Ideas for a new Erlang

Sven-Olof Nyström

February 17, 2009

Abstract

This paper presents some thoughts and ideas on the future development of Erlang. Among the topics are: an alternative to Erlang's selective receive, a simple language mechanism to allow function in-lining across module boundaries, a new mechanism for introducing local variables with a more cleanly defined semantics, and a mini-language to allow the efficient implementation of low-level algorithms.

1 Introduction

Erlang is an interesting example of a concurrent, high-level programming language. It is very instructive to consider how easy it is to write complex concurrent applications in Erlang compared to concurrent programming languages that are based on shared memory, locks, and synchronization. On the other hand, it's far from clear that all features of Erlang are necessarily the ideal ones. Saying "this is a satisfactory solution and we see no reason to try to improve it further" might make sense to some, but for those of us who don't think that the history of programming languages ended with the current version of Erlang, I have assembled a list of possible improvements of Erlang. My intention is that these changes would preserve the current strengths of Erlang, would not affect the basic organization of Erlang applications, but make the language simpler and more powerful.

Erlang might at first seem simple, but as anyone who has studied the language in detail knows, it is actually quite complicated. Consider, for example, selective receive, guards which superficially resembles ordinary expressions but are more restricted and have different execution rules, the rules for how new bindings may be introduced in a context, or the preprocessor and the need for include files, and how records are defined.

Why would one want to simplify Erlang? Complexity obviously has an immediate cost in learning and using the language. In the long run, just like a simple program is often easier to modify and extend than an unnecessarily complicated one, simplifications may allow the language to be extended, making it more powerful. Conversely, new ways of doing things might allow us to get rid of old machinery, thus again simplifying the language.

My proposals are largely independent; most of the time, you can have one without the other, but I think the best result would be obtained if they were combined.

An earlier version of this paper was presented (and discussed) on the Erlang Questions mailing list, June and July 2008.

- In Section 2, a few notes on implementation and backward compatibility.
- Section 3 looks at Erlang's selective receive and proposes an alternative.
- In Section 4, a simple language mechanism to allow function in-lining across module boundaries. This would also allow a better record concept.
- Section 5 proposes a binding mechanism very similar to Erlang's current one, but with a more cleanly defined semantics.
- Section 6 describes a mini-language to allow the efficient implementation of low-level algorithms.
- Finally, in Section 7 I discuss the macro feature of Common Lisp.

2 Implementation and backward compatibility

New versions of a programming language are normally backward compatible at the source code level, so that old code can be run on the new system. If we want to liberate ourselves from the syntax of old Erlang, we need to consider other options.

An alternative strategy would be to base implementation on existing Erlang run-time system, and require that

- processes executing old code can spawn processes executing new code, and vice versa, and
- processes executing old and new code can communicate by sending messages.

These requirements would of course imply that Erlang's data structures remain largely unchanged.

Obviously, one would not want to write a new run-time system from scratch. Allowing old and new Erlang processes to interact makes it possible to incrementally rewrite an Erlang application to take advantage of the new language. This strategy would of course impose some limitations on the new language, but as we shall see, there are still many new options to explore.

3 Channels

The most interesting features of Erlang are of course the primitives for concurrency; process creation and process communication. Recently, there have been many attempts to implement Erlang-like process communication in other languages (see for example [5, 4]). This makes it even more worthwhile to take a second look at Erlang's concurrency model to determine which features are the important ones.

3.1 Selective receive

An Erlang process receives messages through the `receive` expression. The incoming messages are stored in a sequential buffer (normally referred to as a *mailbox*) which is then searched sequentially.

For example, an expression

```
receive
    {hello,Y} when Y < 100  -> ...
    {bye, Z} -> ...
end
```

will accept messages which are either of the form {`hello,...`} (where the second element of the tuple is less than 100) or {`bye, ...`}. If the message {`hello, 42`} is sent to the process the variable `Y` will be bound to the integer `42` and the corresponding branch will be executed. If no matching message is found, the process will suspend until a message arrives.

The tricky part is that receive may need to look at many messages before it finds one that matches. It is however considered bad style to leave many messages in the mailbox, as the cost of searching the mailbox increases with the number of messages stored in it. One often repeated recommendation is to give receive expressions an extra clause that will match any message (which is then removed without processing) [1, Chapter 5].[1]

It is strange that Erlang has a feature that is on one hand so general and powerful, but yet programmers are not supposed take advantage of it! Perhaps selective receive could be replaced with something simpler?

