Metaprogramming and symbolic execution for detecting runtime errors in Erlang programs

Adrián Palacios
(joint work with Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi and Maurizio Proietti)

Technical University of Valencia

HCVS 2018

July 13, 2018
Oxford, England

The Erlang language

Erlang is a programming language with

- integration of functional and concurrent features
- concurrency model based on asynchronous message-passing
- dynamic typing
- hot code loading

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

Motivation (types)

Dynamically typed languages allow rapid development

Many errors are not detected until the program is run (or even later)

- a particular input
- a particular interleaving

In static languages, some errors would be caught at compile time

Motivation (types)

Dynamically typed languages allow rapid development

Many errors are not detected until the program is run (or even later)

- a particular input
- a particular interleaving

In static languages, some errors would be caught at compile time

Motivation (tools)

In the context of Erlang, some tools mitigate these problems:

- **Dialyzer:** Popular tool for performing type inference
- **SOTER:** Model checking and abstract interpretation
- etc.

But these tools are

- not fully automatic in some cases
- only valid for one part of the language (sequential or concurrent)

Erlang

Erlang subset

We consider sequential programs written in a first-order subset of Erlang

In Erlang, a module is a sequence of function definitions

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

The function body *expr* includes

- literals (atoms, integers, float numbers)
- variables, list constructors, tuples
- let/case/try-catch expressions
- function applications and 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.
```

Note that this code

- compiles without warnings
- Dialyzer does not generate any warnings
- crashes when 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

Example program

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

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

Note that this code

- compiles without warnings
- Dialyzer does not generate any warnings
- crashes when 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

Example program

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

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

Note that this code

- compiles without warnings
- Dialyzer does not generate any warnings
- crashes when 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

Erlang-to-CLP translation

CLP terms are obtained from Core Erlang programs

Translating from Core Erlang has many benefits:

- Syntactic sugar has been removed
- Pattern matching performed only in case expressions
- Automatic insertion of catch-all clauses

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('main',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')])]))]))).
```

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('main',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')])]))]))).
```

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('main',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')])]))]))).
```

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('main',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')])]))]))).
```

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('main',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

We define a CLP interpreter in terms of `tr/3`

```
tr(Bound,cf(IEnv,IExp),cf(FEnv,FExp))
```

- `Bound`: The current depth bound
- `cf(IEnv,IExp)`: A source configuration (env. and expression)
- `cf(FEnv,FExp)`: A target configuration (env. and expression)

`tr/3 defines a transition between source and target confs.`

CLP interpreter

We define a CLP interpreter in terms of `tr/3`

```
tr(Bound,cf(IEnv,IExp),cf(FEnv,FExp))
```

- `Bound`: The current depth bound
- `cf(IEnv,IExp)`: A source configuration (env. and expression)
- `cf(FEnv,FExp)`: A target configuration (env. and expression)

`tr/3 defines a transition between source and target confs.`

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_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).
```

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_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).
```

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_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).
```

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_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).
```

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_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).
```

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_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).
```

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_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).
```

The `run/4` predicate

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

- `FName/Arity`: The function to be computed
- `Bound`: The bound depth to be explored
- `In`: The input parameters
- `Out`: The result value

Error detection with `run/4`

If an error is found, `Out` is bound to a term `error(Err)`

where Err represent the error type:

- `match_fail`: A pattern matching error
- `badarith`: Arithmetic function called with non-arithmetic args
- etc.

Error detection with `run/4`

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

When we run this query...

- No answers: Program is error-free up to Bound
- 1+ answers: Error detected, information about
  - error type
  - input type
  - constraints

Error detection with `run/4`

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

When we run this query...

- No answers: Program is error-free up to Bound
- 1+ answers: Error detected, information about
  - error type
  - input type
  - constraints

Error detection with run/4: An example

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

We obtain some answers (error detected)

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

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

Error detection with run/4: An example

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

We obtain some answers (error detected)

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

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

Error detection with run/4: An example

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

We obtain some answers (error detected)

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

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

Error detection with run/4: An example

Let us introduce `int_list/2` to generate a list of integers:

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

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

Reexecute `run/4` using `L` as input:

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

Result: 0 answers (error-free program)

Error detection with run/4: An example

Let us introduce `int_list/2` to generate a list of integers:

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

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

Reexecute `run/4` using `L` as input:

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

Result: 0 answers (error-free program)

Compared to Dialyzer

Similar

Dialyzer: Type inference based on success typings
Our tool: Type-related information on input values (depends on bound)

Different

Dialyzer: Difficult to know where errors come from
Our tool: Might provide this information if we include debugging info

Compared to Dialyzer

Similar

Dialyzer: Type inference based on success typings
Our tool: Type-related information on input values (depends on bound)

Different

Dialyzer: Difficult to know where errors come from
Our tool: Might provide this information if we include debugging info

Compared to SOTER

Different

SOTER:

- Targets concurrent Erlang
- Based on abstract interpretation
- User provides abstractions

Our tool:

- Targets sequential Erlang (for now)
- Support for arithmetics operations using constraint solvers
- No user intervention is required

Conclusions

Conclusions

We have presented our work on a CLP interpreter for Erlang:

1. Erlang programs are translated to CLP terms
2. The CLP interpreter can run these programs on symbolic inputs
3. Error detection up to some bound can be performed

This way, we can perform bounded verification for Erlang programs

Future work

Extend the CLP interpreter to

- support higher-order constructs
- handle concurrent programs

We plan to apply specialization on the CLP interpreter
(given an Erlang program and its symbolic input)

- May enable more efficient computation
- Can be used as input to other tools for analysis and verification (e.g., constraint-based analyzers or SMT solvers)

Thanks for your attention!