Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 191–202
doi: 10.14794/ICAI.10.2017.191

Analysis of Erlang source dependencies in BEAM bytecode*

Dániel Lukács, Melinda Tóth

ELTE Eötvös Loránd Unviversity, Budapest, Hungary
{dlukacs,tothmelinda}@caesar.elte.hu

Abstract

Source dependencies are mostly stored in executable format, and as they can seldom be statically analysed, industrial developers often have to treat these as black boxes. While it would be cost-effective to statically analyse these executables, such endeavours are impeded by the considerable structural and semantical differences between low-level and high-level code. In this paper, we present an algorithm that recovers syntactically valid Erlang syntax trees from low-level BEAM bytecode. This syntax tree then can be used in static analysis, to effectively communicate the semantical content of the dependencies.

Keywords: static analysis, decompilation, BEAM

MSC: 68N15

1. Introduction

Developers of industrial-scale software, with several hundred thousand LOC, often need to rely on external tools such as debuggers and static analysis support, to understand the internal workings of their developed software. Source dependencies are mostly stored in executable representation, which is different from the source language. With time, the original sources of these dependencies can become unavailable in various ways. In these cases, static analysis frameworks not prepared to handle low-level library representations has to stop their analysis at these dependencies, and developers have to fall back to considering these as black boxes. Our goal in this paper is to enable the static analysis of Erlang library dependencies in BEAM bytecode format.

*This research has been supported by the Hungarian Government through the New National Excellence Program of the Ministry of Human Capacities.

The following list summarizes our contributions in this paper.

- We introduce an algorithm that enables the static analysis of Erlang library dependencies in BEAM bytecode format.
- The introduced algorithm represents the collected semantical informations in syntactically valid Erlang syntax tree, that is semantically equivalent, and syntactally similar to the original Erlang sources.
- The generated Erlang code, representing the semantics of the original code, then is readily analysable by RefactorErl.
- We intended to design the framework to general and adaptable enough, so that in the future it can be extended for other emerging languages targeting the BEAM platform, such as Elixir [11].

2. Problem statement

The Erlang programming language [3] is a popular, functional programming language, commonly used to develop multithreaded, distributed, fault-tolerant applications, including telecommunication systems, web servers, and distributes databases. The library dependencies of Erlang programs are usually stored in low-level BEAM bytecode format, which is interpreted by the *BEAM virtual machine* [18]. BEAM, the VM supports efficient and automatic memory allocation via garbage collection and intensely scalable multi-threading via lightweight-threads. BEAM, the language is an imperative, register-based language. Figure 1 depicts a small Erlang function and Figure 2 depicts the low-level bytecode representation of the aforementioned Erlang function. It strongly resembles low-level assembly code, implementing control flow using jump, and conditional jump instructions. Unlike conventional machine instruction sets, BEAM also contains fairly high-level instructions to simplify the expression and execution of the semantics assigned to certain Erlang structures, but also to command the VM to perform tasks related to memory management and thread scheduling.

The RefactorErl [17, 4] framework is an open-source tool for performing static analysis on Erlang software. At its current state, RefactorErl is only capable to completely analyse source code in Erlang. Our goal in this paper is to prepare the RefactorErl framework to provide comprehension facilites for BEAM bytecode, thus allowing it, to analyse all source dependencies, even if they are stored as bytecode.

To achieve this, we first have analyse these bytecode modules, represent the collected semantical informations as syntactically valid Erlang syntax tree, that is semantically equivalent, and syntactally similar to the original Erlang sources, so it can be readily added to the RefactorErl framework, so its already defined static analysis facilities can reused for this purpose, without restrictions.

```
-module(event_handler).
-export([handle_event/1]).

handle_event(Event) ->
  case Event of
    ok -> done;
    {message, _} -> done;
    _ -> unknown_event
  end.
```