Other programming languages based on asynchronous message passing [3, 6] have chosen a much simpler mechanism where the first message in the queue is always selected.

There are a few obvious uses for selective receive: The incoming messages that match a particular pattern form a "virtual channel". When a process sends a request to another process and expects a response, it has to distinguish the response from other incoming messages. The standard solution is to use selective receive. Selective receive can also be used in other situations when wants to distinguish one class of messages from another, for example if one wants to recognize high-priority messages.

Consider as an example of the first use a rather typical Erlang program, loosely following [1, Chapter 5]. A `counter` process maintains the state of a counter. It can react to the messages `inc`, `value`, and `stop`.

The `inc` message causes the `counter` process to increment its internal counter by one, the `value` message is a request for the current value of the counter, and the `stop` message causes the process to halt.

The process is implemented by the function below.

```
counter(Val) ->
    receive
```

[1] I have been told at more than one occasion that an overfull mailbox may cause its process to crash. Perhaps this was true for early Erlang implementations, but when I tested I found no obvious limitations (other than those imposed on the whole system).

```
        inc ->
            counter(Val+1);
        {value,From} ->
            From ! {self(),Val},
            counter(Val);
        stop ->
            true
    end.
```

The code for another process using the counter:

```
...
Counter ! inc,
Counter ! inc, % increment the counter twice
Counter ! {self(), value}, % ask the counter about its
                           % current value
receive

     {Counter, Value} -> % match messages that are tuples
                         % containing the counter as first
                         % argument

true
end,
% the variable Value is now bound to the value of the counter
```

To implement the response of the `value` message in Erlang a simple programming idiom is used. The exchange of messages is described below, where the process A is the one asking for information and the process B is the responding process, e.g., the `counter` process in the example.

1. The process A sends the request to process B. In the request the process identifier of A is included.
2. The receiver (process B) recognizes the message, extracts the process identifier and computes the answer.
3. Process B sends the answer to to A. To make sure that A can recognize the answer, the process identifier of B is included in the answer (obtained by calling `self()`).
4. Process A recognizes the answer by comparing the included process identifier with B's process identifier.

Selective receive also allows the mailbox to be used as a sequential storage area. Value are added to the mailbox by sending them as messages to the process. Erlang's receive can then be used to scan the mailbox for values matching a particular pattern. There are a number of problems with this approach. Erlang's implementation of mailboxes implies a sequential search which will be too expensive when the number of messages are large. Another limitation is that there is no direct way to determine the number of messages in a mailbox. In my opinion, a better solution is to use an explicit data structure (for example, a list or a balanced binary search tree).

3.2 Suggestion: Allow channels as first class objects

Selective receive is a rather unattractive programming language construct with complex semantics. My proposal on how to manage without selective receive is simply to allow channels as first-class objects.

As before, let each process have a "standard" channel for requests from other processes, but allow processes to create additional channels.

Below is a tentative list of primitives (i.e., BIFs) that will allow the use of first-class channels (in particular, their names are very tentative).

1. `current()`

   return the standard channel of current process

2. `new_channel()`

   create a new channel

3. `ne_receive(Ch)`

   read message from channel `Ch` (suspends if message not present). In the example below, I assume that `ne_receive()` reads from the standard channel when called with no arguments.

4. `Ch ! M`

   send message `M` on channel `Ch`

Some functions could be given shorter names; my choice of names is intended to help avoid confusion between old and new Erlang. There should also be functions that allow a receive with timeout.

Using the new primitives, the "counter-example" could be implemented as follows:

```
counter(Val) ->
   case ne_receive() of
     inc ->
         counter(Val+1);
     {value,From} ->
         From ! Val,
          counter(Val);
        stop ->
            true
    end.
```

and the the code that acesses the counter

```
    ...
    Counter ! inc,
    Counter ! inc,
    Q = new_channel(), % create a new channel for this question
    Counter ! {value, Q},
    Value = ne_receive(Q),
```

3.3 Semantics of channels

Channels will be easier to understand and implement if we impose one rule: *only the creator of a channel may receive messages from it.*

My first reason for imposing this limitation was that it seemed intuitive: a channel represents a service provided by its creator, and thus only the creator would know how to perform the service and know what to do with arriving messages. However, a moment's reflection reveals an important practical advantage. The rule implies that a channel can be stored with the process that created it. If the process that created a channel dies, the channel can be safely removed as no other process can access its contents. If the creating process no longer refers to the channel, it can be removed by the garbage collector.

