When **Functional Languages** Meet **Symbolic Evaluation**

Weixin Zhang

Motivation

- Symbolic evaluation is good at program analysis
- It is widely used in imperative languages compared to functional languages
- Functional languages are good candidate for doing symbolic evaluation
- Using symbolic evaluation can improve test coverage dramatically

Contribution

- Implement a symbolic evaluator for a functional language

- Integrate the symbolic evaluator with a SMT solver (Z3) to kick out infeasible paths

Review

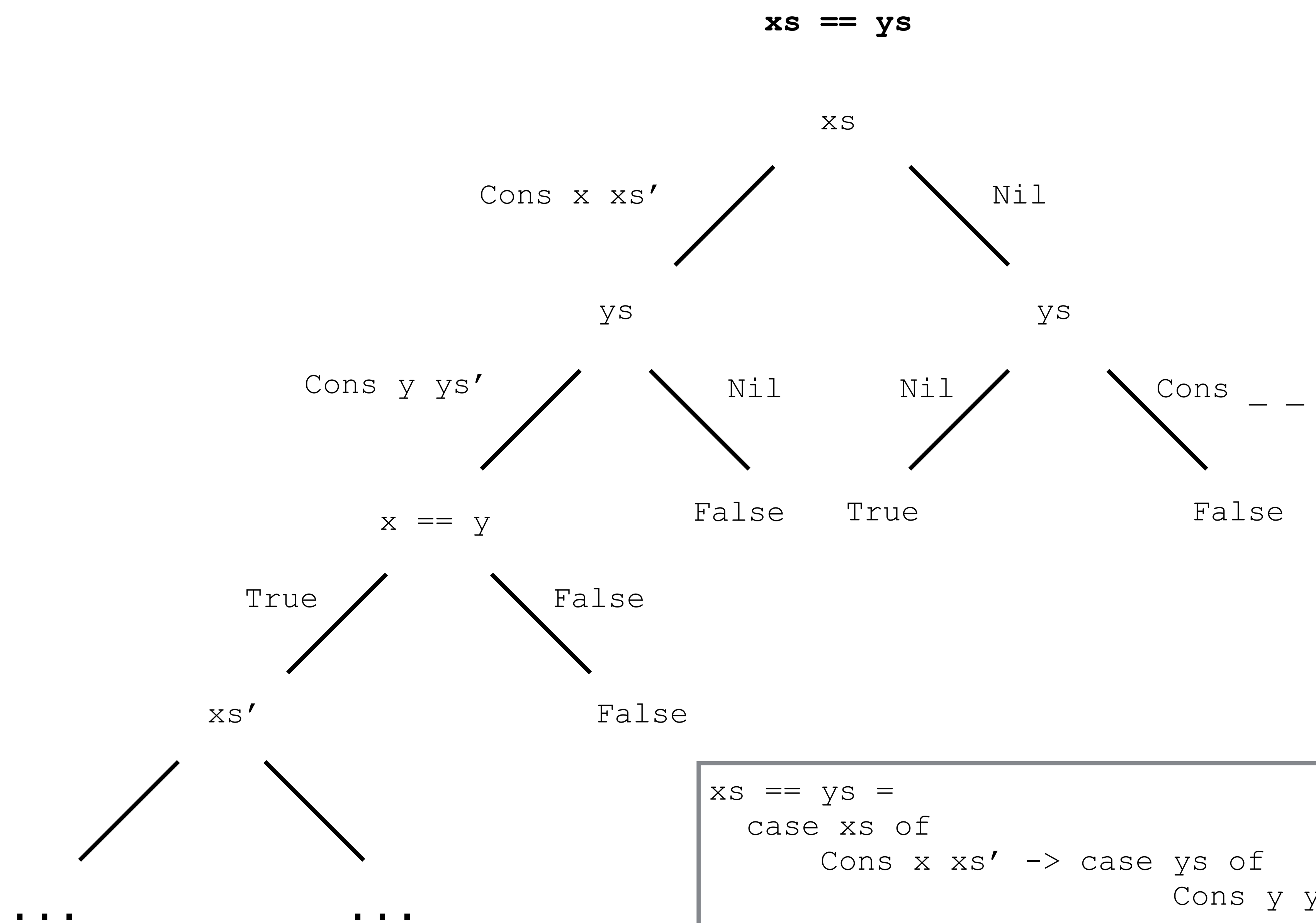
- What is symbolic evaluation (execution)?
  - Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. — *Wikipedia (symbolic execution)*
- What will cause it to execute different part of a program?
  - Imperative languages: **if, for** and **while**
  - Functional languages: **if** and **pattern matching**