Figure 1: Simple Erlang module with a function definition

```
{function, handle_event, 1, 2}.
{label,1}.
{line,[{location,"event_handler.erl",4}]}.
{func_info,{atom,event_handler},
         {atom,handle_event},1}.
{label,2}.
{test,is_tuple,{f,3},[{x,0}]}.
{test,test_arity,{f,5},[{x,0},2]}.
{get_tuple_element,{x,0},0,{x,1}}.
{test,is_eq_exact,{f,5},[{x,1},
                {atom,message}]}.
{jump,{f,4}}.
{label,3}.
{test,is_eq_exact,{f,5},[{x,0},{atom,ok}]}.
{label,4}.
{move,{atom,done},{x,0}}.
return.
{label,5}.
{move,{atom,unknown_event},{x,0}}.
return.
```

Figure 2: Bytecode representation of the `handle_event/1` function

3. Methodology

Compilation, in general, is not a deterministically invertible operation.Therefore, there is no algorithm, that can completely reconstruct (*decompile*) the original source code from the target code, without having some kind of additional information about the original source code. Such lost information are variable names, syntactical sugar, macros and dead code. In addition the compiler may generate new function definitions to compile anonymous functions, that are indiscernible from the original top-level user-defined functions.

In this section, we present a framework for generating high-level, human-readable Erlang code from low-level BEAM bytecode, which is structurally similar to the original source. While the individual steps of the decompiler pipeline are more-or-less well-known techniques in compiler design and analysis, our unique goal of reverse engineering a functional programming language called for a unique blend of the application of the techniques.

3.1. The beginnings: BEAM bytecode

This information loss is almost entirely limited to the syntax of the source code: we almost always expect the source and the target to be completely semantically equivalent, with the only exception being the differences introduced by compiler optimizations.

Still, we do not expect bytecode compilers to produce heavily optimized target bytecode, since the primary purpose of these compilers is to relieve the interpreter

(i.e. the virtual machine) from the complex task of the lexical/syntactical/semantical analysis of a high-level programming language. In our observation, the official `erlc` Erlang compiler also avoids heavy optimizations. It only eliminates dead code detectable in compile-time, and it introduces jumps to avoid redundant generation of certain instructions. Still, even the reversing of these simpler optimizations can be a complex task for structuring.

3.2. Explicitly expressing semantics by using intermediate representation (IR)

As there is no official specification published for the BEAM, there is no guarantee that the instruction syntax and semantics will be kept unchanged between different versions, and different implementations of the Erlang VM. The introduction of IRs reduces our efforts from designing $m \times n$ compilers, to designing just 1 compiler and specifying $n + m$ translations [1].

Another motivating factor to use an IR, is that most of the BEAM instructions have *implicit semantics*: the syntax of such instructions does not explicitly refer to all the parameters affected by the instruction. The translation rule in Figure 3 illustrates how the IR makes explicit the meaning of the BEAM instruction `{call_ext_last, ARITY, {MODULE,FUNCTION,ARITY}, SIZE}`.

$$\frac{\{\texttt{call\_ext\_last}, \texttt{N}, \texttt{MFA}, \texttt{D}\}}{\begin{array}{c}\{\texttt{x},0\} := \{\texttt{call}, \texttt{MFA}, [\{\texttt{x},0\}, ..., \{\texttt{x}, \texttt{N}-1\}]\}; \\ \texttt{vm}\ \{\texttt{deallocate}, \texttt{D}\}; \\ \texttt{return}\ \{\texttt{x},0\};\end{array}}$$

Figure 3: A translation rule for transforming BEAM to IR

It can be seen, that – without an IR – to handle m instructions that perform n tasks, we need to prepare the decompiler to analyze $m \times n$ different cases. (In this example $m = 1$ and $n = 4$.) If we introduce an IR, by first translating the m BEAM instructions each to n IR instructions performing exactly one task, the decompiler only needs to iterate over $m \times n$ simple, well-understood and explicit IR instructions, instead of doing the same amount of complex case-by-case analysis. This considerably reduces the required developer effort.

