An introduction to Core Erlang

Richard Carlsson
Computing Science Department
Uppsala University
Box 311, 751 05 Uppsala, Sweden
richardc@csd.uu.se

ABSTRACT

CORE ERLANG is a new, official core language for the concurrent functional language ERLANG. CORE ERLANG is functional, strict, dynamically typed, pattern matching, higher order and concurrent. It has been developed in cooperation between the High-Performance Erlang project at Uppsala University and the Erlang/Open Telecom Platform development team at Ericsson, and is a central component in the new release of the Erlang/OTP compiler. This paper describes the core language, its relation to ERLANG, and the history of its development. We show examples of translation from ERLANG to CORE ERLANG, illustrating how analysis and transformation of ERLANG programs is greatly simplified by being performed at the core language level.

1. INTRODUCTION

1.1 The Erlang language

ERLANG [1, 3] is a concurrent functional language developed by Ericsson, which is today being used in several large real-world applications, with a code base of millions of lines of code. The main design goals for ERLANG have been increased productivity, error avoidance, concurrency and robustness; for these reasons, it is largely side effect free, the main source of side effects being the sending of messages.

ERLANG processes are very lightweight and are implemented at the abstract machine level, rather than as operating system threads, making it possible in practice to run thousands of processes on a single machine. Communication between processes is handled by asynchronous message passing, and is transparent over network connections. Attempting to receive a message from the process mailbox will suspend the process until a matching message has arrived, or some chosen timeout limit is exceeded.

The code executed by a process is a functional program; when the program terminates, so does the process. The value computed by such a program is always discarded upon termination; thus, unless the program sends a message to another process (or otherwise causes a side effect), its execution will not be visible to the rest of the system. Many ERLANG programs are therefore written to run in a recursive loop, acting on incoming messages by performing some computation and sending off new messages, in a server-like manner. Since the number of iterations is unbounded, ERLANG implementations must perform tail call optimisation, allowing a process to run forever in constant space.

The ERLANG language is higher-order, strict, and dynamically typed. The primary basic types are atoms (names), arbitrarily-sized integers, floating-point numbers, the empty list (written `[]`), and process identifiers. The only data constructors are n-tuples, written `{`t_1`, ..., `t_n`}` and cons cells `[`t_1 `|` t_2`]`. Strings are represented by lists of integers (character codes). A very common idiom in ERLANG is to place an atom in the first element of a tuple, as a tag indicating its contents: for example, the nodes of a syntax tree could be represented by `{constant, ...}`, `{lambda, ...}`, `{apply, ...}`, etc.; a process could in a certain state be interested only in messages on the form `{reply, Sender, Value}`, and so on.

ERLANG has for historical reasons picked up a few syntactical particularities from Prolog, the most prominent being that identifiers are variables if they begin with an uppercase letter, and are otherwise atoms. Atoms that do not begin with a lowercase letter, or contain strange characters, must be written within single-quotes, like `'Atom'` or `'$trange'`.

An ERLANG program consists of a set of modules, where each module defines a number of functions. A module is uniquely identified by its name as an atom. Within a module, each function is identified by its name (also an atom) and its arity. It is thus possible – and often done – to have several function definitions with the same name but with different arities. For each module, only functions that are explicitly declared as exported will be accessible from other modules.

The following is an example of a simple ERLANG module:

```
-module(list).
-export([length/1, reverse/1]).

length([X | Xs]) -> 1 + length(Xs);
length([]) -> 0.
```

```
reverse(Xs) -> reverse(Xs, []).

reverse([X | Xs], Ys) ->
   reverse(Xs, [X | Ys]);
reverse([], Ys) -> Ys.
```

(Note that the function whose name is `reverse/2` is not exported.)

Calls may be qualified with the name of the module, using the syntax `Module:Function(...)`. An external call to the module above could look like e. g. `list:length(L)`.

An important feature of ERLANG is *dynamic code replacement*: any module can be replaced at run-time with a new version. Whenever a module-qualified function call is made, the latest loaded version of the target module is used; this applies even if the target is the same as the caller. Thus, if the recursive loop of a server process always does a *qualified* tail call on each iteration, it will start executing the new code as soon as it loops back to handle the next request. ERLANG implementations must allow for at least two versions of a module to be stored in the system simultaneously: the current and the previous. If later the same module is replaced again, any process that is still running old code may be automatically found and killed, so that that code can be safely purged from the system.