One very straight-forward way to implement channels is to use Erlang's mailboxes, without modification. Each message is represented as a tuple consisting of a channel id and contents. At the receiving end, selective receive picks out the first message sent on a virtual channel using pattern matching. We assume that the standard channel has number 0.

```
new() -> {channel, self(), make_ref()}.

current -> {channel, self(), 0}.

ne_send({channel, P, Id}, Message) -> P ! {Id, Message}.

ne_receive({channel, P, Id}) when P = self() ->
    receive
        {Id, Message} -> Message
    end.
```

This implementation will of course give unsatisfactory performance when a process creates many channels (also, channels will not be garbage collected). A better approach is to have the process allocate a separate memory area for each channel, but that requires modifications to the run-time system.

3.4 Receiving messages on many channels

An earlier version of this paper mentioned the need for a receive expression that could wait for input on more than one channel (i.e., a *multi-receive*). As Richard O'Keefe pointed out, the design of a multi-receive is not straightforward. Should it take a list of channels as argument? Should there also be a provision for timeout? When a multi-receive returns a message, should it also give information about on which channel the message arrived?

Beside these considerations, I have also started to question the need for multi-receive. Suppose a process needs to wait for input on two channels. If the two channels represent the same service, would it not be a better design to use one channel instead of two? Conversely, if the channels represent different services, would it not be better to use two processes?

3.5 Guards

The complex selective receive primitive of old Erlang has forced the Erlang designers to use a programming construct called *guards*, similar to expressions

but with its own evaluation rules and semantics. That function calls are not allowed in a guard is inconvenient as soon as one wants to test a property that is defined by a function.

For example, to test if 42 is an element of the list `L`, one must write a case expression

```
case list:member(42, L) of
    true -> ...;
    false -> ...
end
```

While the reasons for requiring guards in receive-expressions are quite easy to see (one wants to avoid code that interacts with the mailbox or has other side effects), it is harder to see why the same restriction is enforced in other programming constructs, for example if- and case-expressions.

It has sometimes been argued that limiting what can be put in a guard could potentially allow a compiler to make more optimizations, as code evaluating guard tests could be moved around more easily when they are known to contain no side-effects. Note that

1. no Erlang compiler takes advantage of such optimizations today,
2. the potential gains are small, and
3. if it was deemed worthwhile, a compiler could check for simple guards and optimize such code, even if the language allowed arbitrary guards.

3.6 Expressiveness of selective receive vs. channels

We have seen how a set of channels can be simulated by selective receive. In the opposite direction, it is of course always possible to represent a mailbox by an explicit data structure (if the mailbox stores a large number of messages this could even lead to a more efficient solution).

The choice between selective receive is not a matter of efficiency or even of expressiveness. My argument for the solution with channels is that it uses primitives that are easier to understand and reason about, and (I expect) will lead to better concurrent Erlang programs.

4 Linked modules[1]

Erlang's record facility is rather unusual (at least in the world of functional programming languages). A record definition is a directive to the pre-processor, providing definitions on how to expand code that creates, updates, or pattern matches records. Since record definitions are often shared between modules, they are typically put in include files, so that they can be included in each file where they are used. Records are represented as tuples, so all operations on records are translated into code that operate on tuples.

The use of include files and a preprocessor has some undesirable consequences. For example. the compiler is unaware of record definitions and and a

[1] Richard O'Keefe tells me that he expressed similar ideas in a paper titled "Delenda est preprocessor".

compile-time analysis isn't possible (for finding errors or performing optimizations). There is also no easy way to detect at compile-time if different modules use conflicting record definitions.

To me, the obvious alternative is to expand a record definition into a set of function definitions, similar to `defstruct` in Common Lisp [11, Chapter 8]. For example, a definition

```
(defstruct person
  name
  age)
```

creates a type `person`, and generates a function for creating values of the type `person`, a predicate that recognizes values of the type `person` and functions that access and update the fields.