3.3. Efficient handling of jump instructions by using control flow graphs (CFGs)

Low-level languages often implement control flow by using jump instructions (also called `goto` instructions). By treating these jumps as references to specific program points, the idea may naturally emerge to represent the control flow of low-level programs as a graph, and as such, we may consider CFGs as the foundation of most decompiling techniques. In our case, the SSA-form (Single Static Assignment) and

```
procedure handle_event [{x,0}] at 2:
1:
throw {function_clause,{atom,event_handler},
                       {atom,handle_event},1};
2:
if not is_tuple [{x,0}] then goto 3;
if not test_arity [{x,0},2] then goto 5;
{x,1} := {x,0}[0];
if not is_eq_exact [{x,1},{atom,message}]
                               then goto 5;
goto 4;
3:
if not is_eq_exact [{x,0},{atom,ok}]
                          then goto 5;
4:
{x,0} := {atom,done};
return {x,0};
5:
{x,0} := {atom,unknown_event};
return {x,0};
```

Figure 4: Intermediate representation of the bytecode of the `handle_call/1` function

the structuring algorithm will both be based on the CFG. In the following sections we assume the reader is familiar to the fundamental concepts of CFGs, such as basic blocks, flows, entry and exit nodes, choice and join nodes [1]. We will also lean on the concepts of dominance, postdominance and dominator frontier, defined in [7].

3.4. Restoring the single assignment property

During code generation we will have to map BEAM registers to Erlang variables. In general, mapping registers to variables is the inverse problem of the register allocation problem in compiler design, and – unlike the original problem – it can be easily solved.

Unlike Erlang variables, the number of registers in the BEAM is bounded, therefore BEAM bytecode programs will necessarily overwrite the content of some registers. As a consequence, we can not satisfy the single assignment property with a simple one-to-one mapping from registers to variables.

Our goal then, is to map BEAM registers to Erlang variables, in a way so that each variable is only assigned once. To do so, we will employ Static Single Assignment (SSA), a special control flow representation introduced by [7]. The SSA graph is topologically and semantically equivalent to the original CFG, but additionally it satisfies the constraint that each variable is only assigned once during its lifetime. Every CFG can be lifted to SSA form.

With this, we may see the SSA as a mere fix to achieve the single assignment property. Later in this section we shall see that the connection between SSA and Erlang (actually between SSA and functional programming languages in general)

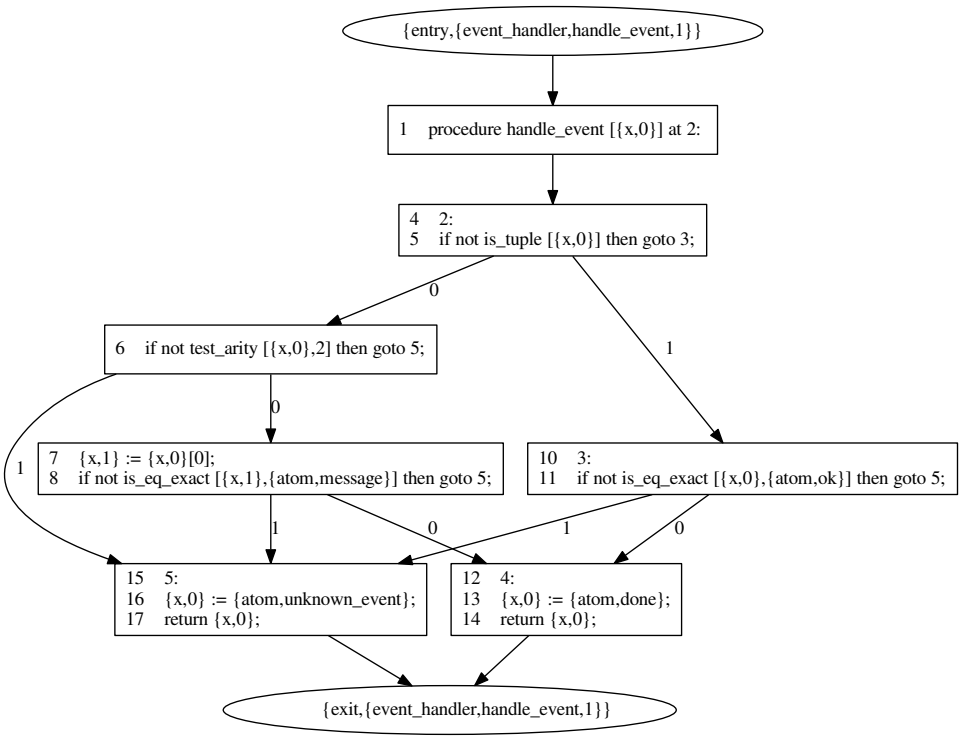

Figure 5: Control flow graph created from the IR of the `handle_call/1` function

goes deeper, and that we can use this connection to transform the imperative BEAM IR into a semantically equivalent, but functional IR.

