EasyCheck

Test Data for Free

Jan Christiansen & Sebastian Fischer
University of Kiel, Germany

FLOPS 2008, Ise, Japan

Curry

```
(?) :: a -> a -> a
x ?   = x
_ ? x = x

unknown :: a
unknown = x where x free
```

Combined Functional Logic Programming

```
insert :: a -> [a] -> [a]
insert x xs     = x : xs
insert x (y:xs) = y : insert x xs

permute = foldr insert []

psort xs | sorted ys = ys
 where ys = permute xs
```

```
> permute unknown
[]                                  More?
[x2]                                More?
[x2,x7]                             More?
[x7,x2]                             More?
[x2,x7,x11]                         More?
[x7,x2,x11]                         More?
[x2,x11,x7]                         More?
[x11,x2,x7]                         More?
[x7,x11,x2]                         More?
[x11,x7,x2]                         More? no
```

```
> (l, permute l) where l free

([],[])                                More?

([x2],[x2])                            More?

([x2,x7],[x2,x7])                      More?

([x2,x7],[x7,x2])                      More?

([x2,x7,x11],[x2,x7,x11])              More?
([x2,x7,x11],[x7,x2,x11])              More?
([x2,x7,x11],[x2,x11,x7])              More?
([x2,x7,x11],[x11,x2,x7])              More?
([x2,x7,x11],[x7,x11,x2])              More?
([x2,x7,x11],[x11,x7,x2])              More? no
```

Curry

- Nondeterminism + Free Variables
- Test Input for **`free`**
- Manual Testing without further support

Property-Based Testing*

```
psortSorts :: [Int] -> Prop
psortSorts xs =
  psort xs -=- mergeSort xs
```

*Thank You QuickCheck!

Non-Determinism

```
> easyCheck1 psortSorts
Falsified by 6th test.
Arguments:
 [0,0]
Results:
 ([0,0],[0,0])
 ([0,0],[0,0])
```

Two Equal Results

```
> psort [0,0]
 [0,0]                                  More?
 [0,0]                                  More?
No more solutions.
```

Deterministic Equality

```
psortSorts :: [Int] -> Prop
psortSorts xs =
  psort xs -=- mergeSort xs
```

Semantic Equivalence

```
psortSorts :: [Int] -> Prop
psortSorts xs =
  psort xs <~> mergeSort xs
```

Success!

```
> easyCheck1 psortSorts
Ok, passed 100 tests.
```

Investigating Input

```
psortSortsSmall :: [Int] -> Prop
psortSortsSmall xs =
  classify (length xs <= 2) "small"
    (psortSorts xs)
```

```
> easyCheck1 psortSortsSmall
OK, passed 100 tests - 45% small.
```

Custom Input

```
shuffle :: Int -> [a] -> [a]
shuffle _    []     = []
shuffle _    [x]    = [x]
shuffle _    [x,y]  = [x,y] ? [y,x]
shuffle len xs@(_:_:_:_) =
  x : shuffle mid           ys
   ++ shuffle (len-mid-1) zs
 where
  mid = len `div` 2 + (0?1)
  (ys,x:zs) = splitAt mid xs
```