1.2 Core Erlang history

Being a functional language semantically not far removed from Scheme [10], ERLANG should be well suited for the application of many interesting analyses and performance improving transformations, from the well-known to the experimental. However, this has not yet been very successfully done. There are several reasons for this, all following from the fact that the development of ERLANG was always pragmatically oriented, rather than theoretically (or even very systematically).[1]

The first reason is that as ERLANG has evolved, its syntax has become rather complex. It is still conceptually a small language, which makes it easy to learn, but the same function can often be expressed syntactically in many different ways. Thus, a program operating on source code must handle so many cases as to become impractical in general.

The second reason is that the original abstract machine used for ERLANG, the JAM [2] stack machine, had an instruction set which was similar enough to the actual source-level language that no intermediate representation was ever used in the compiler. When the new abstract machine, the BEAM [7] (a WAM-like register machine), was introduced, the source language was already fixed, and even though intermediate representations became necessary for the translation, these went as straight as possible to the abstract machine level, in order to get correct, working code with the least amount of trouble. However, both JAM and BEAM code are imperative in nature, and most optimisations from the functional programming world such as algebraic transformations, advanced inlining techniques, and specialisation, are not suitable or even possible to apply at that level. Standard local optimisations are of course done on the BEAM code, but although important, these typically have too limited scope for any drastic improvements.

[1]The positive side of that coin is however that the language is in fact very useful for writing heavy-duty real-world programs.

The third reason is that all attempts to simply reduce the language to a proper subset, ended up in convoluted discussions about whether or not expression A really could be rewritten as expression B, since except in a few cases, neither was originally defined in terms of the other. Then, even when semantic equivalence could be determined, it was generally the case that the rewritten, expanded expression would result in much *less* efficient code when passed through the compiler, because that did in fact need to identify each source code construct directly in order to generate reasonably good output.

In 1998, discussions were started between the High-Performance Erlang (HiPE) group at Uppsala University Computing Science Department and the OTP development team at Ericsson about a standard core language for ERLANG. Several different prototype representations were quickly put together, but there was no agreement on the details of the language.

In 1999, work began on creating a specification, initially by attempting to unify the existing suggestions, then by refinement and closer study of the many fine points of the semantics of ERLANG. Lots of changes were necessary before all involved were able to agree on a common core language, and it was not until late 2000 that version 1.0 of the CORE ERLANG specification [4] was finally available as a technical report.

As of release 8 of the OTP/Open Source ERLANG distribution, scheduled for October 2001, the compiler now uses CORE ERLANG as an intermediate representation, is able to read and write CORE ERLANG code in text form, and has hooks for adding user-defined transformations at the CORE ERLANG level.

1.3 Core Erlang design goals

The following were our main design goals for the CORE ERLANG language:

- It should be a strict, higher-order functional language, with a clear and simple semantics.
- It should be as regular as possible, to facilitate the development of code-walking tools.
- Translation from ERLANG programs to equivalent CORE ERLANG programs should be straightforward, and equally important, it should be straightforward to translate from CORE ERLANG to the intermediate code used in ERLANG implementations.
- There should be a well-defined textual representation of CORE ERLANG, with a simple, preferably unambiguous grammar, making it easy to construct tools for printing and reading programs. This representation should be possible to use for communication be-

tween tools using different internal representations of CORE ERLANG.

- The textual representation should be easy for humans to read and edit, in order to simplify debugging and testing.
- The language should be easy to work with, supporting most kinds of familiar optimisations at the functional programming level.

Of these, the last was by far the most difficult to achieve, trying to accomodate for the current or anticipated needs of everyone involved. In the end, we chose not to enforce any specific convention such as continuation-passing style or A-normal form; it is however an easy task to normalize any CORE ERLANG program according to preference.