3.5. A short preview of control flow structuring

Similarly to conventional machine code, bytecode control flow is also expressed using jump and conditional jump instructions. The main advantage of jumps is that they can be directly executed by the (virtual) machine, while their main disadvantage is that they can be used to implement unstructured control flow [8].

A CFG is said to be *structured* if there exist a partitioning over the CFG edge set, that designates subgraphs such that all of these subgraphs can be mapped to a high-level structure [6]. From now on, we call these subgraphs *regions*. If such a partitioning can not be found, the CFG is said to be *unstructured*. Then, the goal of *structuring algorithms* is to transform an unstructured CFG into a structured one, and then decompose this CFG into a tree of acceptable regions. By matching specific subregions to specific high-level structures, this tree of regions then can be mapped to the syntax tree of a complete program, specified in a high-level programming language. Unfortunately, at least for the prospective analyzer, the

`erlc` Erlang compiler also introduces unstructuredness in the BEAM control flow which has to be eliminated.

Jumping instructions can be represented as `goto`-statements (not possible in Erlang) or function calls (see [2, 5]). Redundant code duplication can also be employed to eliminate certain jumps, but duplicating every choice node in a DAG with n binary choice node may result in a tree with as much as 2^n choices. Instead, we have chosen to use special case-by-case analysis techniques, that can produce control structures that are syntactically closer to the original. If the unstructuredness is still present after applying these special techniques, we can still fallback to the less accurate, but more general methods.

To specify some of the procedures applied in this step, we rely on *term graph rewriting (TGR)* formalism [9, 16]. The TGR representation is expressive, executable and has the potential to be used effectively in a formal analysis of the algorithm.

In general, our structuring algorithm performs the following steps:

1. Preliminary detection of non-branching structures, whose corresponding BEAM CFG features conditionals (eg. list comprehension).

2. Structuring of compound conditions and contracting conditional CFG patterns into one level n-way condtionals.

3. Local detection of structures, including those that feature branching expressions.

4. Transformation of the CFG to a syntax tree of a functional intermediate representation, syntactically close to Erlang. Here we identify regions corresponding to expressions, and transform a graph into an expression tree, by using the SSA information.

Further elaboration on the structuring step of the pipeline can be found in [14].

3.6. Making it functional: Transforming SSA to A-normal form

The SSA transformation introduces ϕ-functions on the dominance frontier of the problematic choice nodes, for each variable that is still considered living at the frontier point. The evaluation of such ϕ-functions would require CPUs (or software interpreters) to keep track of which variable was assigned on which control flow branch: an operation usually not supported in commonly used hardware and software. Therefore, it is up to the compilers (and decompilers) to eliminate ϕ-functions from SSA-form before code generation. We will achieve this by transforming SSA to a functional IR.

It is argued in [2] that SSA, without any modification, can be already considered a functional program flow description. As a summary, we may basically consider the SSA as a call graph, where the SSA blocks represent functions, SSA flows represent call-relations, and ϕ-assignments specify the formal and actual parameters of these