Custom Input

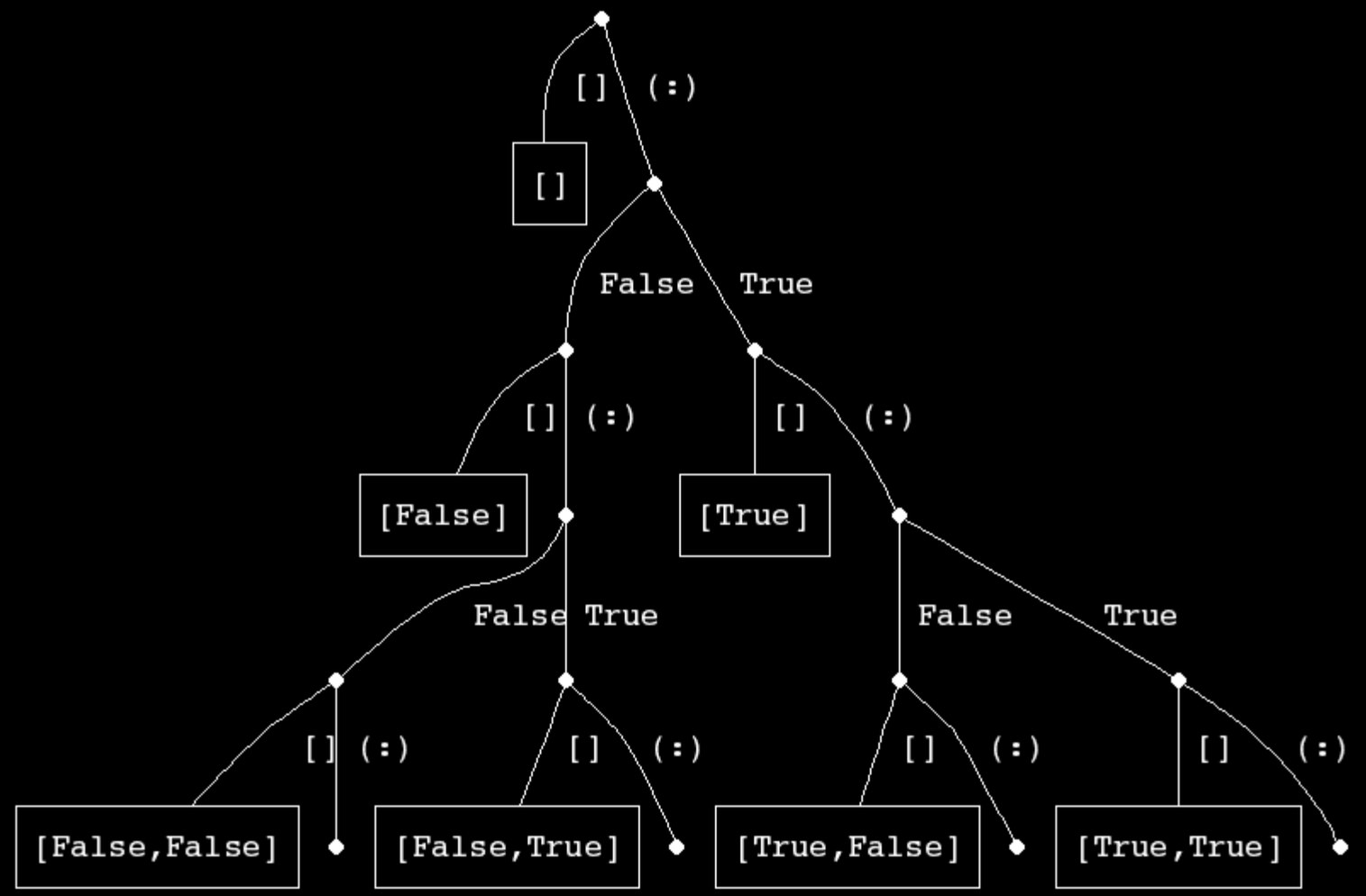
```
psortSortsLen :: Int -> Prop
psortSortsLen len =
  (0 < len && len < 10) ==>
  for (shuffle len [1..len])
    psortSortsSmall


> easyCheck1 psortSortsLen
OK, passed 100 tests - 3% small.
```

EasyCheck

- Like QuickCheck, only simpler!
- Support for Non-Determinism
  - Standard Input = Free Variables
  - Custom Input = Non-Det Operation
  - No fixed strategy or probabilities (yet)

Enumerating Test Input

```
data SearchTree a
  = Value a
  | Or [SearchTree a]

searchTree :: a -> SearchTree a
searchTree external

> searchTree (False ? True)
Or [Value False,Value True]  More?
No more solutions.
```