Symbolic Evaluation: an Example

```
(==)     :: (Eq a) => [a] -> [a] -> Bool
(x:xs)   == (y:ys) = x == y && xs == ys
[]       == []     = True
_        == _      = False
```

```
(==) :: (Eq a) => [a] -> [a] -> Bool
xs == ys =
  case xs of
      Cons x xs’ -> case ys of
                          Cons y ys’ -> if x == y
                                        then xs’ == ys’
                                        else False
                          Nil        -> False
      Nil        -> case ys of
                          Nil      -> True
                          Cons _ _ -> False
```

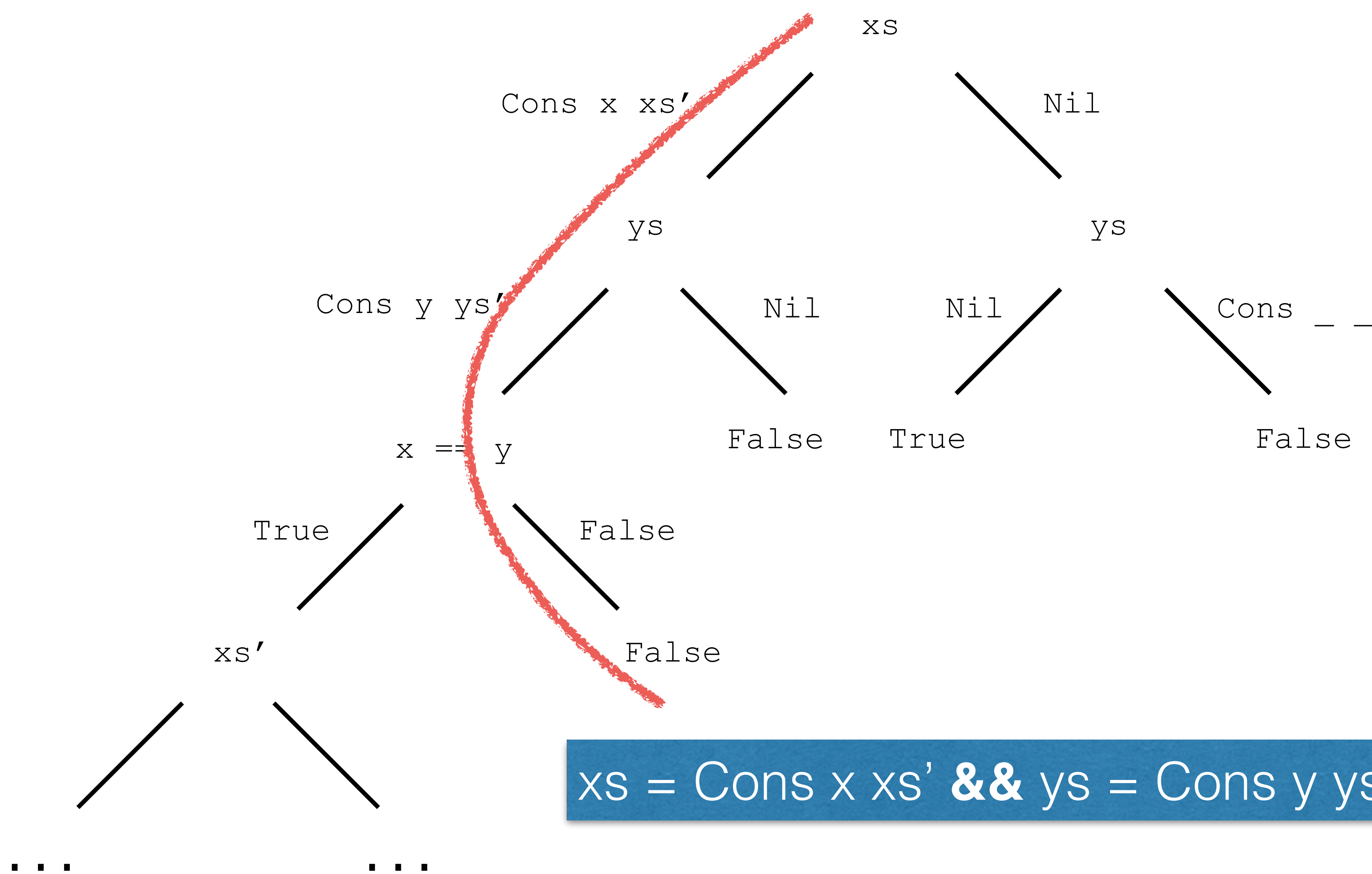
Execution Tree



Infinite

```
xs == ys =
  case xs of
      Cons x xs' -> case ys of
                      Cons y ys' -> if x == y