2. THE CORE ERLANG LANGUAGE

2.1 Modules

As in ERLANG, all CORE ERLANG functions must reside within a module. A module declaration has the following form:

$$
\begin{array}{lll}
\mathit{module} & ::= & \texttt{module}\ \mathit{Atom}\ \texttt{[}\mathit{fname}_{i_1}\texttt{,} \\
 & & \qquad \ldots\texttt{,}\ \mathit{fname}_{i_k}\texttt{]} \\
 & & \texttt{attributes}\ \texttt{[}\mathit{Atom}_1\ \texttt{=}\ \mathit{const}_1\texttt{,} \\
 & & \qquad \ldots\texttt{,}\ \mathit{Atom}_m\ \texttt{=}\ \mathit{const}_m\ \texttt{]} \\
 & & \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \\
 & & \ldots \\
 & & \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{end}
\end{array}
$$

where each *fname* is a *function variable* with a special syntax:

$$\mathit{fname} \ ::= \ \mathit{Atom}\ /\ \mathit{Integer}$$

Function variables were not strictly necessary for the language, but make it easier to see the connection to the original program, and keep the export mechanism simple, since the external names are the same as the internal. The exported functions are listed after the module name, and must be a subset of those defined in the top level of the module.

Each module has a possibly empty set of *attributes*, where the keys, which must be unique, are atoms and the corresponding values are any constant terms. The meaning of module attributes is implementation-dependent; ERLANG uses them for things such as version information. Constant terms are:

$$
\begin{array}{lll}
\mathit{const} & ::= & \mathit{lit} \ \mid\ \texttt{[}\,\mathit{const}_1\ \texttt{|}\ \mathit{const}_2\,\texttt{]} \\
 & \mid & \texttt{\{}\,\mathit{const}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{const}_n\,\texttt{\}} \\
\mathit{lit} & ::= & \mathit{Integer} \ \mid\ \mathit{Float} \ \mid\ \mathit{Atom} \\
 & \mid & \mathit{Char} \ \mid\ \mathit{String} \ \mid\ \texttt{[]}
\end{array}
$$

In CORE ERLANG, all atoms must be written within single quotes, to avoid any confusion with keywords.

2.1.1 *Functions, lambdas and results of expressions*

Lambda abstractions are for apparent reasons known as *funs* in the ERLANG parlance. They were added to the language relatively late,[2] and play a much more central part in CORE ERLANG. The syntax is:

$$
\begin{array}{lll}
\mathit{fun} & ::= & \texttt{fun (}\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\texttt{) ->}\ \mathit{exprs} \\
\mathit{var} & ::= & \mathit{VariableName}
\end{array}
$$

Where no variable may occur twice in $var_1, \ldots, var_n$.

To simplify generation of efficient code, and to allow certain optimisations to be performed at the core language level, a CORE ERLANG expression always produces a *sequence* of values, which could be empty. Value-sequences are written within angular brackets (less than/greater than); if the length is exactly 1, the brackets may be left out. Each component expression $expr_1, \ldots, expr_n$ must produce exactly one value:

$$\mathit{exprs} \ ::= \ \mathit{expr} \ \mid\ \texttt{<}\,\mathit{expr}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{expr}_n\,\texttt{>}$$

The use of an expression must match the number of values it produces, which should be well-defined; if the expression is an argument to a constructor or function call, this number should always be 1. However, it is not possible in general to determine at compile time that a given CORE ERLANG program is correct in this respect.

2.2 Expressions

Basic expressions in CORE ERLANG are variables (ordinary variables and function names), atomic literals, lambda expressions ("funs"), cons cell constructors, and n-tuple constructors; all obviously producing exactly one value. In addition there are `let`-expressions, which bind n variables simultaneously to the values produced by the right-hand side, and the similar sequencing expressions `do`, which bind no variables. Furthermore, `letrec` expressions allow local (recursive) function definitions, which ERLANG itself does not have, but which are often useful in transformations.

