Metaprogramming and symbolic execution for detecting runtime errors in Erlang programs

Emanuele De Angelis[1], Fabio Fioravanti[1], Adrián Palacios[2], Alberto Pettorossi[3] and Maurizio Proietti[4]

[1]University of Chieti-Pescara, Italy
[2]Technical University of Valencia, Spain
[3]Università di Roma 'Tor Vergata', Italy
[4]CNR - IASI of Rome, Italy

CILC 2018

Bolzano, 21 settembre 2018

The Erlang language

Erlang is a dynamically typed functional language supporting

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

These features make it appropiate 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

Motivation

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

Erlang subset

We consider a first-order subset of Erlang and sequential programs

A module is a set of function definitions

$$\text{fun } (X_1, \ldots, X_n) \texttt{ -> } \textit{expr} \text{ end}$$

The function body *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

Bounded Verification for Erlang programs using CLP

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( [cons(var('H'),var('T'))], 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')])]))
      ]
    )
  )
).
```

CLP interpreter

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(Environment, Expression)
```

which defines how to get

- the final configuration `cf(FEnv,FExp)` from
- the initial configuration `cf(IEnv,IExp)` in
- `Bound` computation steps

Transition rules: An example

```
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(Bound1,cf(IEnv,tuple(IExps)),
             cf(EEnv,tuple(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).
```

Error detection with `run/4`

The interpreter provides the predicate

`run(FName/Arity,Bound,In,Out)`

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

If an error is found, then `Out` is bound to a term of the form

`error(Reason)`

where `Reason` represents the error type:

- `match_fail`: evaluation of a match expression failed
- `badarith`: bad argument in an arithmetic expression

Error detection with `run/4`

Bounded verification of Erlang programs can be performed by executing a query of the form

```
?- 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

Error detection with run/4: An example

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))
```

Error detection with `run/4`: An example

A generator for the program input(s), such as:

```
int_list(N,L)
```

can be used to generate lists of integers `L` of length `N`

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

By using `L` to constrain the input of `sum/1` in the query

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

we get no answer, meaning that `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!