[]
(:)
[]
False
True
[]
(:)
[]
(:)
[False]
[True]
False
True
False
True
[]
(:)
[]
(:)
[]
(:)
[]
(:)
[False,False]
[False,True]
[True,False]
[True,True]

Depth-First or Breadth-First Search?

```
[]
[False]
[False,False]
[False,False,False]
...
```

```
[]
[False]
[True]
[False,False]
[False,True]
...
```

Not Good Enough

- depth-first search:

  incomplete (does not reach every node)

- breadth-first search:

  to many small values

  first node of level n after $O(2^n)$ others

Diagonalization

```
diagonal :: [[a]] -> [a]
diagonal = ...

> diagonal [[ (x,y) | y <- [1..]]
                    | x <- [1..]]
[(1,1)
,(1,2),(2,1)
,(1,3),(2,2),(1,3)
,(1,4),(2,3),(3,2),(4,1)
,...
```

Level Diagonalization

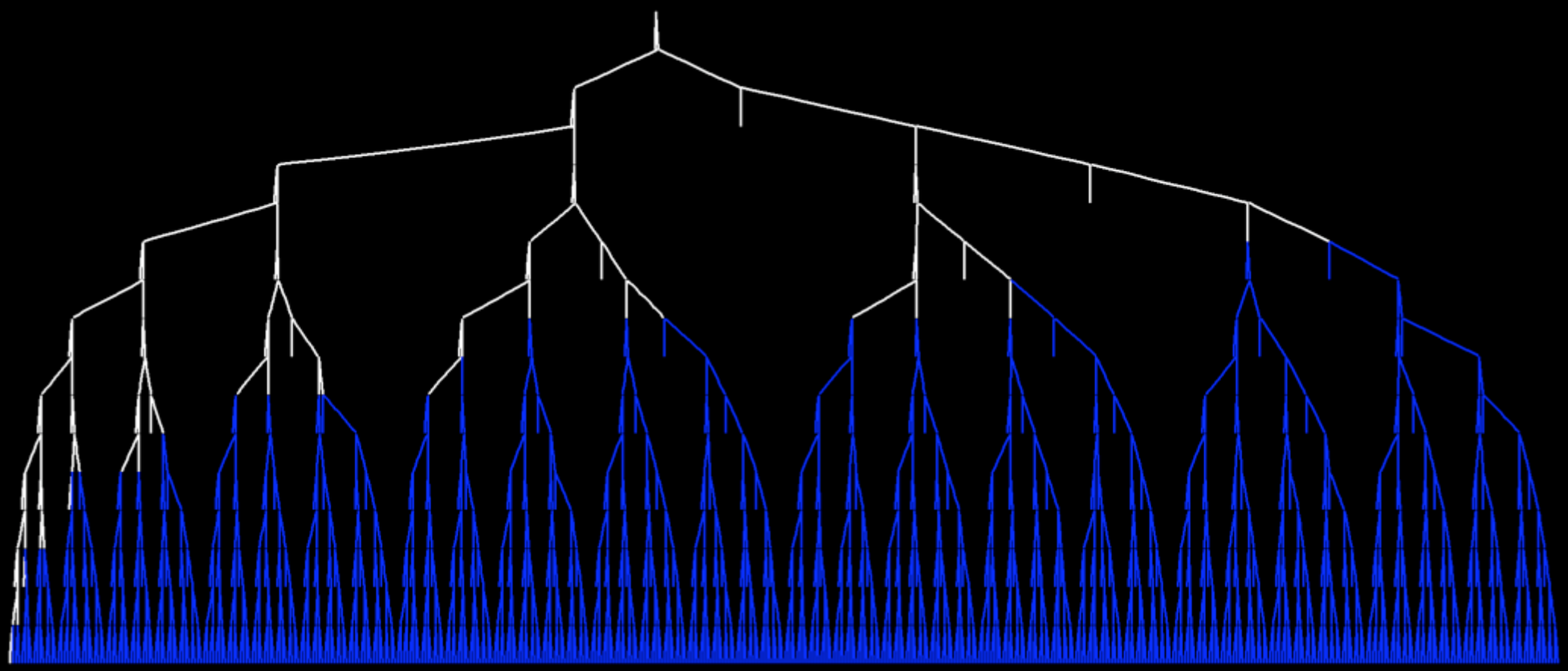
```
levelDiag :: SearchTree a -> [a]
levelDiag t =
  [ x | Value x <-
        diagonal (levels [t]) ]

levels ts =
  if null ts then []
  else ts:levels [ u | Or us <- ts
                     ,     u <- us ]
```