                                    then xs' == ys'
                                    else False
                      Nil        -> False
      Nil        -> case ys of
                      Nil      -> True
                      Cons _ _ -> False;
```

Path Conditions



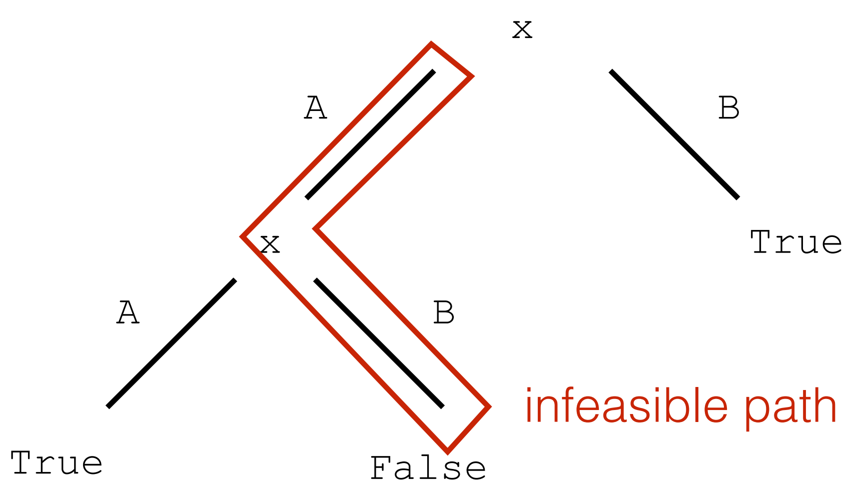
xs = Cons x xs' **&&** ys = Cons y ys' **&&** x != y

Infeasible Paths

- Consider the following program:

```
data T = A | B;

alwaysTrue x =
  case x of
      A -> case x of
               A -> True
               B -> False
      B -> True;
```



- Is `alwaysTrue` always true?

  Yes. The path to False is unsatisfiable since x can not be A and B simultaneously

- How can we know that ?

  SMT solver will tell us!