$$
\begin{array}{lll}
\mathit{expr} & ::= & \mathit{var} \ \mid\ \mathit{fname} \ \mid\ \mathit{lit} \ \mid\ \mathit{fun} \\
 & \mid & \texttt{[}\,\mathit{exprs}_1\ \texttt{|}\ \mathit{exprs}_2\,\texttt{]} \\
 & \mid & \texttt{\{}\,\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\,\texttt{\}} \\
 & \mid & \texttt{let}\ \mathit{vars}\ \texttt{=}\ \mathit{exprs}_1\ \texttt{in}\ \mathit{exprs}_2 \\
 & \mid & \texttt{do}\ \mathit{exprs}_1\ \mathit{exprs}_2 \\
 & \mid & \texttt{letrec}\ \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \\
 & & \cdots \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{in}\ \mathit{exprs} \\
 & \mid & \texttt{apply}\ \mathit{exprs}_0\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{call}\ \mathit{exprs}_{n+1}\texttt{:}\mathit{exprs}_{n+2}\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{primop}\ \mathit{Atom}\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{try}\ \mathit{exprs}_1\ \texttt{catch (}\mathit{var}_1\texttt{,}\ \mathit{var}_2\texttt{) ->}\ \mathit{exprs}_2 \\
 & \mid & \texttt{case}\ \mathit{exprs}\ \texttt{of}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \texttt{end} \\
 & \mid & \texttt{receive}\ \mathit{clause}_1\ \cdots\ \mathit{clause}_n \\
 & & \quad \texttt{after}\ \mathit{exprs}_1\ \texttt{->}\ \mathit{exprs}_2 \\
\mathit{vars} & ::= & \mathit{var} \ \mid\ \texttt{<}\,\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\,\texttt{>}
\end{array}
$$

ERLANG has rather unusual scoping rules, making code traversal difficult, but in CORE ERLANG, scopes are nested as in ordinary lambda calculus.

[2] ERLANG was originally made a first-order language, because it was thought that lambda abstractions would make programs too complicated, and could generally be done without.

2.2.1 Function calls

Core Erlang function calls come in three flavours: *applications*, *remote calls*, and *primitive operations*. The first is simply the application of a functional value (such as a locally defined function) to a list of arguments.

$$\texttt{apply}\ exprs_0\texttt{(}exprs_1\texttt{,}\ \ldots\texttt{,}\ exprs_n\texttt{)}$$
$$\texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}\texttt{(}exprs_1\texttt{,}\ \ldots\texttt{,}\ exprs_n\texttt{)}$$
$$\texttt{primop}\ Atom\texttt{(}exprs_1\texttt{,}\ \ldots\texttt{,}\ exprs_n\texttt{)}$$

A *remote call* (module-qualified call) has two subexpressions apart from the argument list: these must evaluate to atoms, giving the module and function name, respectively, of the "remote" function to be called. Usually, both are statically known at compile time, but they may be dynamically computed. For a remote call, the latest loaded version of the named module is always used; this allows migration from old code to new, as previously described.

Primitive operations are included to make a watertight distiction between operations that are internal to the compiler, and normal functions that are exported by some module. The name of the called operation must be a constant. E. g., it could be safe for the compiler in certain situations to introduce an operation at the Core Erlang level that destructively updates a part of a data structure, but programmers must be prevented from accessing such operations directly from Erlang, since unrestricted use could violate the language semantics.

2.2.2 Error handling

Core Erlang does not specify how exceptions are generated; only how they may be caught. An exception has two components: the *tag* and the *value*. These may be any values, but the tag is expected to signify the class of the exception; in Erlang, possible tags are currently the atoms `'EXIT'` and `'THROW'`.

$$\texttt{try}\ exprs_1\ \texttt{catch}\ \texttt{(}var_1\texttt{,}\ var_2\texttt{)}\ \texttt{->}\ exprs_2$$

In a `try` expression, if evaluation of the attempted expression should fail, raising an exception, then this will be caught and instead the alternative body of the try-expression will be evaluated with var_1 bound to the tag and var_2 to the value of the exception.

2.2.3 Pattern matching

Like Erlang, the core language uses pattern matching for decomposition and branching, but only in two constructs (compared to six in Erlang), of which the first is the `case` switch. The selector expression produces a fixed number of values, and each clause, on the form *patterns* `when` *guard* `->` *body*, must contain the corresponding number of patterns. Variables may not be repeated in the patterns of a clause, and are always binding occurrences, whose scope is the *guard* and *body* expressions.

$$\texttt{case}\ exprs\ \texttt{of}\ clause_1 \cdots clause_n\ \texttt{end}$$

$$
\begin{array}{rcl}
clause & ::= & pats\ \texttt{when}\ exprs_1\ \texttt{->}\ exprs_2 \\
pats & ::= & pat \quad | \quad \texttt{<}pat_1\texttt{,}\ \ldots\texttt{,}\ pat_n\texttt{>} \\
pat & ::= & var \quad | \quad lit \quad | \quad \texttt{[}pat_1\ \texttt{|}\ pat_2\texttt{]} \\
& | & \texttt{\{}pat_1\texttt{,}\ \ldots\texttt{,}\ pat_n\texttt{\}} \\
& | & var\ \texttt{=}\ pat
\end{array}
$$

Patterns include the form *var* = *pat*, which is analogous to `as` patterns in ML [9]. Clauses are tried in order, until the first with a matching pattern is found whose guard evaluates to `true`. If no clause should match, the behaviour is undefined.

