Concolic Execution in Functional Programming by Program Instrumentation

Adrián Palacios

(joint work with Germán Vidal)

Technical University of Valencia

25th Int'l Symposium on Logic-Based Program Synthesis and Transformation

July 14, 2015

Siena, Italy

Test-case generation (imperative programming)

Approaches for TC generation:

- Random input data:
  - Extended use.
  - Poor coverage in general.
- Symbolic execution:
  - Build a search tree with symbolic data.
  - Solve constraints in leaves to produce test cases.
  - Complex constraints should be simplified.
- Concolic execution:
  - Compute a symbolic execution that mimics the concrete execution: collect constraints $c_1, c_2, \ldots, c_n$
  - Solve $\neg c_n$ and produce new input data
  - Push values from concrete execution when constraints are too complex.

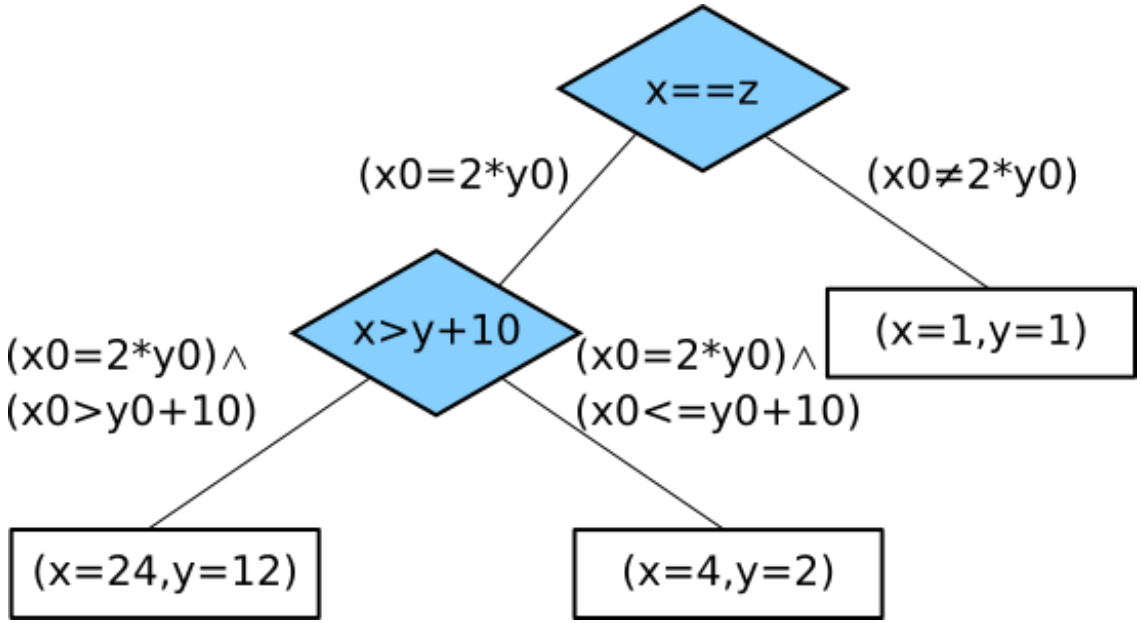
Test-case generation (imperative programming)

Approaches for TC generation:

- Random input data:
  - Extended use.
  - Poor coverage in general.
- Symbolic execution:
  - Build a search tree with symbolic data.
  - Solve constraints in leaves to produce test cases.
  - Complex constraints should be simplified.
- Concolic execution:
  - Compute a symbolic execution that mimics the concrete execution: collect constraints $c_1, c_2, \ldots, c_n$
  - Solve $\neg c_n$ and produce new input data
  - Push values from concrete execution when constraints are too complex.