Encoding Datatypes in SMT solver

```
data T = A | B
```

current approach

```
(declare-sort T)

(declare-fun A () T)
(declare-fun B () T)

(assert (distinct A B))
```

ideal approach

```
(declare-datatypes () ((T A B)))
```

Encoding Path Conditions in SMT solver

```
data PolyList[T] = Nil
                 | Cons T PolyList[T]
```

declare datatype

declare arguments

declare new bindings introduced by pattern matching

assertion

reach a leaf check satisfiability

xs = Cons x xs’ && ys = Cons y ys’ && x != y

```
(declare-sort T)
(declare-datatypes (T)
  ((PolyList nil (cons (cons_1 T) (cons_2 PolyList)))))

(declare-const xs (PolyList (T)))
(declare-const ys (PolyList (T)))

(push)
  (declare-const x T)
  (declare-const xs1 (PolyList (T)))

  (assert (= xs (cons x xs1)))
  (push)
    (declare-const y T)
    (declare-const ys1 (PolyList (T)))
    (push)
       (assert (= ys (cons y ys1)))
       (push)
          (assert (= x y))
           …
       (pop)
       (push)
          (assert (not (= x y)))
          (check-sat)
       (pop)
        …
```

The Pros of Symbolic Execution

```
length :: [Int] -> Integer
length xs =
  case xs of
       []    -> 0
       _:xs' -> 1 + length xs'
```

Apply length once to a list of size	0	1	2	…	n
test cases covered by symbolic value	1	2^64	~(2^64)^2		~(2^64)^n
test cases covered by concrete value	1	1	1	…	1

The Pros of Symbolic Execution

- The drawbacks of black-box testing have already been covered in my last talk
- A typical property in *QuickCheck:*

type signature needed

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = xs == reverse (reverse xs)
```

- But some properties are polymorphic, can we omit the type signature?
  - Yes, you can do this for some properties, e.g.,

```
prop_length xs ys = length xs + length ys == length (xs ++ ys)
```

  - But, be careful! The result may be a **false positive**

```
prop_reverse' :: [()] -> Bool
prop_reverse' xs = xs == reverse xs
```

The Cons of Symbolic Evaluation

- Path Explosion
  - The # of paths grows exponentially with the # of control structures
  - Can even be infinite in the case of programs with unbounded loop iterations (recursion). E.g., (==)
    - Laziness can help us!
- Limited by the power constraint solver
  - Can not handle non-linear / complex constraints
  - Performance bottleneck
    - Reduce execution time by parallelizing independent paths

Case Study: Implement `reverse` in FCore

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse :: [a] -> [a]
reverse xs =
  case xs of
    []      -> []
    (x:xs') -> reverse xs' ++ [x]
```

```
data PolyList[A] = Nil | Cons A PolyList[A];

let rec (===)[A] (xs: PolyList[A]) (ys: PolyList[A]): Bool =
  case xs of
     Nil -> (case ys of
                    Nil -> True
                  |  Cons _ _ -> False)
   | Cons x xs' -> (case ys of
                     Nil        -> False
                      | Cons y ys' -> if x == y && (xs' ===[A] ys')
                                        then True
                                        else False);

let rec append[A] (xs: PolyList[A]) (ys: PolyList[A]): PolyList[A] =
  case xs of
      Nil        -> ys
    | Cons x xs' -> Cons[A] x (append[A] xs' ys);

let rec reverse[A] (xs: PolyList[A]): PolyList[A] =
  case xs of
      Nil        -> Nil[A]
    | Cons x xs' -> reverse[A] xs' `append[A]` Cons[A] x (Nil[A]);
```

Case Study: Write Properties for `reverse`

```
let prop_reverse[A] (xs: PolyList[A]): Bool =
    xs ===[A] (reverse[A] (reverse[A] xs));
prop_reverse
```

```
let prop_reverse_wrong[A] (xs: PolyList[A]): Bool =
    xs ===[A] (reverse[A] xs);
prop_reverse_wrong
```

Future Work

- Change the encoding of datatypes
- Parse the model given by Z3
- Construct counter-example from the parsed model

Reference

- Z3 guide:
    - http://rise4fun.com/z3/tutorial
- Z3 Haskell API:
    - http://iago.bitbucket.org/z3-haskell/doc/0.3.2/Z3-Monad.html

Thank You!