# Metaprogramming and symbolic execution for detecting runtime errors in Erlang programs

Emanuele De Angelis, Fabio Fioravanti
DEC, University "G. d'Annunzio" of Chieti-Pescara
Viale Pindaro 42, 65127 Pescara, Italy
{emanuele.deangelis, fabio.fioravanti}@unich.it

Adrián Palacios
MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
apalacios@dsic.upv.es

Alberto Pettorossi
University of Roma Tor Vergata
Via del Politecnico 1, 00133 Roma, Italy
pettorossi@info.uniroma2.it

Maurizio Proietti
CNR-IASI
Via dei Taurini 19, 00185 Roma, Italy
maurizio.proietti@iasi.cnr.it

Dynamically typed languages, like Erlang, allow developers to quickly write programs without explicitly providing any type information on expressions or function definitions. However, this feature makes those languages less reliable than statically typed languages, where many runtime errors can be easily detected at compile time. In this paper, we present a preliminary work on a tool that, by using the well-known techniques of metaprogramming and symbolic execution, can be used to perform bounded verification of Erlang programs. In particular, by using Constraint Logic Programming, we develop an interpreter that, given an Erlang program and a symbolic input for that program, returns answer constraints that represent sets of concrete data for which the Erlang program generates a runtime error.

## 1 Introduction

Erlang [7] is a functional, message passing, concurrent language with dynamic typing. Due to this type discipline, Erlang programmers are quite familiar with typing and pattern matching errors at runtime, which normally appear during the first executions of freshly written programs. Less often, these errors will be undetected for a long time, until the user inputs a particular value that causes the program to crash or, in the case of concurrent programs, determines a particular interleaving that causes an error to occur.

In order to mitigate these problems, many static analysis tools have been proposed. Here let us recall: (i) Dialyzer [5], which is a popular tool included in Erlang/OTP for performing type inference based on success typings, and (ii) SOTER [3], which is a tool that performs verification of Erlang programs by using model checking and abstract interpretation. However, those tools are not all fully automatic, and they can only be used to cover either the sequential or the concurrent part of an Erlang program, but not both.

In this paper we present a preliminary work on a technique, based on Constraint Logic Programming (CLP) [4], for detecting runtime errors in Erlang programs. In our approach, sequential Erlang programs are first translated into CLP terms and then run by using an interpreter written in CLP. Our CLP interpreter is able to run program on symbolic input data, and it can perform verification of Erlang programs up to a fixed bound on the number of execution steps.

Submitted to:
HCVS 2018

**The Erlang language.** In this work we only consider sequential programs written in a first-order subset of the Erlang language. In this language, a module is a sequence of function definitions, where each function name has an associated definition of the form: fun $(X_1,\ldots,X_n)$ -> *expr* end (for simplicity, we assume that programs are made out of a single module). The body of a function is an expression *expr*, which can include literals (atoms, integers, floats, or the empty list), variables, list constructors, tuples, let expressions, case expressions, try/catch blocks, function applications, and calls to built-in functions.

```
-module(sum_list).
-export([sum/1]).

sum(L) ->
 case L of
   [] -> 0;
   [H|T] -> H + sum(T)
 end.
```

Figure 1: A program in Erlang that computes the sum of all numbers in the input list L.

The Erlang program in Figure 1 will successfully compile with no warnings in Erlang/OTP and will correctly compute the sum of the elements in L provided that L is a list of numbers. Otherwise, the program generates a runtime error. For instance, if the input to sum is an atom, then the program crashes and outputs a pattern matching error (match_fail), because there are no patterns that match an atom. Similarly, if the input to sum is a list of values, where at least one element is an atom, the execution halts with a type error (badarith), when applying the function '+' to a non-numerical argument.

The tool Dialyzer does not generate any warnings when analyzing this program. That is coherent with the Dialyze approach, which only complains about type errors that would guarantee the program to crash. However, it might be the case that we want to perform a more detailed analysis on this program. In the following, we will see how our tool lists all the potential runtime errors, together with the input types that can cause them.

## 2 A symbolic interpreter for detecting Erlang runtime errors

The main component of the verifier is a CLP interpreter that defines the operational semantics of Core Erlang[1]. This executable specification of the semantics enables the execution of Erlang programs represented as Prolog terms. (The translator generates one term for each function definition.)

The interpreter provides a flexible means to perform the *bounded verification* of Erlang programs. Indeed, by using a symbolic representation of the input data, the interpreter allows the exhaustive exploration of the program computations without explicitly enumerating all the concrete input values. In particular, the interpreter can run on input terms containing variables, and it uses constraint solvers to manipulate expressions with variables ranging over integer or real numbers. By fixing a bound to limit the number of computation steps performed by the interpreter, we force the exploration process to be finite, and hence either we detect a runtime error or we prove that the program is error-free up to the given bound.

Let us consider an Erlang program Prog which is represented as a set of Prolog facts of the form fun(FName/Arity,Pars,Body), where Pars and Body represent the parameters and the body, respectively, of a function named FName of arity Arity.

