Concolic Testing of Functional Logic Programs

Jan Rasmus Tikovsky
jrt@informatik.uni-kiel.de

Kiel University

WFLP 2017

Program Testing

Check correctness of programs via testing

⇒ But writing test cases manually is time consuming and tedious...

Program Testing

Check correctness of programs via testing
⇒ But writing test cases manually is time consuming and tedious...

(Automated) Testing Tools and Libraries:

- Random testing
- Property-based testing
- Symbolic Execution
- Concolic Execution

Concolic Execution

Goal: Find enough test cases for full program coverage

Basic Idea: Concrete Execution drives Symbolic Execution

- Execute program with concrete input data
- Collect symbolic information during concrete execution
- Information corresponds to constraints along *one* execution path
- Negate and solve constraints systematically to search for *alternative* execution paths

Concolic Execution Example

```
nthElem []       _          = Nothing
nthElem (x : xs) n | n == 0 = Just x
                   | n >  0 = nthElem xs (n - 1)
```

Example (Initial Call: `nthElem [False] 0`)

Concrete Execution:

- selects second program rule
- yields the result `Just False`

Symbolic Execution:

- starts with symbolic function call `nthElem` ys_s n_s
- constrains them whenever a branch decision is made

Path Constraints:

- $ys_s = x_s : xs_s \wedge n_s = 0$
- Compute input data for alternative execution paths (`nthElem [] 0`)

Curry - A Functional Logic Programming Language

- Haskell-like syntax
- Higher order functions, non-strict semantics, lazy evaluation
- Non-determinism, free variables

Curry

```
not True  = False
not False = True

map _ []       = []
map f (x : xs) = f x : map f xs

insertND x []       = [x]
insertND x (y : ys) = x : y : ys
insertND x (y : ys) = y : insertND x ys
```

Example Calls

```
> map not [True, False]
[False, True]

> insertND 42 [1, 2]
[42, 1, 2]
[1, 42, 2]
[1, 2, 42]

> not x where x free
{x = False} True
{x = True} False
```

Concolic Execution Scheme of Curry

Curry Concolic Testing Interpreter (*ccti*)

Input: Program P with initial call of the function to be tested e

Output: Set of test cases, i.e. input data and corresponding results

Translation of Curry to FlatCurry

Curry

```
nthElem []       _          = Nothing
nthElem (x : xs) n | n == 0 = Just x
                   | n >  0 = nthElem xs (n - 1)

insertND x []       = [x]
insertND x (y : ys) = x : y : ys
insertND x (y : ys) = y : insertND x ys
```

FlatCurry

```
nthElem xs n = case  xs of
  []   -> Nothing
  y:ys -> case  n == 0 of
    True  -> Just y
    False -> case  n > 0 of True  -> nthElem ys (n-1)
                            False -> failed

insertND x xs = case  xs of
  []   -> [x]
  y:ys -> (x : y : ys) ? (y : insertND x ys)
```

Translation of Curry to FlatCurry

Curry

```
nthElem []         _            = Nothing
nthElem (x : xs) n | n == 0 = Just x
                   | n >  0 = nthElem xs (n - 1)

insertND x []       = [x]
insertND x (y : ys) = x : y : ys
insertND x (y : ys) = y : insertND x ys
```

FlatCurry

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed

insertND x xs = case4 xs of
  []   -> [x]
  y:ys -> (x : y : ys) ? (y : insertND x ys)
```

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

```
nthElem [False] 0
```

Symbolic Trace

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

`nthElem [False] 0`

Symbolic Trace

`[]`

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []    -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

`nthElem [False] 0`

$\Rightarrow^*$ `case1 [False] of ...`

Symbolic Trace

`[]`

`[]`

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

`nthElem [False] 0`

$\Rightarrow^*$ `case1 [False] of ...`

$\Rightarrow^*$ `case2 0 == 0 of ...`

Symbolic Trace

`[]`

`[]`

[(case1, 2/2, xs_s, (:))]

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

nthElem [False] 0

$\Rightarrow^*$ case1 [False] of ...

$\Rightarrow^*$ case2 0 == 0 of ...

$\Rightarrow^*$ Just False

Symbolic Trace

[]

[]

[(case1, 2/2, xs_s, (:))]

[(case1, 2/2, xs_s, (:)), (case2, 1/2, n_s == 0)]

Trace Symbolic Information in Deterministic Programs