A clause guard is a limited form of expression – it must produce a single boolean value (either of atoms `true` and `false`), and must not have side effects. Currently, only variables, atomic literals, constructors, `let`-, `do`- and `try`-expressions, and `primop` and `call` calls to statically named functions are allowed in guards. Called operations and functions must be guaranteed to exist and be free from side effects: typical examples are type tests, comparison operators and selectors.

Error handling is also restricted in guards. A `try`-expression in a guard must have the following form:

$$\texttt{try}\ exprs_1\ \texttt{catch}\ \texttt{(}var_1\texttt{,}\ var_2\texttt{)}\ \texttt{->}\ \texttt{'false'}$$

In other words, if the evaluation of the tried expression should fail, the exception is caught and the value `false` is substituted for the result. This is used for supporting the quite unorthodox way Erlang treats exceptions in guard expressions. It is possible that in future versions of Core Erlang, fully general `try` expressions will be allowed in guards.

2.2.4 Receive expressions

Last, we describe the rather intricate asynchronous `receive` expression. Its syntax is very similar to a `case` expression:

$$\texttt{receive}\ clause_1 \cdots clause_n$$
$$\texttt{after}\ exprs_1\ \texttt{->}\ exprs_2$$

but it has no explicit selector, and adds a final "clause" for timeouts.

A `receive` expression implicitly contains a loop, which is the main reason why it cannot be reduced to simpler components in a useful way. The executing process will traverse its mailbox queue, always from the beginning, and either select the first message that matches one of the clauses, removing the message from the mailbox and evaluating the clause, or otherwise suspend until a matching message arrives or the timeout limit is exceeded. Time is given as an integer number of milliseconds, zero meaning immediate timeout if no message matches. The atom `infinity` is used to signal that the timeout should never happen.

It is obvious that attempting to break down this construct will always expose the different possible states during traversal and matching, which could jeopardise the integrity of the mailbox. What is perhaps even more important, decomposition would make it much more difficult to use the core language for process communication analysis. Therefore, the `receive` is carried over practically as it is from Erlang to Core Erlang.

2.3 Syntax summary

Figure 1 shows a summary of the language syntax.

3. TRANSLATION EXAMPLES

As mentioned, the scoping rules of ERLANG are rather unusual; each expression has an *input environment* and an *output environment*. It is e. g. possible to write:

```
f(X) ->
   case X of
      {foo, A} -> B = g(A);
      {bar, A} -> B = h(A)
   end,
   {A, B}.
```

where the values of A and B in the resulting tuple depend on the selected branch. Maintaining input and output environments while traversing ERLANG code is cumbersome and error prone. Translating the above into CORE ERLANG might produce the following code:

```
'f'/1 = fun (X) ->
           let <X1, X2> =
              case X of
                 {foo, A} when 'true' ->
                    let B = apply 'g'/1(A)
                    in <A, B>
                 {bar, A} when 'true' ->
                    let B = apply 'h'/1(A)
                    in <A, B>
              end,
           in {X1, X2}
```

which is much easier to traverse, manipulate and reason about, following ordinary lambda calculus scoping rules. This example also illustrates one of the main reasons why CORE ERLANG expressions produce multiple values.

