A Hybrid Testing Monad

Calvin Beck

John Hughes

Leonidas Lampropoulos

Benjamin C. Pierce

WG2.8, May 2019

```
return a  =  [a]

m >>= k  =  concat [ k x | x <- m ]
```

[a]

The list monad

```
g1 = [2,3,4]
g2 = \x -> [x^2, x^3, x^4]

g = do x <- g1
       y <- g2 x
       return (x,y)
```

→

```
[(2,4),(2,8),(2,16),(3,9),(3,27),(3,81),(4,16),(4,64),(4,256)]
```

[a]

The backtracking monad

Cf. SmallCheck,
Lazy Smallcheck,
LeanCheck,
etc.

(Eliding size parameter)

```
return a = \r -> a

m >>= k =
  \r0 -> let (r1,r2) = split r0
             m'      = k (m r1)
         in m' r2
```

Pseudo-random number source

StdGen -> a

The random generation monad

Cf. QuickCheck,
etc., etc., etc.

(Eliding size parameter)

StdGen -> [a]

↑

A hybrid “random generation + backtracking” monad

```
return a = \r -> [a]

m >>= k =
  \r0 -> let (r1,r2) = split r0
             aux r [] = []
             aux r (a:as) =
               let (r1, r2) = split r
               in (k a) r1 ++ aux r2 as
         in aux r2 (m r1)
```

StdGen -> [a]

Random generation
with backtracking!

Pure random generation

Pure enumeration

15-20% overhead compared to original QuickCheck

All the combinators from both!

plus some interesting new ones…

```haskell
retry :: Int -> Gen a -> Gen a

retry 0 g = \r -> []
retry n g = \r -> let (r1, r2) = split r
                  in g r1 ++ retry (n-1) g r2
```

```haskell
randomOrder :: (Int, Int) -> Gen Int
```

```haskell
takeG :: Int -> Gen a -> Gen a
```

```haskell
weightedAllof :: [(Int, Gen a)] -> Gen a
```

And a bunch of others…

Plan

- When does backtracking help?
- Two case studies
- A little analytical model
- Discussion!

When does backtracking help?

1. If g1 is expensive and g2 is cheap, we may want to reuse each result from g1 to generate several results from g2

2. If g1 and/or g2 can fail, we may want to retry g2 several times for each successful result from g1

1. If g1 is expensive and g2 is cheap, we may want to reuse each result from g1 to generate several results from g2

2. If g1 and/or g2 can fail, we may want to retry g2 several times for each successful result from g1

Pure random

```
prop :: (A,B) -> Bool

g1 :: Gen A
g2 :: A -> Gen B

g :: Gen (A,B)
g = do
  x <- g1
  y <- g2 x
  return (x,y)
```

Hybrid

```
prop :: (A,B) -> Bool

g1 :: Gen A
g2 :: A -> Gen B

g :: Gen (A,B)
g = do
  x <- g1
  y <- retry 42 $ g2 x
  return (x,y)
```

1. If g1 is expensive and g2 is cheap, we may want to reuse each result from g1 to generate several results from g2

2. If g1 and/or g2 can fail, we may want to retry g2 several times for each successful result from g1

Pure random

```
prop :: (A,B) -> Bool

g1 :: Gen (Maybe A)
g2 :: A -> Gen (Maybe B)

g :: Gen (Maybe (A,B))
g = do
  xo <- g1
  case xo of
    Nothing -> return Nothing
    Just x -> do
      yo <- g2 x
      case yo of
        Nothing -> return Nothing
        Just y -> return $ Just (x,y)
```

Hybrid

```
prop :: (A,B) -> Bool

g1 :: Gen A
g2 :: A -> Gen B

g :: Gen (A,B)
g = do
  x <- g1
  y <- g2 x
  return (x,y)
```

1. If g1 is expensive and g2 is cheap, we may want to reuse each result from g1 to generate several results from g2

2. If g1 and/or g2 can fail, we may want to retry g2 several times for each successful result from g1

Pure random

```
prop :: (A,B) -> Bool

g1 :: Gen (Maybe A)
g2 :: A -> Gen (Maybe B)

g :: Gen (Maybe (A,B))
g = do
  xo <- g1
  case xo of
    Nothing -> return Nothing
    Just x -> do
      yo <- g2 x
      case yo of
        Nothing -> return Nothing
        Just y -> return $ Just (x,y)
```