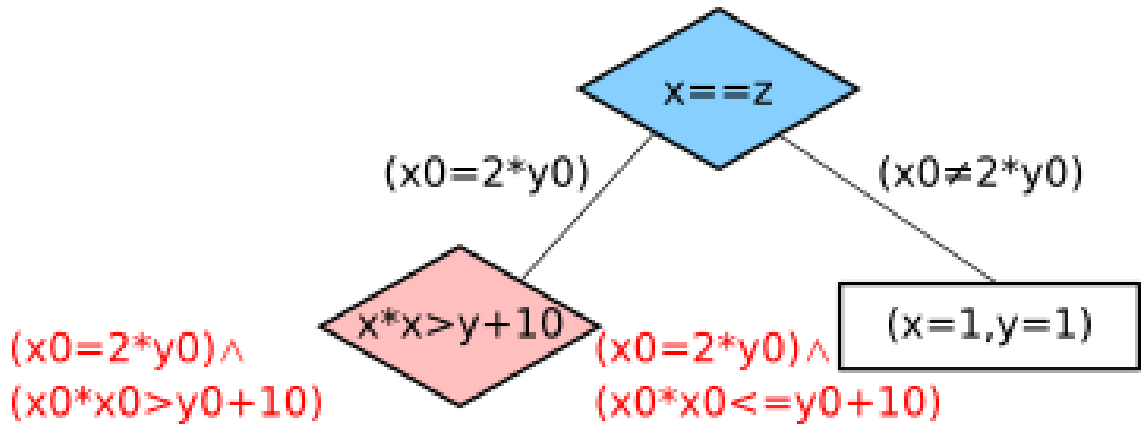
Symbolic execution: simple example

```
void foo(int x, int y){
    z = 2 * y;
    if(x == z){
        if(x > y + 10){
            error;
        }
    }
}
```



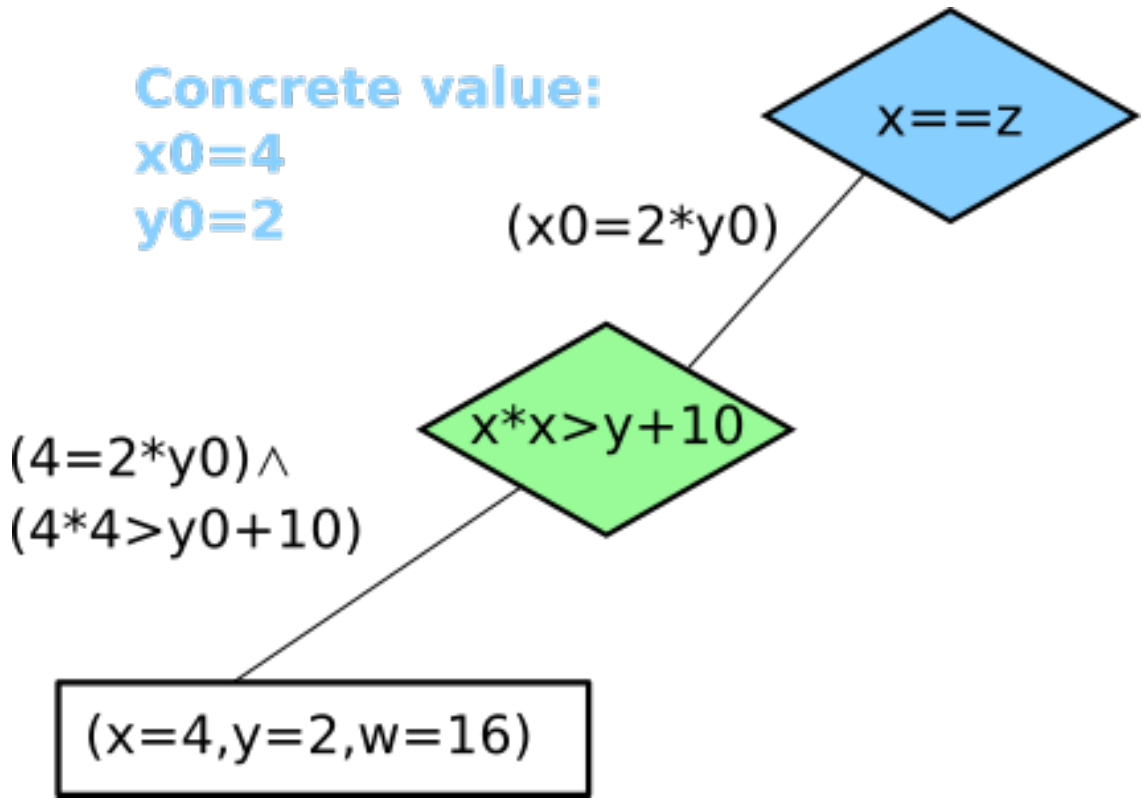
Symbolic execution: Complex example

```
void foo(int x, int y){
    z = 2 * y;
    if(x == z){
        if(x * x > y + 10){
            error;
        }
    }
}
```



