Metaprogramming and symbolic execution for detecting runtime errors in Erlang programs

Emanuele De Angelis\textsuperscript{1}, Fabio Fioravanti\textsuperscript{1}, Adrián Palacios\textsuperscript{2}, Alberto Pettorossi\textsuperscript{3} and Maurizio Proietti\textsuperscript{4}

\textsuperscript{1}University of Chieti-Pescara, Italy
\textsuperscript{2}Technical University of Valencia, Spain
\textsuperscript{3}Università di Roma ‘Tor Vergata’, Italy
\textsuperscript{4}CNR - IASI of Rome, Italy

CILC 2018
Bolzano, 21 settembre 2018
Erlang is a **dynamically typed** functional language supporting

- concurrency (based on asynchronous message-passing) and
- hot code loading

These features make it appropriate for **distributed, fault-tolerant** applications (Facebook, WhatsApp)

+ Dynamically typed languages allow **rapid development**

- Many errors are **not detected** until

  - the program is run on a **particular input**
  - a **particular interleaving** of processes is performed
Some tools mitigate these problems

- **Dialyzer**: Discrepancy AnalyZer for ERlang (included in the Erlang/OTP development environment)
- **PropEr**: PropErty-based testing tool for ERlang
- **CutER**: Concolic Unit Testing tool for ERlang

Our proposal:

**Bounded verifier** based on **Constraint Logic Programming (CLP)**

- Erlang programs automatically translated into CLP
- CLP interpreter to run them using symbolic inputs
We consider a first-order subset of Erlang and sequential programs.

A module is a set of function definitions

\[
\text{fun } (X_1, \ldots, X_n) \rightarrow expr \text{ end}
\]

The function body \textit{expr} includes

- literals (atoms, integers, float numbers)
- variables, list constructors, tuples
- match (=), case-of and try-catch expressions
- function applications
- calls to built-in functions (BIFs)
Example program

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

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

This code
- compiles without warnings
- crashes when the input is not a list (of numbers)

Our tool is able to
- list all potential runtime errors
- provide information about input types that cause them
The translation from Erlang to **Core Erlang** simplifies the program
- pattern matching in case-of expressions only
- explicit catch-all clauses in case-of expressions
- function applications with variables and literals only

The **CLP** encoding is *automatically* obtained from Core Erlang
Erlang-to-CLP translation: An example

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

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

fundef( lit(atom,'sum_list'), var('sum',1),
    fun([var('@c0')],
        case(var('@c0'),
        [
            clause( [lit(list,nil)], lit(atom,'true'), lit(int,0)),
            clause( [lit(list,nil)], lit(atom,'true') ,
                let([var('@c1')], apply(var('main',1),[var('T')]),
                    call(lit(atom,'erlang'),lit(atom,'+'),[var('H'),var('@c1')]))),
            clause( [var('@c2')], lit(atom,'true') ,
                primop(lit(atom,'match_fail'),
                    [tuple([lit(atom,'case_clause'),var('@c2')])])
            ]
        )
    )).

Emanuele De Angelis
Metaprogramming and symbolic execution for Erlang programs
7 / 14
The operational semantics is given in terms of a transition relation

\[ tr(Bound, cf(IEnv, IExp), cf(FEnv, FExp)) \]

between configurations of the form

\[ cf(\text{Environment}, \text{Expression}) \]

which defines how to get

- the final configuration \( cf(FEnv, FExp) \) from
- the initial configuration \( cf(IEnv, IExp) \) in
- \textbf{Bound} computation steps
Transition rules: An example

\[ \text{tr}(\text{Bound}, \text{cf}(\text{IEnv}, \text{IExp}), \text{cf}(\text{FEnv}, \text{FExp})) :\]
\[
\text{IExp} = \text{apply}(\text{FName}/\text{Arity}, \text{IExps}),
\text{lookup}_\text{error}_\text{flag}(\text{IEnv}, \text{false}),
\text{Bound}>0,
\text{Bound1 is Bound}-1,
\text{fun}(\text{FName}/\text{Arity}, \text{FPars}, \text{FBody}),
\text{tr}(\text{Bound1}, \text{cf}(\text{IEnv}, \text{tuple}(\text{IExps})),
\text{cf}(\text{EEnv}, \text{tuple}(\text{EExps}))),
\text{bind}(\text{FPars}, \text{EExps}, \text{AEnv}),
\text{lookup}_\text{error}_\text{flag}(\text{EEnv}, \text{F1}),
\text{update}_\text{error}_\text{flag}(\text{AEnv}, \text{F1}, \text{BEnv}),
\text{tr}(\text{Bound1}, \text{cf}(\text{BEnv}, \text{FBody}), \text{cf}(\text{CEnv}, \text{FExp})),
\text{lookup}_\text{error}_\text{flag}(\text{CEnv}, \text{F2}),
\text{update}_\text{error}_\text{flag}(\text{EEnv}, \text{F2}, \text{FEnv}). \]
Error detection with \textit{run/4}

The interpreter provides the predicate

\begin{verbatim}
run(FName/Arity,Bound,In,Out)
\end{verbatim}

whose execution evaluates the application of the function \texttt{FName} of arity \texttt{Arity} to the input arguments \texttt{In} in at most \texttt{Bound} steps. \texttt{Out} is the result of the function application.

If an error is found, then \texttt{Out} is bound to a term of the form

\begin{verbatim}
error(Reason)
\end{verbatim}

where \texttt{Reason} represents the error type:

\begin{itemize}
  \item \texttt{match fail}: evaluation of a match expression failed
  \item \texttt{badarithmetic}: bad argument in an arithmetic expression
\end{itemize}
Bounded verification of Erlang programs can be performed by executing a query of the form

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

- No answer: the program is **error-free up to Bound**
- 1+ answer(s): **error(s) detected**, each answer provides
  - the error type (the `Reason`)
  - the input that causes the error
  - some constraints on the computation that raises the error
By executing

?– run(sum/1,20,In,error(Reason)).

we obtain some answers (*error detected*)

In = [cons(lit(Type,_V),lit(list,nil))],
Reason = badarith,
dif(Type,int), dif(Type,float)

In = [L],
Reason = match_fail,
dif(L,cons(_Head,_Tail)), dif(L,lit(list,nil))
A generator for the program input(s), such as:

\[
\text{int_list}(N,L)
\]

can be used to generate lists of integers \( L \) of length \( N \)

\[
L = \text{cons}(\text{lit}(\text{int},N1),\text{cons}(\text{lit}(\text{int},N2),...))
\]

By using \( L \) to constrain the input of \( \text{sum}/1 \) in the query

\[
?- \text{int_list}(L,100), \text{run}(\text{sum}/1,100,L,\text{error}(\text{Reason})).
\]

we get no answer, meaning that \( \text{sum}/1 \) is error-free up-to 100
Conclusions & Future work

Bounded verifier for sequential Erlang programs:
- Translator from Core Erlang to CLP
- CLP Interpreter

Extend the CLP interpreter to
- support higher-order functions
- handle concurrent programs

Specialize the CLP interpreter to
- improve the efficiency of the verification process
- apply to the specialized interpreter other tools for analysis and verification (e.g., constraint-based analyzers or SMT solvers)

Thanks for your attention!
Conclusions & Future work

Bounded verifier for sequential Erlang programs:
- Translator from Core Erlang to CLP
- CLP Interpreter

Extend the CLP interpreter to
- support higher-order functions
- handle concurrent programs

Specialize the CLP interpreter to
- improve the efficiency of the verification process
- apply to the specialized interpreter other tools for analysis and verification (e.g., constraint-based analyzers or SMT solvers)

Thanks for your attention!