A CutEr Tool

Kostis Sagonas
Overview

- Testing
  - demo: unit, property-based, and concolic testing in Erlang
- Concolic execution for Erlang
  - demo
- Support for type specifications
  - short demo
- CutEr: A Concolic Unit Testing tool for Erlang
- A “real” experience from using CutEr
  - short demo
- Concluding remarks & future work
Testing

- Testing is important
- **Unit testing** is the most widely used method
  - **Tools:** xUnit, EUnit
- Functional languages have mainly explored variants of **property-based random testing**
  - **Tools:** Haskell QuickCheck, EQC, PropEr, Triq, ...

A CutEr Tool @ Erlang Factory 2016 - https://github.com/aggelgian/cuter
First demo!
A first example

• A program unit:

```erlang
classify(L) ->
    case lists:sum(L) of
        S when S < 0 -> negative;
        S when S < 4711 -> small;
        S when S > 4711 -> big;
        _ -> erlang:error(badmatch)
    end.
```

A function that classifies a list of numbers

• In general, pattern matching in Erlang provides a powerful mechanism for program assertions

\[
[42, x, x | _] = f(\ldots)
\]
In imperative languages, researchers have argued for the benefits of **concolic testing**.

- Fully automatic testing approach
- **Concolic** = **Concrete** + **Symbolic**
- Aims to achieve high path coverage

Tools: DART, CUTE, Symbolic Java PathFinder, jCUTE, SAGE, ...
Concolic execution

• Also known as dynamic symbolic execution

• Main idea:
  – during concrete execution, collect symbolic constraints on program inputs that cause the program to follow a specific execution path and
  – use these constraints to force execution of other paths

• Properties/advantages:
  – concrete execution makes available accurate information about program state which may not be easily accessible when using e.g. random testing or static analysis techniques
Implementation of concolic execution

- Symbolic execution is enabled by instrumenting the program with code that collects *path constraints* without disrupting its concrete execution
- Each variable that depends on input has both a concrete and a symbolic value associated to it
- Path constraints are expressed in an appropriate logic
- Off-the-shelf constraint solvers, often SMT ones, are used to solve these constraints and generate new inputs that will steer the future test runs to explore unexplored paths
- The execution paths can be expressed as a *symbolic execution tree*
  - each leaf node has a path constraint describing the input values that force the program to follow that specific path
A second example

-module(sf2).
-export([foo/1]).

foo(L) ->
    lists:foreach(fun fcmp/1, L).

fcmp(X) ->
    case cmp(X) of
        gt -> ok;
        lt -> ok
    end.

cmp(X) when X > 42 -> gt;
cmp(42) -> eq;
cmp(X) when X < 42 -> lt.
Second demo!
module example [foo/1] =
    foo/1 = fun (_cor0) ->
        call lists:foreach (fcmp/1, _cor0)
    end

fcmp/1 = fun (_cor0) ->
    case <apply cmp/1 (_cor0)> of
        <gt> when true -> ok
        <lt> when true -> ok
        <_cor1> when true -> FAIL
    end

cmp/1 = fun (_cor0) ->
    case <_cor0> of
        <X> when call erlang:'>' (_cor0, 42) -> gt
        <42> when true -> eq
        <X> when call erlang:'<' (_cor0, 42) -> lt
        <_cor1> when true -> FAIL
    end
Control flow graphs of functions

```erlang
module example [foo/1] =
  foo/1 = fun (_cor0) ->
    call lists:foreach (fcmp/1, _cor0)
  end

fcmp/1 = fun (_cor0) ->
case <apply cmp/1 (_cor0)> of
  <gt> when true -> ok
  <lt> when true -> ok
  <cor1> when true -> FAIL
end

cmp/1 = fun (_cor0) ->
case <cor0> of
  <x> when call erlang:’>’ (_cor0, 42) -> gt
  <42> when true -> eq
  <x> when call erlang:’<’ (_cor0, 42) -> lt
  <cor1> when true -> FAIL
end
```

https://github.com/aggelgian/cuter
Control flow graphs of functions

module lists [..., foreach/2, ...] = ...

foreach/2 = fun (_cor1,_cor0) ->
    case <_cor1,_cor0> of
        <F,[H|T]> when true ->
            do apply F (H)
                apply foreach/2 (F, T)
        <F,[]> when call erlang:is_function (_cor0, 1) -> ok
        <_cor3,_cor2> when true -> FAIL
    end

...
L ↔ [17]; L

L = [H|T]

L = [H|T] \land L = []

X ↔ 17; hd(L)

X > 42

L = [H|T] \land \neg hd(L) > 42

X = 42

L = [H|T] \land \neg hd(L) > 42 \land hd(L) = 42

X < 42

X > 42 \land X = 42 \land X < 42

Y ↔ !t

Y = gt

42.0

Y ↔ !t

Y = !t

L ↔ []; tl(L)

L = [H|T]

L = [] and

is_function(F, 1)

F ↔ fcmp/1

T@7

ok
Search strategy

• Which decision node to reverse?