Concolic execution: Example

```
void foo(int x, int y){
    z = 2 * y;
    if(x == z){
        if(x * x > y + 10){
            error;
        }
    }
}
```



Concolic execution (functional programming)

Few approaches to concolic testing for functional programming:

- Preliminary approach to symbolic execution in Erlang [PSI'14]
- CutEr, a new concolic testing tool for Erlang [PPDP'15]

Some approaches make use of an augmented interpreter to also deal with symbolic values. This has some drawbacks:

- There is a huge implementation effort.
- It is difficult to maintain.
- It does not scale up well.

We propose a novel approach based on instrumenting an (Erlang) program.

The language

Erlang (main features)

Main features of Erlang:

- Integration of functional and concurrent features.
- Concurrency model based on message-passing
- Dynamic typing.
- Hot code loading.

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

Because of its growing popularity, powerful testing and verification techniques are required.

Erlang syntax

An Erlang program is a set of function definitions, with the form:

$$f(X_1, \ldots, X_n) \rightarrow s.$$

where the sentence s can be

- an expression e (made of vars, atoms, functions, ...)
- a sequence of sentences s_1, s_2
- a case statement case e of $pat_1 \rightarrow s_1; \ldots; pat_n \rightarrow s_n$ end
- pattern matching $pat = e$
- ...

Erlang code is translated to Core Erlang, an intermediate language used by the Erlang compiler. This language is appropiate for defining analysis and transformation techniques.

From Erlang to Core Erlang

```
f(X, Y)  →  g(X),
            case h(X) of
              a → A = h(Y),
                  g(A);
              b → g(h([]))
            end.
```

```
f/2 =
fun (X, Y) → do  apply g/1 (X),
                 case apply h/1 (X) of
                   a → let Z = apply h/1
                           in apply g/1 (Z);
                   b → let V = apply h/1
                           in apply g/1 (V);
                   W → fail
                 end.
```

Flat language

For our instrumentation to be correct, we need to make explicit the return values from expressions. Thus, we require the following to be patterns:

- The name and arguments of a function application.
- The return value of a function.
- The argument and return value of a case expression.

$$
\begin{array}{rcl}
\mathit{pgm} & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow \mathsf{let}\ X = e\ \mathsf{in}\ X. \mid \mathit{pgm}\ \mathit{pgm} \\
\mathrm{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [p_1|p_2] \mid \{p_1, \ldots, p_n\} \\
& \mid & \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
& \mid & \mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e \\
& \mid & \mathsf{let}\ p_1 = \mathsf{case}\ p_2\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \\
\mathit{clauses} & ::= & p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \\
\mathrm{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1, \ldots, p_n\} \mid \mathrm{a} \mid X \\
\mathrm{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1, \ldots, v_n\} \mid \mathrm{a}
\end{array}
$$

Flat language

For our instrumentation to be correct, we need to make explicit the return values from expressions. Thus, we require the following to be patterns:

- The name and arguments of a function application.
- The return value of a function.
- The argument and return value of a case expression.

$$
\begin{array}{rcl}
\mathit{pgm} & ::= & \mathrm{a}/n = \text{fun } (X_1, \ldots, X_n) \rightarrow \text{let } X = e \text{ in } X. \mid \mathit{pgm}\ \mathit{pgm} \\
\text{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [p_1|p_2] \mid \{p_1, \ldots, p_n\} \\
& \mid & \text{let } p = e_1 \text{ in } e_2 \mid \text{do } e_1\ e_2 \\
& \mid & \text{let } p = \text{apply } p_0\ (p_1, \ldots, p_n) \text{ in } e \\
& \mid & \text{let } p_1 = \text{case } p_2 \text{ of } \mathit{clauses} \text{ end in } e \\
\mathit{clauses} & ::= & p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \\
\text{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1, \ldots, p_n\} \mid \mathrm{a} \mid X \\
\text{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1, \ldots, v_n\} \mid \mathrm{a}
\end{array}
$$

Instrumented Semantics

Events

Five types of events will be enough to reconstruct the symbolic execution:

- call($params, vars, p, [p_1, \ldots, p_n]$)
- exit($params, vars, p$)
- bind($params, vars, p, p'$)
- case($params, vars, i, p_0, p_i, [(p_0, 1, p_1), \ldots, (p_0, n, p_n)]$)
- exitcase($params, vars, p, p'$)

These events will give us static information about the execution of the program.