In order to perform the bounded verification of the Erlang program Prog, the interpreter provides the predicate run(FName/Arity,Bound,In,Out), whose execution evaluates the application of the function FName to the input arguments In, by constructing an evaluation tree of depth at most Bound. The arguments In are represented as a Prolog list (written using square brackets) of length Arity. The result is represented by Out. If the evaluation of the function application generates an error, then Out is

[1] Core Erlang is the intermediate language used by the standard Erlang compiler, which removes most of the syntactic sugar present in Erlang programs.

bound to a term of the form `error(Err)`, where `Err` is an error name (e.g., `match_fail`, indicating a match failure, or `badarith`, indicating an attempt to evaluate an arithmetic function on a non-arithmetic input), meaning that the specific error `Err` had occurred. Hence, the bounded verification of a given Erlang program can be performed by executing a query of the form:

```
?- run(FName/Arity,Bound,In,error(Err)).
```

where `FName` is a constant, `Arity` and `Bound` are non-negative integers, and `In` and `Err` are, possibly non-ground, terms.

Any answer to the query is a successful detection of the error `Err` generated by evaluating the application of the function `FName` to the input `In`. If no answer is found, then it means that no error is generated by exploring the computation of `FName` up to the value of `Bound`, and we say that the program `Prog` is correct up to the given bound.

Now let us see the bounded verifier in action by considering the `sum_list` program of the previous section and the following query:

```
?- run(sum/1,20,In,error(Err)).
```

Among the answers to the query, we get the following constraints relative to the input `In` and the error `Err`:

```
In=[cons(lit(Type,_V),lit(list,nil))], Err=badarith, dif(Type,number)
```

meaning that if `sum/1` takes as input a list (represented as a Prolog term of the form `cons(Head,Tail)`) whose head is not a numeric literal (denoted by the constraint `dif(Type,number)`), then a `badarith` error occurs, that is, a non-numerical argument is given as input to an arithmetic operator. Another answer we get is:

```
In=[L], Err=match_fail, dif(L,cons(_Head,_Tail)), dif(L,lit(list,nil))
```

meaning that if `sum/1` takes as input an argument `L` which is neither a `cons` nor a `nil` term, then a `match_fail` error occurs. Note that, due to the recursive definition of `sum`, the bound is essential to detect this error.

Let us now introduce the predicate `int_list(L,M)` that generates lists `L` of integers of length up to `M`. For instance, the query

```
?- int_list(L,100).
```

generates the answer

```
L=cons(lit(int,N1),cons(lit(int,N2),...))
```

where `L` is a list of length 100 and $\texttt{N1}, \texttt{N2}, \ldots, \texttt{N100}$ are variables. If we give `L` as input to `sum` as follows:

```
?- int_list(L,100), run(sum/1,100,L,error(Err)).
```

the bounded verifier terminates after 0.142 seconds[2] with answer `false`, meaning that if the input to `sum` is any list of 100 integers, then the program is correct up to the bound `100`. Note that no concrete integer element of the list is needed for the verification of this property.

Now we sketch the implementation of the operational semantics of Erlang expressions. The semantics is given in terms of a transition relation of the form `tr(Bound,ICfg,FCfg)` that defines how to get the final configuration `FCfg` from the initial configuration `ICfg` in `Bound` transition steps. Configurations are pairs of the form `cf(Env,Exp)`, where `Env` is the environment mapping program variables to values and `Exp` is a term representing an Erlang expression.

[2] the query has been executed using SWI-Prolog v7.6.4 (`http://www.swi-prolog.org/`) on an Intel Core i5-2467M 1.60GHz processor with 4GB of memory under GNU/Linux OS

The environment is extended with a boolean flag that keeps track of the occurrence of any runtime error during program execution. The value of the error flag `F` in the environment `Env` can be retrieved by using the predicate `lookup_error_flag(Env,F)`. The value of the flag in a given environment `EnvI` can be updated using the predicate `update_error_flag(EnvI,F,Env0)`, thereby deriving the environment `Env0` whose error flag is set to `F`. If the evaluation of `IExp` generates the error `Err`, then `FExp` is a term of the form `error(Err)` and the error flag is set to `true`.

```
tr(Bound,cf(IEnv,IExp),cf(FEnv,FExp)) :-
  IExp = apply(FName/Arity,IExps),
  lookup_error_flag(IEnv,false),
  Bound>0,
  Bound1 is Bound-1,
  fun(FName/Arity,FPars,FBody),
  tr_list(Bound1,IEnv,IExps,EEnv,EExps),
  bind(FPars,EExps,AEnv),
  lookup_error_flag(EEnv,F1),
  update_error_flag(AEnv,F1,BEnv),
  tr(Bound1,cf(BEnv,FBody),cf(CEnv,FExp)),
  lookup_error_flag(CEnv,F2),
  update_error_flag(EEnv,F2,FEnv).
```

Figure 2: definition of the operational semantics of `tr/3` for a function application `apply/2`.