In a corresponding Erlang solution, we would also need a way to specfy the expansion of pattern matching of records into appropriate function calls. (Richard O'Keefe's abstract pattern proposal [9] could also be implemented in this way.)

One problem with this solution is that record operations would be much more expensive than they are today, as every record operation would entail the overhead of a function call. (This is probably why records are implemented using the pre-processor.) If a record is defined in the same module, the compiler could simply inline the function call, i.e., replace the call with the definition of the function. However, if a record definition is to be shared between many modules, all record operations would have to be implemented as calls between modules. Now, in Erlang such calls are even more expensive than ordinary calls, because of Erlang's hot code update. For the same reason, they can't be in-lined (however, see [7, 8]).

4.1 Proposal: linked modules

My proposal is simple: Introduce a declaration

```
-linked_module(m).
```

indicating that the code of the current module depends on the module `m`. The implication of a `linked_module` declaration should be that when a module A links to a module B, the compiler should be free to assume that the module B remains the same as long as long as module A stays loaded. This should allow interprocedural optimizations such as inlining.

4.2 Implementation

Now, there has to be some form of machinery to make sure that the promise is not broken when code is updated. One solution is to record the link information in the run-time system and require that, as in the example, module A is reloaded whenever B is reloaded. Another approach would be to compile two versions of module A, one that assumes that module B is the current one, and one that does not. If module B changes, the run-time changes A to the version that makes no assumptions on B.

At this time it is perhaps a good idea to consider what happens today when an include file is changed. I haven't found any description in the literature of

how this is handled in the development of large-scale telephony systems, but a moment's reflection leads to some educated guesses. Clearly, the run-time has no way of knowing which Erlang files depend on the include file. Besides handling the dependencies manually, this information could be encoded in make-files (or similar) which would then cause the relevant source files to be recompiled. It would then be up to the development team to make sure that the new version of the module is re-loaded.

4.3 Summary

With a `linked_module` declaration, function calls from module A to module B can be implemented as efficiently as function calls internal to a module. If one module defines a record type which is used by many others, and efficiency is a concern, we can avoid the function call overhead. The declaration would also be useful in other cases where we want to avoid the overhead of function calls between modules. One very striking case is the use of functions to define constants, for example (in Erlang's standard library):

```
pi() -> 3.1415926535897932.
```

With inlining of function calls, calling this function is as efficient as writing the constant directly.

A limitation of my proposal is that it would not make sense to include a link declaration whenever a module referred to another—in an application that did, a hot code upgrade would require reloading the whole program. Instead, it is up to the developers to determine the cases where one module relies heavily on definitions of another, and link those modules.

5 Better treatment of bindings

Erlang has chosen an unconventional model for introducing local bindings, for example, this code

```
X = 42,
Y = X+X,
Y*Y.
```

contains two bindings (not assignments). If we compare with the equivalent code in, say, the functional programming language SML

```
let val x = 42
    val y = x+x
in
    y*y
end
```

we find that the SML code is larger and requires four keywords where Erlang requires none. Other functional programming languages use similar constructs. In my opinion, the greatest advantage of Erlang's approach is when code that introduces bindings needs to be mixed with conditionals, say

```
case X of
   1 -> Y = 2, Z = [1,2,3];
   2 -> Y = 3, Z = [5,6,7]
end
...
```

or

```
X = ...
case X of ...
   1 -> Y = X*X, ...
   ...
end
```

The corresponding code with explicit let expressions becomes more involved and harder to read:

```
let {Y, Z} =
   case X of
     1 -> {2, [1,2,3]};
     2 -> {3, [5,6,7]}
   end
in
   ...
end

let X = ...
    in
       case X of ...
           1 ->
               let Y = X*X
               in
                 ...
               end
       ...
       end
end
```

The two solutions using explicit let are far less attractive than the ones using Erlang-style binding. The ability to export bindings from branches in a conditional is convenient, but the rules that govern exactly which code is allowed are complex. One irregularity is that bindings introduced in an anonymous function are not visible outside the function.

My proposal gives a binding mechanism that from a practical point of view is very similar to the the current one, but with a simple type system that specifies exactly which combinations of bindings. One guiding principle in my system is: *An expression may return a value or export bindings, but may not do both.*

In the following, I give a simple inductively defined subset of an Erlang-like language and give a set of inference rules that specify which expressions are legal.