Instrumented semantics (notation)

Statements have the form:

$$\pi, \theta \vdash e \Downarrow_{\tau} p$$

where:

- π is the context.
- θ is the environment.
- e is an expression.
- τ is a sequence of events.
- p is a pattern.

Instrumented semantics for 'apply'

$$\frac{\begin{array}{c}\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_{\epsilon} \mathrm{f}/m \quad \ldots \quad \langle vs, ps\rangle, \theta \vdash p_m \Downarrow_{\epsilon} p'_m \\ \langle [\overline{Y_m}], [\mathrm{bv}(e_2)]\rangle, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_1} p' \qquad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p''\end{array}}{\langle vs, ps\rangle, \theta \vdash \mathrm{let}\ p = \mathrm{apply}\ p_0\ (\overline{p_m})\ \mathrm{in}\ e \Downarrow_{\mathrm{call}(vs,ps,p,[\overline{p_m}])+\tau_1+\mathrm{exit}([\overline{Y_m}],[\mathrm{bv}(e_2)],p''_2)+\tau_2} p''}$$

$$\mathrm{if}\ \mathrm{f}/m = \mathrm{fun}\ (\overline{Y_m}) \to e_2 \in \mathrm{pgm},\ \mathrm{ret}(e_2) = p''_2,$$
$$\mathrm{match}(\overline{Y_m}, \overline{p'_m}) = \sigma,\ \mathrm{match}(p, p') = \sigma'$$

Example program

$$\begin{array}{rcll} \mathrm{main}/1 & = & \mathsf{fun}\ (X) \rightarrow & \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X, X)\ \mathsf{in}\ W \\ \mathrm{app}/2 & = & \mathsf{fun}\ (X, Y) \rightarrow & \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of} \\ & & & \quad [\,] \rightarrow Y \\ & & & \quad [H|T] \rightarrow \mathsf{let}\ W_2 = \mathsf{apply}\ \mathrm{app}/2\ (T, Y)\ \mathsf{in}\ [H|W_2] \\ & & & \quad \mathsf{end} \\ & & & \mathsf{in}\ W_1 \end{array}$$

Example computation with input [a]:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_2, \sigma_4 \vdash Y \Downarrow_{\epsilon} [\mathrm{a}] \qquad \pi_2, \sigma_5 \vdash W_1 \Downarrow_{\epsilon} [\mathrm{a}]}{\pi_2, \sigma_4 \vdash \mathsf{let}\ W_1 = \mathsf{case} \ldots \Downarrow_{\tau_1} [\mathrm{a}] \qquad \pi_2, \sigma_6 \vdash [H|W_2] \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]} \\ \pi_2, \sigma_3 \vdash \mathsf{let}\ W_2 = \mathsf{apply} \ldots \Downarrow_{\tau_2} [\mathrm{a}, \mathrm{a}] \qquad \pi_2, \sigma_7 \vdash W_1 \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]} \\ \pi_2, \sigma_2 \vdash \mathsf{let}\ W_1 = \mathsf{case} \ldots \Downarrow_{\tau_3} [\mathrm{a}, \mathrm{a}] \qquad \pi_1, \sigma_8 \vdash W \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]}{\pi_1, \sigma_1 \vdash \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X, X)\ \mathsf{in}\ W \Downarrow_{\tau_4} [\mathrm{a}, \mathrm{a}]}$$

Associated sequence of events

$$\begin{array}{l}
\mathsf{call}([X],[W],W,[X,X]) \\
\mathsf{case}([X,Y],[W_1,W_2],2,X,[H|T],[(1,X,[\,]),(2,X,[H|T])]) \\
\mathsf{call}([X,Y],[W_1,W_2],W_2,[T,Y]) \\
\mathsf{case}([X,Y],[W_1,W_2],1,X,[\,],[(1,X,[\,]),(2,X,[H|T])]) \\
\mathsf{exitcase}([X,Y],[W_1,W_2],W_1,Y) \\
\mathsf{exit}([X,Y],[W_1,W_2],W_1) \\
\mathsf{exitcase}([X,Y],[W_1,W_2],W_1,[H|W_2]) \\
\mathsf{exit}([X,Y],[W_1,W_2],W_1) \\
\mathsf{exit}([X],[W],W)
\end{array}$$

The computed sequence of **static events** allows us to reconstruct a symbolic execution that follows the steps of the concrete execution that generated the trace.