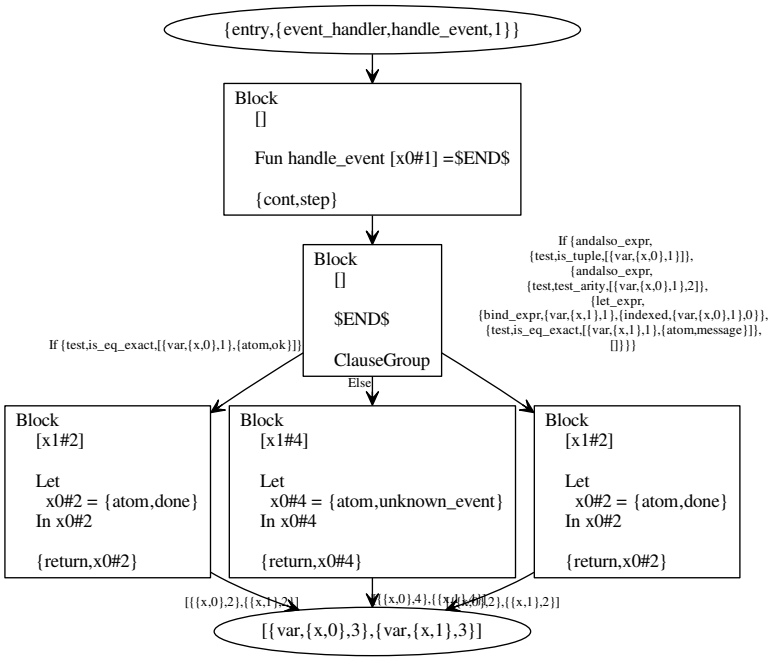

Figure 6: Control Flow Graph after structuring transformations

functions. The two potential target functional representations and their transformations reviewed by us are the A-Normal Form (ANF) [5], and the Continuation Passing Style (CPS) [13]. As among our goals were readability and syntactical resemblance to the original source, we chose ANF, as the base of the intermediate representation.

3.7. Generating Erlang code from structured CFG

The goal of the code generation phase is to utilize all information gathered until this point, and finally produce the Erlang source code, that is semantically equivalent to the BEAM bytecode input, and syntactically resembles the original Erlang source which was compiled into BEAM.

The main advantage of using a functional IR instead of directly using Erlang, comes from its simplicity: its less complicated to parse the structured SSA into ANF, than it is to parse it into Erlang. The ANF-like syntax tree is easier to traverse and analyze. And again, translating an ANF-like syntax tree to Erlang syntax tree is less complicated, than it is to parse the structured SSA directly to Erlang.

While much of the extended ANF can be straightforwardly mapped to an Erlang syntax tree, there are still expressions in the ANF that have no direct correspondance in Erlang: for example in Figure 7, the guard features variable binding, which would

```
FUN handle_event [x0#1] =
  IF
    WHEN
      ANDALSO
        {test,is_tuple,[x0#1]}
        ANDALSO
          {test,test_arity,[x0#1,2]}
          LET x1#1 = {indexed,x0#1,0}
          IN  {test,is_eq_exact,[x1#1,{atom,message}]} ->
      LET x0#2 = {atom,done}
      IN x0#2
    WHEN
      {test,is_eq_exact,[x0#1,{atom,ok}]} ->
      LET x0#2 = {atom,done}
      IN x0#2
    WHEN true ->
      LET x0#4 = {atom,unknown_event}
      IN x0#4
```

Figure 7: Functional IR code generated from the SSA, after the structuring phase

be invalid syntax in Erlang. The detection and removal of these from the guards can be realized by a simple semantical analysis step, where we infer the patterns based on the information collected from the variable bindings, or even the guards.

```
-module(event_handler).
-export([handle_event/1]).

handle_event(X0_1) ->
    if
        (is_tuple(X0_1)
        andalso (size(X0_1) =:= 2
        andalso element(1, X0_1) =:= message)) ->
            X0_2 = done,
            X0_2;
        X0_1 =:= ok ->
            X0_2 = done,
            X0_2;
        true ->
            X0_4 = unknown_event,
            X0_4
    end.
```

Figure 8: The final Erlang source generated from the functional IR

```
-module(event_handler).
-export([handle_event/1]).

handle_event(Event) ->
  case Event of
    ok -> done;
    {message, _} -> done;
    _ -> unknown_event
  end.
```

Figure 1: Function definition in the original source dependency

In Figure 8, the final generated Erlang code can be seen. While it semantically equivalent to the code in Figure 1, and the structure of the generated code resembles the original, there are still unused variables, unnecessary variable bindings and redundant conditions, remnants of the BEAM windowing generation, the explicitness

of the IR, and the unoptimized SSA. Live variable analysis, iterated variable elimination, and semantic analysis of the condition can be utilized to get the original and the generated code even closer to each other.