Given a set of variables $v \in \text{Var}$, the set of expressions is defined inductively:

$$\begin{array}{rl} e ::= & v \\ | & v = e_1 \\ | & e_1, e_2 \\ | & e_0(e_1, \ldots, e_n) \\ | & \text{fun}(v_1, \ldots, v_n) \rightarrow e_0 \\ | & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \end{array}$$

A type is either

1. T – represents the type of an expression that returns a value
2. $d(S)$ – where $S \subseteq \text{Var}$; an expression exporting bindings to the variables in S.

When an expression e evaluated in an environment where the variables of S are defined has the type τ, write

$$S \models e : \tau$$

We can now define the inference rules.

$$\frac{v \in S}{S \models v : \text{T}} \tag{1}$$

$$\frac{S \models e_1 : \text{T}, v \notin S}{S \models v = e_1 : d(S \cup \{v\})} \tag{2}$$

$$\frac{S \models e_1 : T \qquad S \models e_2 : \tau}{S \models e_1, e_2 : \tau} \tag{3a}$$

$$\frac{S_1 \models e_1 : d(S_2) \qquad S_2 \models e_2 : \tau}{S_1 \models e_1, e_2 : \tau} \tag{3b}$$

$$\frac{S \models e_i : \text{T}, \text{forall } i \leq n}{S \models e_0(e_1, \ldots, e_n) : \text{T}} \tag{4}$$

$$\frac{S \cup \{v_1 \rightarrow T, \ldots v_n \rightarrow T\} \models e_0 : T}{S \models \text{fun}(v_1, \ldots, v_n) \rightarrow e_0 : T} \tag{5}$$

$$\frac{S \models e_1 : \text{T}, \qquad S \models e_2 : \tau, \qquad S \models e_3 : \tau}{S \models \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \tag{6}$$

Rule 1. When the expression is a variable it must be defined in the current environment. A variable is always bound to a term.

Rule 2. An expression that binds a variable. The variable must not be present in the current environment. The resulting environment contains a binding of the variable. If we want to permit binding expressions when the left-hand side variable is already bound (as things work in Erlang today) the condition $v \notin S$ should be removed. Pattern match expressions can be handled similarly.

Rule 3. Evaluation of a pair of expressions (e_1, e_2). If e_1 evaluates to a term (3a) the value is discarded and e_2 is evaluated. If e_1 evaluates to a binding environment, e_2 is evaluated in that environment (3b).

Rule 4. In a function call, the expression giving the function and all parameters must all return terms. The result of a function call is also a term.

Rule 5. The body of an anonymous function must evaluate to a term (this rule could be relaxed). An anonymous function is a term.

Rule 6. The condition of an if-expression must always evaluate to a term. The then- and else-branches of an if may return a value or export bindings, but they must agree and either both return a term, or return binding environments with the same domains.

One might consider a slightly weaker version of Rule 6 in which one branch may introduce a binding that is not defined in the other branch:

$$\frac{S \models e_1 : \mathrm{T}, \qquad S \models e_2 : d(S_2), \qquad S \models e_3 : d(S_3)}{S \models \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : d(S_2 \cap S_3)} \tag{5'}$$

(Only bindings present in both branches will be returned.)

Given rules 1 through 5 we can deduce, for example, that the expression

```
X = 42, if X>X*X then Y=X-1 else Y = X+1
```

will have the type $d(\{X, Y\})$ when evaluated in the empty environment. The expression

```
X = 42, (io:format("hello"), Y = 99)
```

will have the same type. However,

```
if X>0 then Y=X-1 else 42
```

cannot be typed.

The whole point of this exercise was to re-formulate in simple mathematical language what currently is only explained in complex prose.

We will rule out some code that is currently legal, but it should be straightforward to rewrite such code. For example, an expression may either produce bindings *or* a result, but may never do both. In an expression `(X = 42, X*X)` the value of `X times X` is returned, the binding of `X` is *not* exported. Similarly, the body of a `fun` may contain bindings, but they may not be exported.

5.1 Summary

A mechanism for introducing local bindings, similar to the current one, but with a more cleanly defined semantics.

6 A mini language

This section was inspired by a recent white paper [10], but similar ideas have been explored in the high-performance computing community.