The ERLANG module shown in Section 1.1 can be automatically translated to the following CORE ERLANG code, using the fact that in ERLANG, infix operators such as + belong to the standard library module named `erlang`:

```
module 'list' ['length'/1, 'reverse'/1]
attributes []
'length'/1 =
   fun (X1) ->
      case X1 of
         [X | Xs] when 'true' ->
            let X2 = apply 'length'/1(Xs)
            in call 'erlang':'+'(1, X2)
         [] when 'true' -> 0
         X2 when 'true' ->
            call 'erlang':'exit'('no_clause')
      end
'reverse'/1 =
   fun (X1) -> apply 'reverse'/2(X1, [])
'reverse'/2 =
   fun (X1, X2) ->
      case <X1, X2> of
         <[X | Xs], Ys> when 'true' ->
            let X3 = [X | Ys]
            in apply 'reverse'/2(Xs, X3)
         <[], Ys> when 'true' -> Ys
         <X3, X4> when 'true' ->
            call 'erlang':'exit'('no_clause')
      end
   end
```

We see that all pattern matching is moved to `case` expressions, and that some of the run-time error checking done by ERLANG is made explicit by adding default clauses; in the `reverse/1` function, this has been removed by eliminating unreachable code. The evaluation order of ERLANG expressions is enforced by introducing `let` bindings during translation. (In the above example this was not strictly necessary.)

Translating operators like + to qualified calls to the module `erlang` is done in order to have a normalised representation of standard operators, making it possible to write program transformations that are not implementation dependent. Instead of presenting a list of official CORE ERLANG "primops", we use the fact that these functions already have definitions in the ERLANG standard library. There should be no runtime penalty induced by this, however, because the compiler is allowed to recognize calls to such built-in functions (BIFs) and generate more efficient code for these, using the assumption that they will not be redefined. In other words, a later transformation stage could rewrite such calls to `primop` applications, making the program representation dependent of the particular compiler implementation, as a prelude to low-level code generation.

4. RELATED WORK

Dam and Fredlund [5] and Huch [8] have used core fragments or subsets of ERLANG for purposes of program analysis (also and independently using the name "Core Erlang"). These subsets are however incomplete with respect to representing all possible ERLANG programs, and are not entirely suitable as an intermediate code representation in the compilation process.

Feeley and Larose [6] (the ETOS project) compile ERLANG by translation to Scheme[10], but the generated Scheme code is too tightly coupled to their implementation to be useful as a generic intermediate format for ERLANG. Furthermore, it does not seem feasible to perform e. g. program verification or process communication analysis using the output from their translation.

5. CONCLUDING REMARKS

We have described the CORE ERLANG language, its relation to ERLANG, and the history of its development. CORE ERLANG is today being used in the Erlang/OTP compiler, which is the standard compiler for the ERLANG language.[3] Although the compiler back end had to be modified to handle some of the constructs, this was not a major problem, and in several cases cleaned up the implementation. Fur-