• We use two metrics:
  – If a decision node exists whose reversed (red) label has not yet been visited, reverse it
  – Else reverse the decision node which is closer to the root

• Stop when there are no decision nodes left to reverse
Depth-bounded search

• **Depth** counts `case` constructs that precede the decision node

• All constraints related to patterns and guards of a specific `case` construct are considered to be at the same depth

• Prune decision nodes whose depth exceeds a threshold
Support for type specifications

- Type specifications impose additional constraints on program inputs

- For the first demo program:

  ```erlang
  -type ret() :: 'negative' | 'small' | 'big'.
  -spec classify([number()]) -> ret().
  ```

- For the second demo program:

  ```erlang
  -spec foreach(fun((T) -> term()), [T]) -> ok.

  -spec foo([term()]) -> ok.
  -spec foo([integer()]) -> ok.
  ```
-spec foo([term()]) -> ok.

\[ is\_list(L) \land L = [H|T] \land \text{hd}(L) > 42 \]

\[ is\_list(L) \land L = [H|T] \land \text{hd}(L) > 42 \land \text{hd}(L) = 42 \]

\[ is\_list(L) \land L = [] \]

\[ is\_list(L) \land X = \text{hd}(L) \land X > 42 \land X = 42 \land X < 42 \]

-spec foo([integer()]) -> ok.

\[ is\_integer\_list(L) \implies L = [] \lor is\_integer(\text{hd}(L)) \land is\_integer\_list(\text{tl}(L)) \]

\[ X = \text{hd}(L) \land is\_integer(X) \land X > 42 \land X = 42 \land X < 42 \]
Third demo!
The first example with some twists

• A program unit:

```erlang
classify(L) when length(L) < 4 -> tiny;
classify(L) ->
    case lists:foldl(fun erlang:'+'/2, 0, L) of
        S when S < 0 -> negative;
        S when S < 4711 -> small;
        S when S > 4711 -> big
    end.
```

A function that classifies a list of numbers
One more twist

• A program unit:

```erlang
classify(_, L) when length(L) < 4 -> tiny;
classify(F, L) ->
    case lists:foldl(F, 0, L) of
        S when S < 0 -> negative;
        S when S < 4711 -> small;
        S when S > 4711 -> big
    end.
```

A function that classifies a list of numbers
CutEr: Concolic Unit Testing for Erlang

Concolic execution component

Supervisor

Code Server

Monitor

Interpreter

Interpreter

Interpreter

Path exploration component

Scheduler

Constraint solving component

Solver

External SMT solver (Z3)
SMT solving with Z3

• Define the most general type, i.e. `term()`

```erlang
Term, TList, IList = Datatypes('Term, TList, IList')

Term.declare('int', ('ival', IntSort()))
Term.declare('real', ('rval', RealSort()))
Term.declare('atm', ('aval', IList))
Term.declare('lst', ('lval', TList))
Term.declare('tpl', ('tval', TList))

TList.declare('nil')
TList.declare('cons', ('hd', Term), ('tl', TList))

IList.declare('anil')
IList.declare('acons', ('ahd', IntSort()), ('atl', IList))
```
Encoding values in Z3

• For example, for the terms:
  – 42
  – [17, 42]
  – {42, ok}

```plaintext
t1 = Term.int(42)
t2 = Term.lst(TList.cons(Term.int(17),
    TList.cons(Term.int(42), TList.nil)))
t3 = Term.tpl(TList.cons(Term.int(42),
    TList.cons(Term.atm(
        IList.acons(111, IList.acons(107, IList.anil))
    ), TList.nil)))
```
Encoding axioms in Z3

• For example, the path constraint:

\[ \neg (L = [H|T]) \land \neg (L = []) \]

\begin{verbatim}
Not(And(Term.is_lst(L), TList.is_cons(Term.lval(L)))))
Not(And(Term.is_lst(L), TList.is_nil(Term.lval(L)))))
\end{verbatim}
CutEr

• Available on GitHub:
  
  https://github.com/aggelgian/cuter

• Requires Erlang/OTP 17.x or 18.x

Current known limitations:

• Does not support maps (yet!)

• Support for recursive types is still incomplete
A bigger unit to test

• A post in the erlang-bugs mailing list:

  http://erlang.org/pipermail/erlang-bugs/2015-May/004944.html

• Module `otp_internal` from Erlang/OTP 18.0-rc1

Loïc Hoguin <essen@ninienes.eu>

Sat May 2 17:11:30 CEST 2015

- Previous message: [erlang-bugs] FreeBSD FPE issue on ERTS_FP_CHECK_INIT Re: ERTS_FP_CHECK_INIT error of HiPE in 18.0-rc1 running on FreeBSD 10.1-STABLE
- Next message: [erlang-bugs] Crash on compile with deprecated functions (18-rc1)
- Messages sorted by: [ date ] [ thread ] [ subject ] [ author ]

Hello,

Some of my applications don't compile anymore because they have the ssl:negotiated_next_protocol instead of negotiated_protocol.

Problem is there is a crash instead of a nice error:

src/gun.erl: internal error in lint_module;
crash reason: {case_clause, {deprecated, {ssl, negotiated_protocol}}}

  in function  otp_internal:obsolete/3 (otp_internal.erl, line 33)
in call from erl_lint:deprecated_function/5 (erl_lint.erl, line 3551)
in call from erl_lint:check_remote_function/5 (erl_lint.erl, line 3527)
in call from erl_lint:expr/3 (erl_lint.erl, line 2166)
in call from erl_lint:expr/3 (erl_lint.erl, line 2111)
in call from erl_lint:expr/3 (erl_lint.erl, line 2250)
in call from erl_lint:exprs/3 (erl_lint.erl, line 2044)
in call from erl_lint:icrt_clause/3 (erl_lint.erl, line 3029)
Demo!
Concluding remarks

- This presentation:
  - Concolic testing for the “functional” subset of Erlang
  - CutEr: a tool that implements this approach

https://github.com/aggelgian/cuter

- Future Work
  - Better search strategies
  - Experiment with more SMT solvers
  - Handle concurrency
Thanks!