The idea is to look at a very restricted imperative language, say

- bounds of loops must be constant or affine (a linear combination of loop indexes)
- conditions of if-statements must be affine

- array indexes must be affine

Here, any value that is unchanged within the loop is considered to be constant.

In the white paper, it was shown how a compiler could map code of this type (implementing a video encoder) to the CELL architecture [2]. (The cell architecture is a rather attractive, low cost design intended for computer games containing 8 vector processors (SPE's) connected in a network).

Note that in the context of Erlang, code written in this restricted language satisfies many useful properties. The code is guaranteed to terminate, further, we can actually estimate the execution time with reasonable accuracy (as a function of the parameters). This in turn means that mini-language code could be set up to execute directly in an Erlang process, with no adverse effects on scheduling. Another interesting point is that one can set strong bounds for the amount of memory that will be allocated by the mini-language code. This should simplify the interaction with the GC. It is acceptable if the code uses side-effects internally, as long as they are not visible to the surrounding code. As long as the code is limited in its data types it is possible to determine through analysis that this is the case.

A mini-language code fragment would input and output Erlang values of those types that it can conveniently handle; integers (of specified range) and floating point numbers, tuples of integers and floats, and binaries. Compile-time analysis would guarantee that it does not modify its input.

An initial goal might be to develop a mini-language compiler that produces code with performance comparable to C, but ultimately one would like the mini-language compiler to take advantage of inexpensive vector processors which no doubt will be (even more) common in the future and vectorize the code. (There are of course other types of parallel computers, but they all benefit from source code that is easy to analyze and transform.)

There are some algorithms that are often needed in typical Erlang applications that are hard to implement with maximum efficiency in Erlang today; for example data compression, encryption and signal processing. These should be a good fit for the mini-language.

Moreover, the combination of a low-level vectorizable language and a concurrent language coordinating computations on a higher level would be ideal for machines that combine different types of parallelism. Erlang could be used as a coordination language as Erlang processes offer coarse-grain parallelism while the mini-language code would take full advantage of vector processors.

6.1 Design of the mini language

Regardless of the syntactic form, the mini-language will have to satisfy the following:

- Loops must be bounded (this implies that the mini-language would not be Turing complete).
- All data structures must have static type information.
- Side-effects must not be visible outside the mini-language code.
- It should be possible to determine the running time of the mini language code by static analysis. If a computation is determined to "take too long"

it either needs to be run on a separate process (to guarantee real-time properties) or divided into sub-computations.

In this example I chose a C-like syntax because I expect most readers to be familiar with this. It is of course very reasonable to consider other notations.

```
dot_product(T1, T2) ->
   <mini-language>
      tuple(int32) T1, T2; // T1 and T2 should be tuples of integers
                           // in the range -2^31 .. 2^31-1
      int32 p = 0;
      for(int32 i=0; i++; i< T1.length) {
          p =+ T1[i] * T2[i];
      }
      return p;
   </mini-language>.
```

Note that the type declarations of the input data cannot be trusted blindly. The mini-language code must check that `T1` and `T2` are indeed tuples, and that each element is indeed a 32-bit integer. The type checking might be done incrementally, as the elements of the tuples are accessed—again, the restricted form will simplify analysis and allow the compiler to ensure that type checking is only done once, and only on data that will be accessed by the mini-language code. The for-loop is bounded by the size of the first tuple. Since the size of a tuple does not change, this value is constant in the context of the loop. If the tuple `T2` is smaller than `T1`, the mini-language code should generate an error.

The mini-language may be imperative in style or functional, but note that the presence of side-effects will actually complicate analysis, transformation, and parallelization and make the language harder to implement efficiently. It may be worthwhile to take a look at SISAL, a functional language designed for parallel computers [13]. The notation of list comprehensions might also serve as a starting point.

One of the challenges in the design of the mini language is keeping it simple, in order to allow precise reasoning about dependencies and execution times. It should integrate well with the Erlang code, and (of course) be appropriate for expressing a wide range of low-level algorithms.

Finally, a quote from the Erlang FAQ, and keep in mind that this is precisely the class of problems the mini-language would target.

> "The most common class of 'less suitable' problems [for Erlang] is characterised by performance being a prime requirement and constant-factors having a large effect on performance. Typical examples are image processing, signal processing, sorting large volumes of data and low-level protocol termination."