Program Instrumentation

Transformation

We instrument the program by replacing each function definition:

$$\mathrm{f}/k = \text{fun } (X_1, \ldots, X_k) \rightarrow \text{let } X = e \text{ in } X$$

with a new function definition of the form:

$$\mathrm{f}/k = \text{fun } (X_1, \ldots, X_k) \rightarrow [\![\text{let } X = e \text{ in out}(\text{“bind}(vs, bs, X, \text{ret}(e))\text{”}, \text{out}(\text{“exit}(vs, bs, X)\text{”}, X))]\!]_{\mathsf{F}}^{vs,bs}$$

Notice that:

- Predefined function **out/2** outputs its first argument and returns its second argument.
- We propagate values $vs = [\overline{X_k}]$ and $bs = [\text{bv}(e)]$.
- We also propagate a flag that determines if an exitcase event should be generated.

Program instrumentation for 'apply'

$$[\![\text{let } W = \text{apply } p_0\ (\overline{p_n}) \text{ in } e]\!]_b^{vs,bs} = \text{let } W = \text{out}(\text{“call}(vs, bs, W, [p_1, \ldots, p_n])\text{”}, \text{apply } p/0\ (p_1, \ldots, p_n)) \text{ in } [\![e]\!]_b^{vs,bs}$$

Instrumented program

$$\begin{array}{l}
\mathrm{main}/2 \;=\; \mathsf{fun}\ (X) \to \quad \mathsf{let}\ W = \ \mathrm{out}(\text{“call}([X],[W],W,[X,X])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{apply}\ \mathrm{app}/2\ (X,X))\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathrm{out}(\text{“exit}([X],[W],W)\text{”},W)
\end{array}$$

$$\begin{array}{l}
\mathrm{app}/2 \;=\; \mathsf{fun}\ (X,Y) \to\\
\quad \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of}\\
\qquad\qquad [\,] \to\ \mathrm{out}(\text{“case}([X,Y],[W_1,W_2,H,T],1,X,[\,],\mathit{alts})\text{”},\\
\qquad\qquad\qquad\qquad \mathrm{out}(\text{“exitcase}([X,Y],[W_1,W_2,H,T],W_1,Y)\text{”},Y))\\
\qquad\qquad [H|T] \to\ \mathrm{out}(\text{“case}([X,Y],[W_1,W_2,H,T],2,X,[H|T],\mathit{alts})\text{”},\\
\qquad\qquad\qquad\qquad \mathsf{let}\ W_2 = \ \mathrm{out}(\text{“call}([X,Y],[W_1,W_2,H,T],W_2,[T,Y])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{apply}\ \mathrm{app}/2\ (T,Y)))\\
\qquad\qquad\qquad\qquad \mathsf{in}\ \mathrm{out}(\text{“exitcase}([X,Y],[W_1,W_2,H,T],W_1,[H|W_2])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\quad [H|W_2])\\
\quad \mathsf{in}\ \mathrm{out}(\text{“exit}([X,Y],[W_1,W_2,H,T],W_1)\text{”},W_1)
\end{array}$$

where $\mathit{alts} = [(1,X,[]),(2,X,[H|T])]$.

The execution of this program **should correspond to** the one using the instrumented semantics previously shown.

Conclusions

Conclusions and future work

Our paper is the first approach to concolic execution by program instrumentation for functional programming.

This approach is easier to maintain and scales up better (execution is done using the standard environment).

In the near future, we will:

- Develop a tool for concolic testing.
- Design heuristics for this algorithm.
- Improve implementation to make it fully automatic.
- Handle concurrency.

Thanks for your attention!