In Figure 2 we present the clause for `tr/3` which implements the semantics of function applications represented using terms of the form `apply(FName/Arity,IExps)`, where `FName` is the name of a function of arity `Arity` applied to the expressions `IExps`. The transition only applies if: (i) no error has occurred so far, that is, `lookup_error_flag(IEnv,false)`, and (ii) the bound has not been exceeded, that is, `Bound>0`. Then, the function definition `fun(FName/Arity,FPars,FBody)` is retrieved and the following operations are performed: (i) the value of the bound `Bound` is decreased, (ii) the list of the actual parameters `IExps` is evaluated in `IEnv`, thereby deriving the list of expressions `EExps` and the new environment `EEnv` (it may differ from `IEnv` in the error flag and new variables occurring in the expressions `IExps` may have been added), (iii) the formal parameters `FPars` are bound to the expressions `IExps` to form the new environment `AEnv`, and (iv) the error flag in `AEnv` is updated to value `EEnv`, thereby deriving the environment `BEnv`. Finally, the body `FBody` is evaluated in `BEnv` to get the final expression `FExp`. The final environment `FEnv` equals to `EEnv` except for the error flag, which is set to the value obtained from the callee function.

Each rule of the operational semantics for Erlang programs is translated into a clause for the predicate `tr/3`. These clauses are omitted for lack of space.

We can now present the definition of `run/4`, which depends on `tr/3`:

```
run(FName/Arity,Bound,In,Out) :-
  lookup_fun_pars(FName/Arity,FPars),
  bind(FPars,In,IEnv),
  tr(Bound,cf(IEnv,apply(FName/Arity,FPars)),cf(FEnv,Out)).
```

The predicate `run` retrieves the formal parameters `FPars` of `FName/Arity` and creates an environment `IEnv` where those parameters are bound to the input values `In`. Then, it evaluates the application of `FName` to its parameters, thereby producing the final expression `Out`.

## 3 Conclusions and future work

We have presented a work in progress for the development of a CLP interpreter for detecting runtime errors of Erlang programs. An Erlang program is first translated into a set of Prolog terms, and then

the CLP interpreter is run on these terms together with symbolic input data. At present, our interpreter is able to deal with first-order sequential Erlang programs, but we think that the extension to higher-order functions can be achieved by following a similar approach. In the future, we also plan to consider concurrency with an appropriate technique for handling the state explosion problem. For instance, we may employ a partial order reduction technique [1] to obtain the minimal set of concurrent behaviours for a given program, and then generate the associated executions using our interpreter.

Let us briefly compare our work with the static analysis tools available for Erlang. Unlike Dyalizer [5], our tool computes answer constraints that describe type-related input patterns which lead to runtime errors. However, as already mentioned, due to the bounded symbolic execution, our interpreter may terminate with no answer, even if runtime errors are possible for concrete runs which exceed the given bound. One of the weaknesses of Dialyzer is that it is hard to know where typing errors come from. An extension of Dialyzer that provides an explanation for the cause of typing errors has been proposed to overcome this problem [6]. We believe that we are able to provide a similar information if we include debugging information in the clauses generated by our Erlang-to-CLP translation.

Unlike SOTER [3], which is based on abstract interpretation, our CLP interpreter provides full support to arithmetic operations through the use of constraint solvers. Moreover, the symbolic interpreter does not require any user intervention (except for the bound), while in SOTER the user is responsible for providing a suitable abstraction.

Besides being useful on its own for bounded verification, the CLP interpreter for Erlang may be the basis for more sophisticated analysis techniques. In particular, by following an approach developed in the case of imperative languages, we intend to apply CLP transformation techniques to specialize the interpreter with respect to a given Erlang program and its symbolic input [2]. The specialized CLP clauses may enable more efficient bounded verification, and they can also be used as input to other tools for analysis and verification (such as constraint-based analyzers and SMT solvers), which have already been shown to be effective in other contexts.

## References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (2014): *Optimal dynamic partial order reduction*. *ACM SIGPLAN Notices* 49(1), pp. 373–384.

[2] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti (2017): *Semantics-based generation of verification conditions via program specialization*. *Science of Computer Programming* 147, pp. 78–108.

[3] Emanuele D'Osualdo, Jonathan Kochems, and C-H Luke Ong (2013): *Automatic verification of Erlang-style concurrency*. In: *Static Analysis Symposium*, LNCS 7935, Springer, pp. 454–476.

[4] Joxan Jaffar and Michael Maher (1994): *Constraint Logic Programming: A Survey*. *Journal of Logic Programming* 19/20, pp. 503–581.

[5] Tobias Lindahl and Konstantinos Sagonas (2006): *Practical type inference based on success typings*. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM, pp. 167–178.

[6] Konstantinos Sagonas, Josep Silva, and Salvador Tamarit (2013): *Precise explanation of success typing errors*. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, ACM, pp. 33–42.

[7] Robert Virding, Claes Wikström, and Mike Williams (1996): *Concurrent Programming in ERLANG,* 2nd Ed. Prentice Hall International, Ltd., Hertfordshire, UK.