([return(Value), '$bind'(Var)|T], MyId, MyMsgs, Env) :- !,
     Var = Value,
     reduce2(T, MyId, MyMsgs, Env).
reduce2([{X}|T], MyId, MyMsgs, Env) :-
     call(X), !,
     reduce2(T, MyId, MyMsgs, Env).
reduce2([Lhs|More], MyId, MyMsgs, Env) :-
     eqn4(Lhs, Rhs), !,
     append(Rhs, More, More1),
     reduce2(More1, MyId, MyMsgs, Env).


send(Id, Msg, [job(Id, Goals, Messages)|T],
           [job(Id, Goals, Messages1)|T]) :-
     !,
     append(Messages, [Msg], Messages1).
send(Id, Msg, [H|T], [H|T1]) :-
     send(Id, Msg, T, T1).
```

The meta language primitive `spawn(Id, Goals)` schedules a new process with name `Id` and goal list `Goals`. `send(Id,Message)` sends the message `Message` to the process `Id`. `receive` is a function which returns the first message in the mailbox of the process.

The scheduling algorithm used in the interpreter is extremely simple, all processes proceed until they either terminate or until they try to receive a message and no messages have been sent to the process.

In reality we used a more sophisticated scheduling algorithm which combined time slices together with context switching when a process could no longer proceed.

The relations `eqn2/2` define a new meta language - a simple program in terms of this meta language could be the following:

```
eqn2(go,[
     spawn(sender,[sender(5)]),
     spawn(catcher, [catch])
     ]).

eqn2(sender(0), [
     send(catcher, stop)
     ]).
eqn2(sender(N), [
     {write(sending(pip(N))), nl},
     send(catcher, pip(N)),
     {N1 is N - 1},
     sender(N1)
     ]).

eqn2(catch,[
     X = receive,
     {write(received(X)), nl},
     catch(X)
     ]).
eqn2(catch(stop),
     []).
eqn2(catch(_),[
     catch
     ]).
```

This creates two process called `sender` and `catcher`. `sender` sends five messages to `catcher` and then terminates. `catcher` receives these messages and terminates when it receives the termination message `stop`:

```
| ?- reduce2([go], startup, [], []).
spawning(sender)
spawning(catcher)
terminating(startup)
resuming(sender)
sending(pip(5))
sending(pip(4))
sending(pip(3))
sending(pip(2))
sending(pip(1))
terminating(sender)
resuming(catcher)
received(pip(5))
received(pip(4))
received(pip(3))
received(pip(2))
received(pip(1))
received(stop)
stopped
```

2.6 Properties of the first interpreter

The interpreters of the previous sections show how we can establish a processed based model of computation in Prolog. There are many things wrong with the above interpreters, the meta languages are *user hostile* and any mistakes in the user meta language programs will crash the interpreters in a number of bizarre ways.

The interpreters are themselves written with appalling lack of concern for efficiency - our actual interpreter used difference lists to eliminate all `append`s from the program and was somewhat more efficient (though less clear!) than the versions we have presented.

In the early stages, languages features could be implemented in a few hours - invariably all the discussions preceding any language change took much longer than the time taken to implement the change. During the development period the language interpreter itself underwent a very large number of small changes - and several large changes which required total re-writes.

The interpreter itself proved extremely robust and highly portable. The first version was a NU Prolog version running on a VAX 11/750. It was then ported to Quintus Prolog on the same machine. The user group had access to PC's so it was ported to ALS Prolog Running on PC/AT's. We then ported it to SICStus Prolog and changed from our VAX to SUN3/60's and then to SICStus Prolog running on SPARC's.

None of the above changes of Prolog system or computer caused us more than momentary and easily solved problems. We had far more problems due to the incompatibility of operating systems (especially between MS-DOS and UNIX) than we ever had between the different Prolog systems (and this despite the fact the Prolog is still not a standardized language!).

Using the techniques described in sections 2.1 - 2.5 and the syntax described in section 4.1 we wrote the first usable interpreter for a concurrent language - this language had the following characteristics:

- concurrent processes
- selective message passing
- process groups
- error signaling and trapping mechanism based on process groups
- *in line* prolog code
- time slices
- round robin scheduling
- tracing interprocess messages
- debugging individual processes
- I/O to virtual channels (Ports)

All of which took approximately 1100 lines of Prolog code.

The first interpreter was then used to experiment with different language constructs, scheduling mechanisms, programming styles etc. aimed at finding a convenient and elegant way of solving simple telephony programming problems.

For these experiments we used an Ericsson MD110 PABX [MD82] (Private Automatic Branch Exchange) which we controlled from our Prolog interpreter.

After a period of experimentation, during which we made rapid, almost daily changes to the interpreter, meta language and programs in the meta language, a style of programming and version of the interpreter emerged which we felt was good enough to be used in a wider context.

3 User Group

Once the first interpreter had stabilized a user group was formed. This was a group of application programmers (outside the computer science laboratory - CSL) who were interested in experimenting with new architectures and ways of programming a small PABX.

The formation of the user group forced a measure of stability on the development process. CSL members were obliged to deliver their ideas according to a time schedule and produce working and documented versions of the software.

For over a year CSL and user group members met once or twice a week in an attempt to understand each other's problems - this process was invaluable, the users with their long experience of applications programming suggested many valuable changes to the language (which were easily incorporated in the interpreter) - CSL members were able to convince the applications programmers why certain imperative styles of programming led to problems and should consequently be avoided. This process, while at times somewhat frustrating was in retrospect one of the key factors which lead to the subsequent acceptance of the language.

It is interesting to note that once the user group started using the language the language started changing again - despite the fact that the language had *converged* prior to releasing it!.

It appears that the language *converged* twice, firstly to fulfill the internal need of the CSL developers, secondly as feedback was obtained from the users.

4 Development of the language

4.1 Syntactic development

We take näive reverse and factorial as examples. The syntax used in section 2.4 was:

```
eqn1(fact1(0), [
      return(1)
      ]).
 eqn1(fact1(N), [
      {N1 is N - 1},
      F1 = fact1(N1),
      {Result is N * F1},
      return(Result)
      ]).

 eqn1(nrev([H|T]) , [
      T1 = nrev(T),
      Result = concat(T1, [H]),
      return(Result)
      ]).
 eqn1(nrev([]), [
      return([])
      ]).
```

By defining the infix operator `--->` we can write these as valid Prolog data structures and omit the functor symbol `eqn1`:

```
fact1(0) --->
     return(1).
fact1(N) --->
     {N1 is N - 1},
     F1 = fact1(N1),
     {Result is N * F1},
     return(Result).

nrev([H|T]) --->
     T1 = nrev(T),
     Result = concat(T1, [H]),
     return(Result).
nrev([]) --->
     return([]).
```

The above syntax was the first syntax which *real users* were subject to. The disadvantage of this syntax is that error messages, run time diagnostics etc. are in terms of Prolog data structures - not Erlang.

After the language had been in use for about year and reached relative stability we wrote a parser for Erlang which gave the language a syntax of its own and allowed us to invent syntaxes and structures which were not valid Prolog.

In the new syntax:

```
nrev([H|T]) ->
     T1 = nrev(T),
     concat(T1, [H]).
nrev([]) ->
     [].

fact(0) ->
     1.
fact(N) ->
     N1 = N - 1,
     F1 = fact1(N1),
     N * F1.
```

After a short time with the new syntax we adopted a purely functional notation which avoided the need for superfluous temporary variables; in the current syntax we now write:

```
nrev([H|T]) -> concat(nrev(T), [H]);
nrev([])    -> [].

fact(0) -> 1;
fact(N) when N > 0 -> N * fact(N - 1).
```

The *effort* involved for each of these syntaxes can be seen from the following:

- Original Prolog Clauses: 0 lines of prolog code
- Infix operator `--->` plus code to read and convert Erlang functions to a normalized form - 91 lines of Prolog code
- Parser in Prolog - 1057 lines of Prolog (629 lines in the parser, 262 in the tokeniser, 166 in the pretty printer).

The parser was first written as a Prolog DCG but the idea of using a DCG was rapidly discarded due to problems concerned with providing meaningful and precise error diagnostics. The solution to this was to completely flatten the Erlang grammar into an LL(1) grammar and write a conventional recursive descent parser for the language. At the same time we removed the possibility for users to define their own infix operators.

4.2 Language changes suggested by the new syntax

Moving to a fully parsed version of the language suggested the introduction of the following features:

- tuples (written `{Item1, Item2, Item3, ...}`) for storing fixed numbers of items.
- functions with zero arity are denoted `func()` whereas atoms with the same name have no parentheses.
- a module system
- function guards

5 Semantic embedding

Changing from a Prolog to a new syntax had more than a cosmetic effect. Because it was no longer Prolog, we were forced to think more in terms of the meaning of a language construct, rather than to just rely on the meaning of the statement when it was translated into the equivalent Prolog.

Often the semantics of an Erlang expression were not the result of a conscious design decision but were the accidental result of the implementation in Prolog - we call this phenomena *semantic embedding* - the semantics of Prolog become accidentally embedded in Erlang.

We first became aware of this phenomena when we wrote the Erlang parser - at a later stage when writing the compiler we were forced to completely remove any remaining dependencies upon Prolog.

As an example, consider the Erlang expression:

```
f(X, Y) ->
     Z = 3 * X + Y + 4 * 6 - X,
     ...
```

In the interpreter this was translated to the Prolog:

```
     ...
     Z is 3 * X + Y + 4 * 6 - X,
     ...
```

So the *meaning* of an arithmetic expression in Erlang was defined to be the *meaning* of whatever the corresponding expression was in Prolog.

When we wrote a parser for Erlang we had to define a grammar for arithmetic expressions, defining the grammar forced us to think about if we wanted to follow the Prolog grammar for arithmetic expressions or use some other grammar.

Another accidental consequence of the interpreter being written in Prolog was that in the original version of Erlang all variables in Erlang were represented as Prolog logical variables.

Eventually we decided that logical variables as in Prolog had certain undesirable properties in Erlang. This was because sending a message containing a logical variable between two processes allows the processes to communicate by an *invisible* mechanism (binding the variable) and not by message passing - we wanted all acts of interprocess communication to be explicitly achieved through message passing.

6 Defining documents

Erlang first appeared as a simple Prolog meta interpreter. At any particular point in time the interpreter *was* the language definition. The language itself had no defining documents and was defined totally in terms of its interpreter.

Eventually we started writing manuals, reference guides etc. which described the language. Since the language was undergoing rapid change the interpreter and the manuals were often out of phase. In the early stages of development if the interpreter and the manual differed, then the interpreter was *correct* and the manual *wrong*, this allowed us to answer any question about the language by *try it and see*.

As the language matured we came to a point where this approach had outlived its usefulness - we then reversed our policy - if the language and manual differed then it was the manual which was *correct* and the implementation which was *wrong*. At this stage we took a lot of care to ensure that the manual was correct - this is *very* difficult, while the Prolog system provided a very careful *proof reading* of the interpreter no such help could be found to help get the manual right! - (this is exactly the opposite of what is supposed to be conventional practice, one is supposed to write the defining documents first and then make the implementation - our way is much easier - it is easier to *describe* something which exists than to *imagine* the properties of something which does not yet exist!).

The shift in emphasis from the interpreter to the manual as the defining document reflected the achievement of a new level of maturity in the language.

7 Performance

The decision to concentrate on performance represented a milestone in the development. We had now entered a period where we were only interested in the performance of the language, we could leave the semantics unchanged while concentrating on implementation.

Our first idea was to cross compile Erlang to a concurrent logic programming language, this seemed natural since all our effort has been directed towards adding a notion of concurrency to Prolog and in a concurrency logic programming language concurrency is free!

We then made implementations loosely based on the WAM [WA83] and its descendants.

7.1 Cross compilation to concurrent logic programming languages

Our first attempts were to write cross compilers (in Prolog) from Erlang to Parlog [CL84] and STRAND [FO89] - of these the resulting STRAND based implementation was an order of magnitude faster than the Prolog interpreter - but still not fast enough to meet our requirements.

The cross compilation process which is described in detail in chapter 13 of [FO89] turned all Erlang functions of Arity N into STRAND goals of Arity N + 8. The resulting programs did not, however, achieve the kind of performance which we required.

We encountered certain problems in compilation to languages like STRAND. Our models of concurrency were different - Erlang provides concurrency at the process level, STRAND has much finer grain concurrency, our strategy for error handling and was different, these and other problems forced us to think in terms of our own implementation.

7.2 Compiler to a WAM like machine

The next stage of development was concerned with building a high performance Erlang machine.

The STRAND and Parlog implementations neatly side stepped several nasty implementation issues - bounded time garbage collection - real-time scheduling etc. The first compiled version forced us to think about such issues.