4. Related work

The majority of the literature reviewed by us is concerned with the decompilation reverse machine code generated from programs originally written in imperative languages, such as C. Examples of such decompilers are dcc [6], Hex-Rays [10], and Dream [12]. In general, architecture-dependent machine code is heavily optimized, features dynamic memory addressing, and does not segment code and data, this is a more difficult problem in general. On the other hand, as Erlang is a functional language, we had to extend the convential practices with new techniques, such the structuring of functional pattern matching expressions (detailed in our current paper), and the inclusion of the SSA-to-ANF transformation [5].

Similar comparisons can be made between machine code and other bytecode languages, like the bytecode of Java Virtual Machine (JVM). [15] describes a grammar-based approach to decompile JVM bytecode. As one of our future goals is to target different compilers and languages of the BEAM platform, and therefore we decided for using an approach, which – albeit methodologically less uniform – promises easier extendability and adaptability in support of future development.

5. Conclusions

In this paper we presented an algorithm, in the form of a decompiler pipeline, for decompiling BEAM bytecode, that allows Erlang developers to use RefactorErl to statically analyse bytecode dependencies in a similar way, as if they were represented in the Erlang language.

The framework achieves this by first analysing the BEAM, and then representing the collected semantic information in syntactically valid Erlang representation, that is semantically equivalent, and syntactically similar to the original Erlang sources.

In the decompiler pipeline each function goes through this pipeline independently, thus it allows us to parallelize the evaluation of our algorithm. The generated Erlang code, as a representation of the semantics of the original bytecode, is then readily analysable by RefactorErl.

While the decompiler pipeline together with the presented structuring algorithm solves the more involved problems of syntax tree recovery, further work still must be done to increase syntactical accuracy (eg. recognition of match expressions, list comprehensions, anonymous functions), to guarantee full coverage of all possible BEAM programs (exception handling), and to extend the framework for other emerging languages, such as Elixir, that target the BEAM platform.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers principles, techniques, and tools, *Addison-Wesley, Reading, MA*, 1986.

[2] Andrew W. Appel, SSA is functional programming, *ACM SIGPLAN NOTICES*, 33(4):17–20, 1998.

[3] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[4] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, M. Tejfel, and M. Tóth, Refactorerl - source code analysis and refactoring in Erlang, In *Proc. of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, 2011.

[5] Manuel M.T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski, A functional perspective on SSA optimisation algorithms, *Electronic Notes in Theoretical Computer Science*, 82(2):347 – 361, 2004.

[6] Cristina Cifuentes, *Reverse Compilation Techniques*, PhD thesis, Queensland University of Technology. School of Computing Science, 1994.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, An efficient method of computing static single assignment form, In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[8] Edsger W. Dijkstra, Letters to the editor: Go to statement considered harmful, *Commun. ACM*, 11(3):147–148, 1968.

[9] H. Ehrig, M. Pfender., and H. J. Schneider, Graph-grammars: An algebraic approach, In *Proc. of the 14th Annual Symposium on Switching and Automata Theory*, pages 167–180. IEEE Computer Society, 1973.

[10] Ilfak Guilfanov, Decompilers and beyond, *Presentation at Black Hat USA*, 2008.

[11] Plataformatec, José Valim, Elixir programming language, http://elixir-lang.org/, 2017.

[12] E. Gerhards-Padilla K. Yakdan, S. Eschweiler and M. Smith, No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations, *The 2015 Network and Distributed System Security Symposium*, At San Diego, CA, USA, 2015.

[13] Richard A. Kelsey, A correspondence between continuation passing style and static single assignment form, In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM.

[14] D. Lukács, *Recovering high-level control structures from erlang source dependencies in beam bytecode*, TDK Thesis, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, 2017.

[15] Godfrey Nolan, *Decompiling Java*, Apress, 2004.

[16] Rinus Plasmeijer and Marko Van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.

[17] M. Tóth and I. Bozó, Static analysis of complex software systems implemented in erlang, *Central European Functional Programming Summer School, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS)*, Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[18] Robert Virding, Hitchhiker's tour of the beam, *Presentation at Erlang Factory SF Bay Area*, 2013.