```
nthElem xs n = case1 xs of
  []   -> Nothing
  y:ys -> case2 n == 0 of
    True  -> Just y
    False -> case3 n > 0 of True  -> nthElem ys (n-1)
                            False -> failed
```

Concrete Execution

nthElem [False] 0

$\Rightarrow^*$ case1 [False] of ...

$\Rightarrow^*$ case2 0 == 0 of ...

$\Rightarrow^*$ Just False

Symbolic Trace

[]

[]

[(case1, 2/2, xs_s, (:))]

[(case1, 2/2, xs_s, (:)), (case2, 1/2, n_s == 0)]

- Program branches: case expressions and non-det. choices
- Trace symbolic information: case ids, branch numbers, symbolic variables, matching constructors
- Comparison operations on numerical literals: associated constraint

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
```

```
[]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
```

```
[]
[]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []    -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True :  False :  []) ?  (False :  insertND True [])
```

```
[]
[]
[(case4, 2/2, xs_s, (:))]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True : False : []) ? (False : insertND True [])
```

```
[]
[]
[(case4, 2/2, xs_s, (:))] ? [(case4, 2/2, xs_s, (:))]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

insertND True [False]
$\Rightarrow^*$ case4 [False] of ...
$\Rightarrow^*$ (True : False : []) ? (False : insertND True [])
$\Rightarrow^*$ {[True, False]}

[]
[]
[(case4, 2/2, xs_s, (:))] ? [(case4, 2/2, xs_s, (:))]
{[(case4, 2/2, xs_s, (:))]}

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True :  False :  []) ?  (False :  insertND True [])
⇒* {[True, False]} ∪ (False :  case4 [] of ...)
```

```
[]
[]
[(case4, 2/2, xs_s, (:))]  ?  [(case4, 2/2, xs_s, (:))]
{[(case4, 2/2, xs_s, (:))]} ∪ [(case4, 2/2, xs_s, (:))]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True :  False :  []) ?  (False :  insertND True [])
⇒* {[True, False]} ∪ (False :  case4 [] of ...)
⇒* {[True, False]} ∪ {[False, True]}
```

```
[]
[]
[(case4, 2/2, xs_s, (:))]  ?  [(case4, 2/2, xs_s, (:))]
{[(case4, 2/2, xs_s, (:))]} ∪ [(case4, 2/2, xs_s, (:))]
{[(case4,2/2,xs_s,(:))]} ∪ [(case4,2/2,xs_s,(:)), (case4,1/2,ys_s,[])]
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True :  False :  []) ?  (False :  insertND True [])
⇒* {[True, False]} ∪ (False :  case4 [] of ...)
⇒* {[True, False]} ∪ {[False, True]}
⇒* {[True, False], [False, True]}
```

```
[]
[]
[(case4, 2/2, xs_s, (:))]  ?  [(case4, 2/2, xs_s, (:))]
{[(case4, 2/2, xs_s, (:))]} ∪ [(case4, 2/2, xs_s, (:))]
{[(case4,2/2,xs_s,(:))]} ∪ [(case4,2/2,xs_s,(:)), (case4,1/2,ys_s,[])]
{[(case4,2/2,xs_s,(:))], [(case4,2/2,xs_s,(:)), (case4,1/2,ys_s,[])]}
```