Inspiration for the compiled version of Erlang came from the WAM - since Erlang had grow out of Prolog it seemed natural that the techniques used to make fast Prolog Systems should also be applicable to Erlang. Our first major difficulty here was in understanding the WAM since at the time there were no easily understandable descriptions of the WAM (Now things are better, see [HA91]). The turning point came after having read Chapter 11 of [MA88] whose lucid description of a WAM like machine gave us enough understanding to make our own machine.

A compiler from Erlang to our own machine (JAM) and an emulator for the JAM machine were written. Initially both the compiler and the emulator were written in Prolog. The resulting machine was blindingly slow! (5 Erps (Erlang Reductions Per Second), 0.5 Erps when tracing the instructions) - but served to debug the instruction set and work out the structure of the call frames - the primary advantage was that we could use Prolog's underlying memory management to write the emulator.

Having worked out the instruction set and the architecture of the machine the compiler was re-written in Erlang itself and the emulator in 'C'. This version after three total re-writes now runs 70 times faster than the original prolog interpreter.

A more detailed description of the implementation of the language can be found in [AR91] where details of the JAM, the instruction registers used in the virtual machine and the implementation of concurrency can be found.

8 Current Status

Erlang is now a well established language for writing real-time software. Three complete implementations exist, one in Prolog, the other two are stand-alone *WAM - like* implementations.

Art the time of writing Erlang is in use at some 30 sites and is spreading at the rate of about 1-2 sites/month - it has been used for a number of prototypes and a small number of internal projects.

The language is freely available for non commercial use (information requests to `erlang@erix.ericsson.se`)

Acknowledgements

The development of the Erlang language and operating system is part of the on-going research program of the Ellemtel Computer Science Laboratory.

We would like to thank the head of the CSL Bjarne Däcker, for his help and support throughout the project - we would also like to thank all our users who have made developing the language so much fun, who have suggested valuable improvements to the language and who have stoically accepted all our non backwards compatible changes to the language.

References

[AR86] Armstrong, J. L., Elshiewy, N. A. and Virding, S. R. *The Phoning Philosopher's Problem or Logic Programming for Telecommunications Applications.* Third IEEE Symposium on Logic Programming, September 23-26, 1986, Salt Lake City, Utah.

[AR89] Armstrong, J.L., Williams, M.C. Using Prolog for Rapid *Prototyping of Telecommunications Systems.* 7th Int Conf. on Software Engineering for Telecommunication Switching Systems, Bournemouth 3 - 7 July 1989.

[AR90] Armstrong, J. L. and Virding, S. R. *Erlang an Experimental Telephony Programming Language.* XIII International Switching Symposium - Stockholm, Sweden. May 27 - June 1, 1990.

[AR91] Armstrong, J. L., Däcker B., Virding, S. R. and Williams, M.C. *Implementing a functional language for highly parallel real time applications.* 8th Int Conf. on Software Engineering for Telecommunication Switching Systems, Florence 30 March - 1 April 1992.

[BR86] - Brooks, F, B. Jnr. *No Silver Bullet - Essence and accidents of software engineering.* Information Processing 86. H.J. Kugler (ed.) Elsevier Science Publishers B.V. (North Holland), 1986.

[CL84] Clark, K.L. and Gregory. *PARLOG: Parallel programming in logic.* Research Report DOC 84/4, Department of Computing, Imperial College. London. April 84 (Revised June 1985).

[DÄ86] Däcker B., Elshiewy N., Hedeland P., Welin C-W. and Williams M. C. *Experiments with Programming Languages and Techniques for Telecommunication Applications.* Proc. 6th Int Conf. on Software Engineering for Telecommunication Switching Systems, Eindhoven, The Netherlands, 14-18 April 1986.

[FO89] *STRAND: New Concepts in Parallel Processing.* Ian Foster and Steven Taylor. Prentice Hall, 1989.

[HA91] Hassan A. K. *Warren's Abstract Machine: A tutorial Reconstruction.* MIT Press, Logic Programming Series. Cambridge, MA, 1991.

[MA88] Maier, David and Warren, D.S. *Computing With Logic* - The Benjamin/Cummings Publishing Company, Inv. 1988.

[MD82] Reprint of Ericsson Review articles on MD 110 from nos 1 and 2 1982

[PA86] Parnas, D.L., Clements, P.C. *A rational design process: How and Why to Fake It.* IEEE Transactions on Software Engineering VOL, SE- 12, No. 2, February 1986.

[PE91] *A PBX software prototype using a real-time declarative language* Persson, Mats, Ödling, K., Ericsson (in preparation)

[WA83] Warren, David, H.D. *An abstract Prolog instruction set.* Technical note 309, SRI International, Menlo PPark, CA, October 1983.