[3]The Erlang/OTP compiler, runtime system and libraries are available as Open Source from `http://www.erlang.org`.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{Atom}\ \texttt{[}\mathit{fname}_{i_1}\texttt{,}\ \ldots\texttt{,}\ \mathit{fname}_{i_k}\texttt{]} \\
 & & \texttt{attributes}\ \texttt{[}\mathit{Atom}_1\ \texttt{=}\ \mathit{const}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{Atom}_m\ \texttt{=}\ \mathit{const}_m\ \texttt{]} \\
 & & \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \cdots \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{end} \\
\mathit{fname} & ::= & \mathit{Atom}\ \texttt{/}\ \mathit{Integer} \\
\mathit{const} & ::= & \mathit{lit} \mid \texttt{[}\mathit{const}_1\ \texttt{|}\ \mathit{const}_2\texttt{]} \mid \texttt{\{}\mathit{const}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{const}_n\texttt{\}} \\
\mathit{lit} & ::= & \mathit{Integer} \mid \mathit{Float} \mid \mathit{Atom} \\
 & \mid & \mathit{Char} \mid \mathit{String} \mid \texttt{[]} \\
\mathit{fun} & ::= & \texttt{fun}\ \texttt{(}\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\texttt{)}\ \texttt{->}\ \mathit{exprs} \\
\mathit{var} & ::= & \mathit{VariableName} \\
\mathit{exprs} & ::= & \mathit{expr} \mid \texttt{<}\mathit{expr}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{expr}_n\texttt{>} \\
\mathit{expr} & ::= & \mathit{var} \mid \mathit{fname} \mid \mathit{lit} \mid \mathit{fun} \\
 & \mid & \texttt{[}\mathit{exprs}_1\ \texttt{|}\ \mathit{exprs}_2\texttt{]} \mid \texttt{\{}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{\}} \\
 & \mid & \texttt{let}\ \mathit{vars}\ \texttt{=}\ \mathit{exprs}_1\ \texttt{in}\ \mathit{exprs}_2 \\
 & \mid & \texttt{do}\ \mathit{exprs}_1\ \mathit{exprs}_2 \\
 & \mid & \texttt{letrec}\ \mathit{fname}_1\ \texttt{=}\ \mathit{fun}_1 \cdots \mathit{fname}_n\ \texttt{=}\ \mathit{fun}_n\ \texttt{in}\ \mathit{exprs} \\
 & \mid & \texttt{apply}\ \mathit{exprs}_0\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{call}\ \mathit{exprs}_{n+1}\texttt{:}\mathit{exprs}_{n+2}\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{primop}\ \mathit{Atom}\texttt{(}\mathit{exprs}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{exprs}_n\texttt{)} \\
 & \mid & \texttt{try}\ \mathit{exprs}_1\ \texttt{catch}\ \texttt{(}\mathit{var}_1\texttt{,}\ \mathit{var}_2\texttt{)}\ \texttt{->}\ \mathit{exprs}_2 \\
 & \mid & \texttt{case}\ \mathit{exprs}\ \texttt{of}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \texttt{end} \\
 & \mid & \texttt{receive}\ \mathit{clause}_1 \cdots \mathit{clause}_n\ \texttt{after}\ \mathit{exprs}_1\ \texttt{->}\ \mathit{exprs}_2 \\
\mathit{vars} & ::= & \mathit{var} \mid \texttt{<}\mathit{var}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{var}_n\texttt{>} \\
\mathit{clause} & ::= & \mathit{pats}\ \texttt{when}\ \mathit{exprs}_1\ \texttt{->}\ \mathit{exprs}_2 \\
\mathit{pats} & ::= & \mathit{pat} \mid \texttt{<}\mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\texttt{>} \\
\mathit{pat} & ::= & \mathit{var} \mid \mathit{lit} \mid \texttt{[}\mathit{pat}_1\ \texttt{|}\ \mathit{pat}_2\texttt{]} \mid \texttt{\{}\mathit{pat}_1\texttt{,}\ \ldots\texttt{,}\ \mathit{pat}_n\texttt{\}} \\
 & \mid & \mathit{var}\ \texttt{=}\ \mathit{pat}
\end{array}
$$

Figure 1: Syntax summary

thermore, the core language makes it possible to continue extending ERLANG in a more systematic way than before.

Our experience (e.g., from the implementation of a recent, advanced inlining algorithm, likely to be incorporated in the standard compiler) is that analysis and transformation of ERLANG programs is greatly simplified by being performed at the core language level, which has only a few, simple and well-defined constructs, and uses ordinary scoping rules.

6. ACKNOWLEDGMENTS

I would like to thank Kostis Sagonas for his advice and comments on versions of this paper.

7. REFERENCES

[1] Armstrong, Virding, Wikström, and Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice Hall, 1996.

[2] J. Armstrong, B. Däcker, R. Virding, and M. Williams. Implementing a functional language for highly parallel real-time applications. In *Software Engineering for Telecommunication Switching Systems*, Florence, March 1992.

[3] J. Barklund and R. Virding. Erlang 4.7.3 reference manual. Draft version 0.7, June 1999.

[4] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000-030, Department of Information Technology, Uppsala University, November 2000.

[5] M. Dam and L. Fredlund. On the verification of open distributed systems. Technical Report R97-01, Swedish Institute of Computer Science, 1997.

[6] M. Feeley and M. Larose. Compiling Erlang to Scheme. In *LNCS 1490*, pages 261–272. Springer-Verlag, 1998.

[7] B. Hausman. Turbo Erlang: Approaching the speed of C. In E. Tick, editor, *Implementations of Logic Programming Systems.* Kluwer Academic Publishers, 1994.

[8] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 261–272. ACM Press, 1999.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[10] G. J. Sussman and G. L. Steele Jr. SCHEME, an interpreter for extended lambda calculus. AI Memo 349, Mass. Inst. of Technology, Artificial Intelligence Laboratory, Cambridge, Mass., December 1975.