Trace Symbolic Information in Non-Deterministic Programs

```
insertND x xs = case4 xs of []    -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)
```

```
insertND True [False]
⇒* case4 [False] of ...
⇒* (True :  False :  []) ?  (False :  insertND True [])
⇒* {[True, False]} ∪ (False :  case4 [] of ...)
⇒* {[True, False]} ∪ {[False, True]}
⇒* {[True, False], [False, True]}
```

```
[]
[]
[(case4, 2/2, xs_s, (:))]  ?  [(case4, 2/2, xs_s, (:))]
{[(case4, 2/2, xs_s, (:))]} ∪ [(case4, 2/2, xs_s, (:))]
{[(case4,2/2,xs_s,(:))]} ∪ [(case4,2/2,xs_s,(:)), (case4,1/2,ys_s,[])]
{[(case4,2/2,xs_s,(:))], [(case4,2/2,xs_s,(:)), (case4,1/2,ys_s,[])]}
```

- Encapsulate non-determinism ⇒ multisets of results and traces

Semantics for Concolic Execution

- Evaluate an expression e w.r.t. a heap Γ and an incoming symbolic trace T to a value v, an updated heap Δ and an extended trace Υ

$$\Gamma, T : e \Downarrow \Delta, \Upsilon : v$$

- Heap: Map variables to expressions or mark them as free

$$\Gamma, \Delta, \Theta \in Heap = \mathcal{V} \to \{\mathrm{free}\} \uplus Exp$$

- Symbolic Trace: List of symbolic information for selected branches

$$T, \Upsilon, X \in Trace$$

Semantics for Concolic Execution (Excerpt)

Extension of FlatCurry semantics [Hanus, Peemöller - WFLP 2014]

Value $\Gamma \quad : v \Downarrow \Gamma \quad : v$ where $v = c(\overline{x_n})$ or $v \in \mathcal{V}$ with $\Gamma[v] = \mathrm{free}$

VarExp $$\frac{\Gamma \quad : e \Downarrow \Delta \quad : v}{\Gamma[x \mapsto e] \quad : x \Downarrow \Delta[x \mapsto v] \quad : v}$$ where $e \notin \{\mathrm{free}\}$

Or $$\frac{\Gamma \quad : e_i \Downarrow \Delta \quad : v}{\Gamma \quad : e_1 \ ?\ e_2 \Downarrow \Delta \quad : v}$$ where $i \in \{1,2\}$

Select $$\frac{\Gamma \quad : x \Downarrow \Delta \quad : c(\overline{y_n}) \qquad \Delta \quad : \sigma(e_i) \Downarrow \Theta \quad : v}{\Gamma \quad : \mathtt{case}_{id}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta \quad : v}$$

where $p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}$

Semantics for Concolic Execution (Excerpt)

Extension of FlatCurry semantics [Hanus, Peemöller - WFLP 2014]

Value $\Gamma, T : v \Downarrow \Gamma, T : v$ where $v = c(\overline{x_n})$ or $v \in \mathcal{V}$ with $\Gamma[v] = \mathrm{free}$

VarExp $\dfrac{\Gamma, T : e \Downarrow \Delta, \Upsilon : v}{\Gamma[x \mapsto e], T : x \Downarrow \Delta[x \mapsto v], \Upsilon : v}$ where $e \notin \{\mathrm{free}\}$

Or $\dfrac{\Gamma, T : e_i \Downarrow \Delta, \Upsilon : v}{\Gamma, T : e_1 \ ?\ e_2 \Downarrow \Delta, \Upsilon : v}$ where $i \in \{1, 2\}$

Select

$$\dfrac{\Gamma, T : x \Downarrow \Delta, \Upsilon : c(\overline{y_n}) \qquad \Delta, \Phi : \sigma(e_i) \Downarrow \Theta, X : v}{\Gamma, T : \mathtt{case}_{id}\ x\ \mathtt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta, X : v}$$

where $p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}, x_s$ fresh symbolic variable,
$\Phi = \Upsilon ++ [(id, i/k, x_s, c)]$

Search Strategy of *ccti*

- Symbolic traces → paths through symbolic execution tree
- Single trace entry → node of symbolic execution tree