7 Lisp-style Macros

Now, Erlang has already a front end that implements Lisp style macros [14] so I'll be brief.

Lisp's macros [12, Chapter 7] offer a way to extend the syntax of the language. Suppose that you find that the current control structures are insufficient,

or you want a new way to write expressions or even a new way of defining things. In all cases, Lisp style macros would allow you to do it. In Erlang, and in most other programming languages you would have to modify the implementation to extend the language in this manner.

One more example: It has been noticed for a long time that it would be very useful to have a variation of Erlang's if-expressions that allowed arbitrary expressions as guards ('`cond`' has been reserved as a keyword for this purpose). Now, in a language with Lisp-style macros it would be a trivial exercise to define a macro with this semantics.

A final example: Using a parser generator such as Yecc (Erlang's Yacc) is probably the most efficient way to build a parser. Still, it's a rather unattractive system with its own, crude, syntax. A yecc file is first translated to Erlang code which then has to be compiled, thus complicating the build process.

In the (incomplete) example below, I show what a Yacc-style grammar would look like in Common Lisp. There are two macros to define the grammar; `defparser` defines a parser with start symbol, terminals and non-terminals, and `def-nt` defines the productions of a non-terminal. (The LR-tables can either set up at compile-time, or when the parser is first run.)

```
(defparser expression-parser
    :start-symbol expression
    :terminals (int id + - * / |(| |)|)
    :non-terminals (expression term factor))

(def-nt expression
  ((expression + term) (+ $1 $3))
  ((expression - term) (- $1 $3))
  ((term) $1))
```

Now, it should be said that writing Common Lisp macros is harder than defining ordinary function. It is still very worthwhile for libraries and other code that is intended to be re-used many times.

One issue that needs to be resolved is the use of a macro that is defined in another module. It is straight-forward to apply the rules for macro-expansion as long as the other module has been loaded before the current one. However, what happens is the module containing the macro definition is reloaded? One solution is to require the two modules to be linked, as discussed in Section 4.

Must a language with Lisp-style macros have a Lisp-like syntax? It should not be necessary; any language with a well-defined internal representation could in principle be equipped with macros. I don't know of any language that combines Lisp-style macros with a syntax that isn't Lisp-like, but it would be interesting to see how it would work out in practice.

Acknowledgments

I would like to thank Per Gustafsson and Tobias Lindahl for comments on earlier versions of this paper.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] Nicholas Blachford. Cell architecture explained version 2, 2005. http://www.blachford.info/computer/Cell/Cell0_v2.html.

[3] Jack B. Dennis. First version of a data flow procedure language. Technical Report 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975. First published in B. Robinet (ed), *Programming Symposium: Proceedings Colloque sur la Programmation*, Springer LNCS 19, April 1974., 362-376.

[4] erlang-in-lisp manual, August 2008. http://common-lisp.net/project/erlang-in-lisp/.

[5] Klaus Harbo. cl-muproc: Erlang-inspired multiprocessing in Common Lisp. In Edi Weitz and Arthur Lemmens, editors, *European Common Lisp Meeting 2006*, 2006.

[6] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, pages 471–475. North-Holland, 1974.

[7] Thomas Lindgren. Module merging: aggressive optimization and code replacement in highly available systems. UPMAIL Technical Report 154, Computing Science Department, Uppsala University, March 1998.

[8] Thomas Lindgren. Cross-module optimizations. In *Proceedings of the seventh Erlang user conference*, September 2001.

[9] Richard O'Keefe. Abstract patterns for Erlang. In *Fourth International Erlang/OTP User Conference*, September 1998.

[10] Parallelization using polyhedral analysis. CoSy White Paper, March 2008. http://www.ace.nl/compiler/paper-polyhedral.pdf.

[11] Kent Pitman, editor. *The Common Lisp Hyperspec*. LispWorks, 2005. http://www.lispworks.com/documentation/common-lisp.html.

[12] Peter Seibel. *Practical Common Lisp*. Apress, 2005. http://gigamonkeys.com/.

[13] A tutorial introduction to Sisal. http://www2.cmp.uea.ac.uk/~jrwg/Sisal/.

[14] Robert Virding. Lisp Flavoured Erlang, March 2008. The Erlang Questions mailing list.