Level Diagonalization

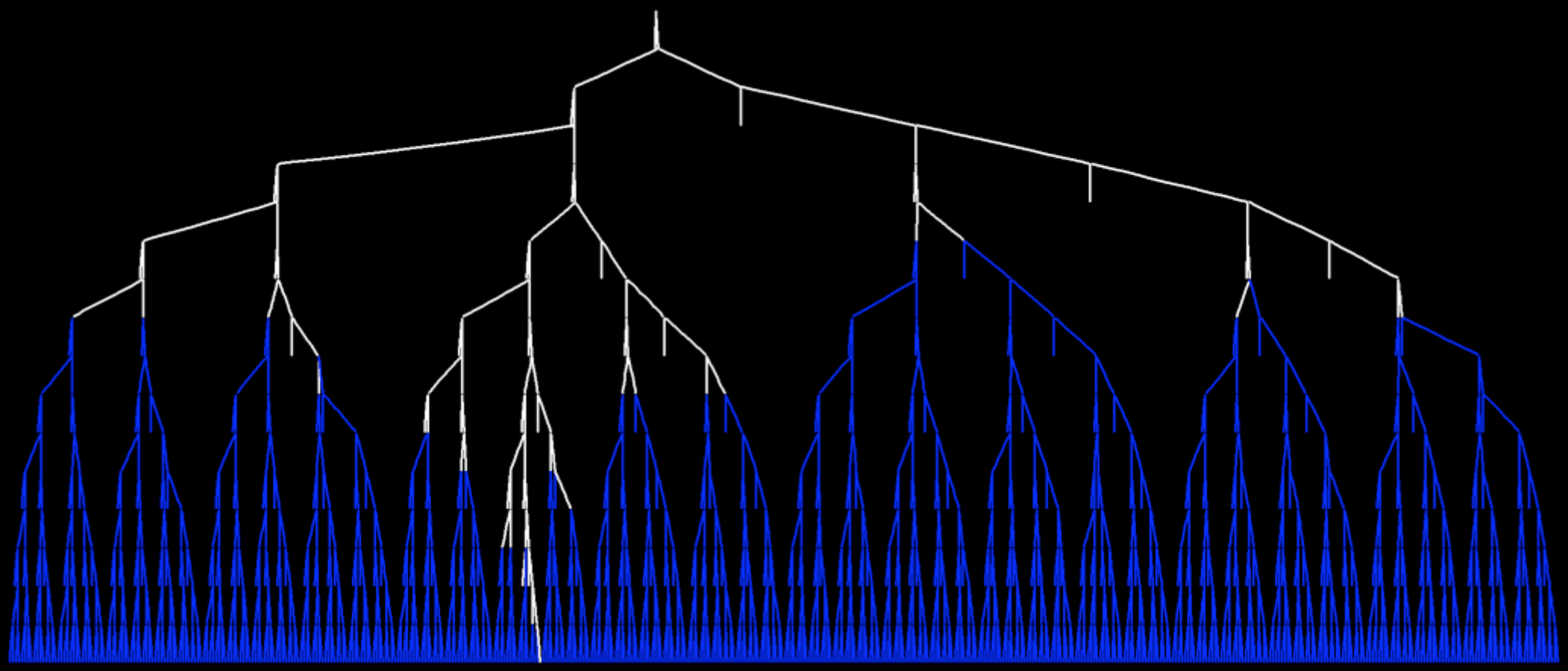
- complete (reaches every node)
- large values early