Search Algorithm

1. Update symbolic execution tree with traces / mark branches as visited
2. Select node closest to the root with unvisited branches
3. Negate path constraint of that node
4. Try to solve constraints along path from root to selected node
    - sat: Compute new input data from model
    - unsat: Mark branches as visited and continue with 2

Selection and Solving of Path Constraints

Symbolic Trace for `nthElem [False] 0`

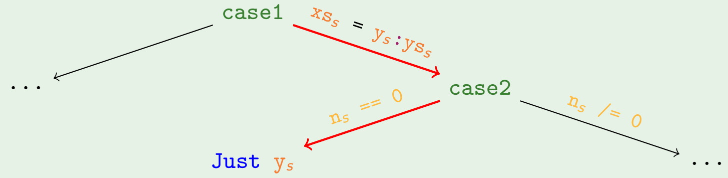
[(case1, 2/2, xs_s, (:)), (case2, 1/2, n_s == 0)]

Selection and Solving of Path Constraints

Symbolic Trace for nthElem [False] 0

[(case1, 2/2, xs_s, (:)), (case2, 1/2, n_s == 0)]

Example (Symbolic Execution Tree for nthElem)

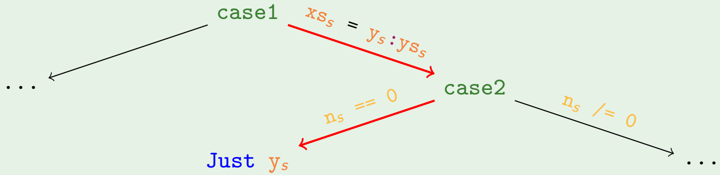


Selection and Solving of Path Constraints

Symbolic Trace for `nthElem [False] 0`

`[(case1, 2/2, xs`$_s$`, (:)), (case2, 1/2, n`$_s$` == 0)]`

Example (Symbolic Execution Tree for `nthElem`)



- Path constraint: $xs_s = y_s : ys_s \wedge n_s = 0$
- Search Strategy: Select node case1 for negation
- Apply SMT solver to modified constraints: $\forall y_s, ys_s : \neg(xs_s = y_s : ys_s)$
- Receive input data for alternative execution path: `nthElem [] 0`

Application of *ccti*

Test cases found for `nthElem`

```
nthElem [False]    0 -=- Just False
nthElem []         0 -=- Nothing
nthElem [False]    1 -=- Nothing
failing $ nthElem [False] (-1)
```

Tested Function	Initial Arguments	Cases	*ccti* Tests	Minimum Tests[1]	Full Coverage
`nthElem`	`[False], 0`	3	4	3	yes
`insertND`	`True, []`	1	2	2	yes
`addNat`	`IHi, IHi`	4	9	6	yes
`perm`	`[False]`	2	1	1	no
`semRE`	`Lit A`	2	3	1	no

[1] Number of test cases sufficient for full program coverage

Current Limitations of *ccti*

Curry

```
insertND x xs = case4 xs of []   -> [x]
                            y:ys -> (x : y : ys) ? (y : insertND x ys)

perm xs = case5 xs of []   -> []
                      y:ys -> insertND y (perm ys)
```

- For `perm [False]` *ccti* finds only single test case
- No coverage of second branch of `case4`
- **Problem:** Second branch of `case5` is already covered by top-level call
  $\Rightarrow$ Not considered for recursive call of `perm`
  $\Rightarrow$ `insertND` is never called with non-empty list
- **Solution:** Apply alternative coverage criterion for search

Conclusion

- *ccti* - concolic testing of functional logic programs
- Extension of FlatCurry semantics to trace symbolic information
- Application of SMT solver for solving of path constraints
- First evaluation: applicable for test case generation
- Global branch coverage insufficient for some examples

Future Work

- Implementation of alternative strategies and code coverage criteria
- Further evaluation of *ccti*:
  - More complex programs
  - Comparison with narrowing-based test case generation (CurryCheck)