  first node of level n after $O(n^2)$ others

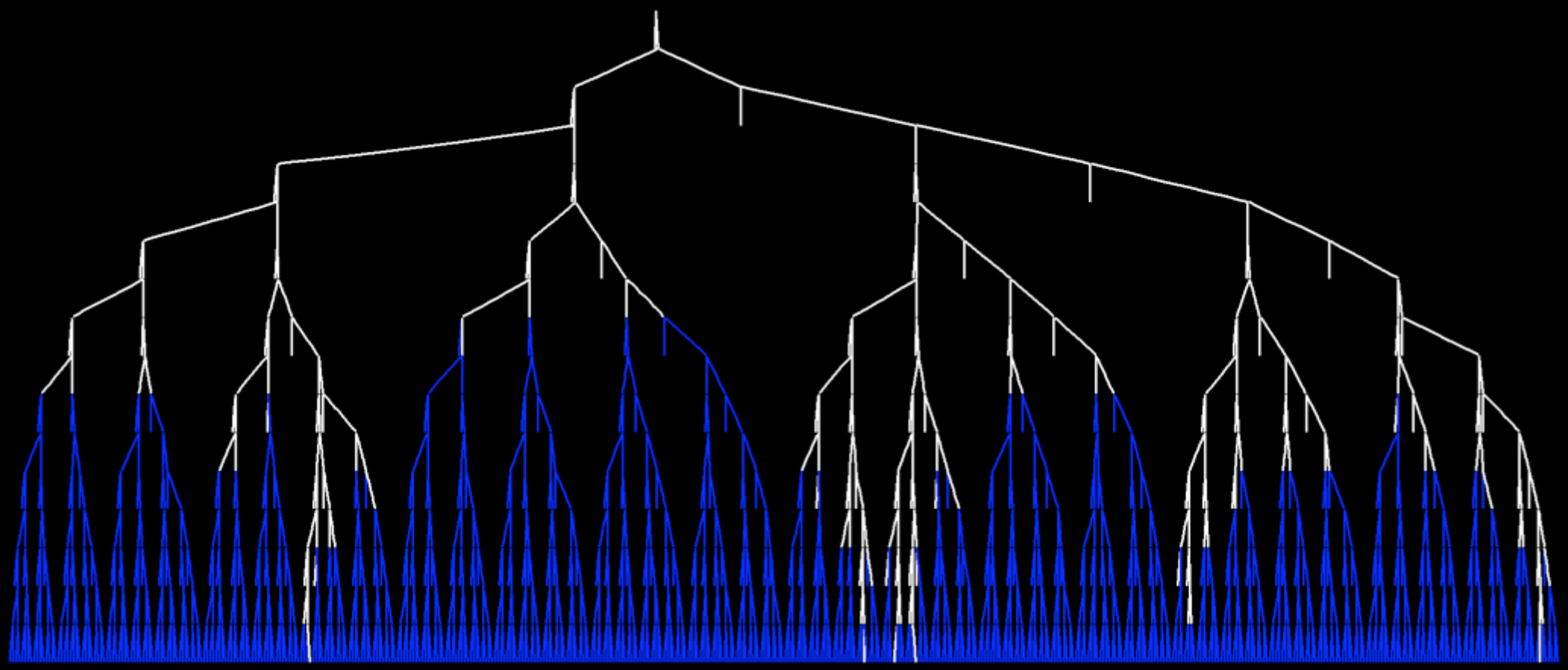
Left Biased



Randomization



Multiple Searches



Conclusions

- Free Variables and Non-Determinism

  Easy way to describe test input!

- Separated Declaration and Enumeration

  Old code benefits from new strategies

- Randomized Level Diagonalization

  complete, advancing, balanced