Hybrid

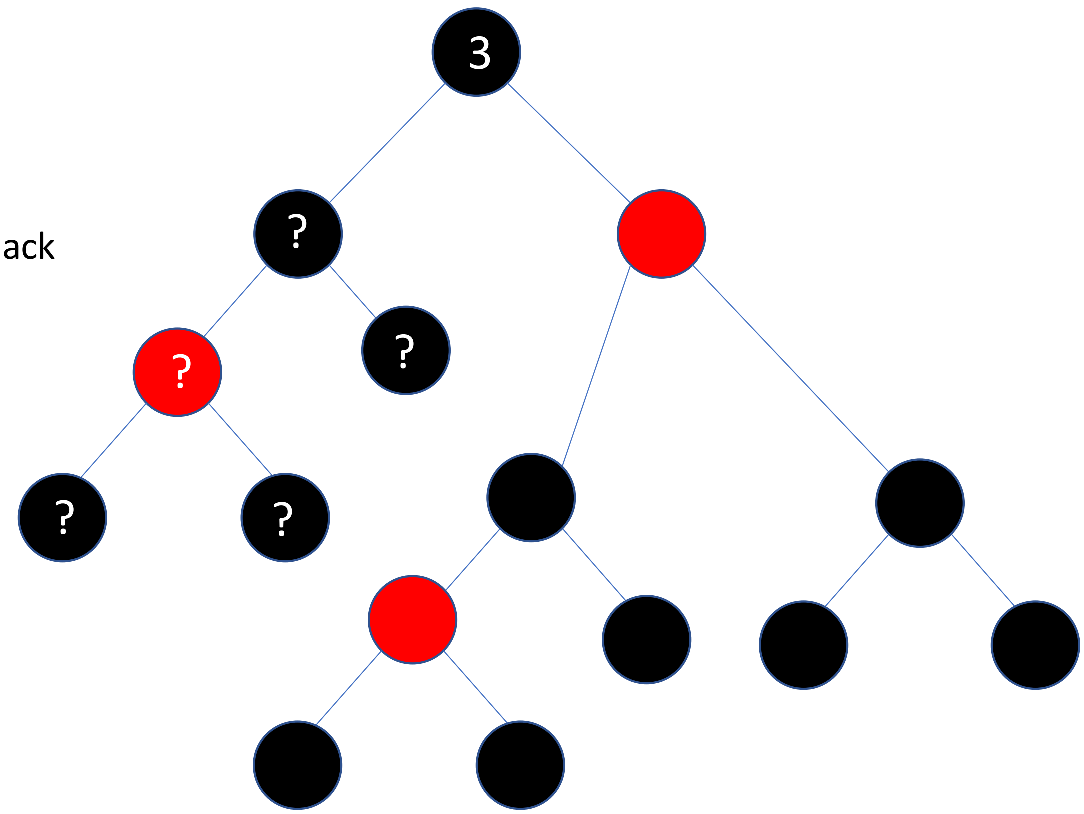
```
prop :: (A,B) -> Bool

g1 :: Gen A
g2 :: A -> Gen B

g :: Gen (A,B)
g = do
  x <- g1
  y <- retry 42 $ g2 x
  return (x,y)
```

Case studies

Case Study: Red-Black Trees

Review: Red-black tree invariants…

- Each node's label is greater than any in its left subtree and less than any in its right subtree
- Root and leaves are black
- Red nodes have black children
- Every path to a leaf has the same number of black nodes



Interesting test case for backtracking because purely random generation can sometimes get "stuck"…

Original QuickCheck

```
genRBT :: Int -> Color -> Int -> Int -> Gen (Maybe (Tree Int))

genRBT 0 R lo hi = return $ Just Empty
genRBT 0 B lo hi
  | hi - lo <= 1 = return $ Just Empty
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     elements [Just Empty, Just (Node R x Empty Empty)]
genRBT bh c lo hi
  | hi - lo <= 1 = return Nothing
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     c' <- if c == R then return B else elements [B, R]
     let bh' = if c' == B then bh - 1 else bh
     if (not (x - lo >= 2 ^ bh' && hi - x >= 2 ^ bh')) then
      return Nothing
     else do
      ml <- genRBT bh' c' lo x
      mr <- genRBT bh' c' x hi
      case (ml, mr) of
       (Just l, Just r) -> return $ Just $ Node c' x l r
       _ -> return Nothing
```

Issues:

1. Ugly: “Maybe plumbing” all over the place
2. Slow: Backtracks all the way to the beginning each time!

Original QuickCheck

```
genRBT :: Int -> Color -> Int -> Int -> Gen (Maybe (Tree Int))

genRBT 0 R lo hi = return $ Just Empty
genRBT 0 B lo hi
  | hi - lo <= 1 = return $ Just Empty
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     elements [Just Empty, Just (Node R x Empty Empty)]
genRBT bh c lo hi
  | hi - lo <= 1 = return Nothing
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     c' <- if c == R then return B else elements [B, R]
     let bh' = if c' == B then bh - 1 else bh
     if (not (x - lo >= 2 ^ bh' && hi - x >= 2 ^ bh')) then
      return Nothing
     else do
      ml <- genRBT bh' c' lo x
      mr <- genRBT bh' c' x hi
      case (ml, mr) of
       (Just l, Just r) -> return $ Just $ Node c' x l r
       _ -> return Nothing
```

Hybrid

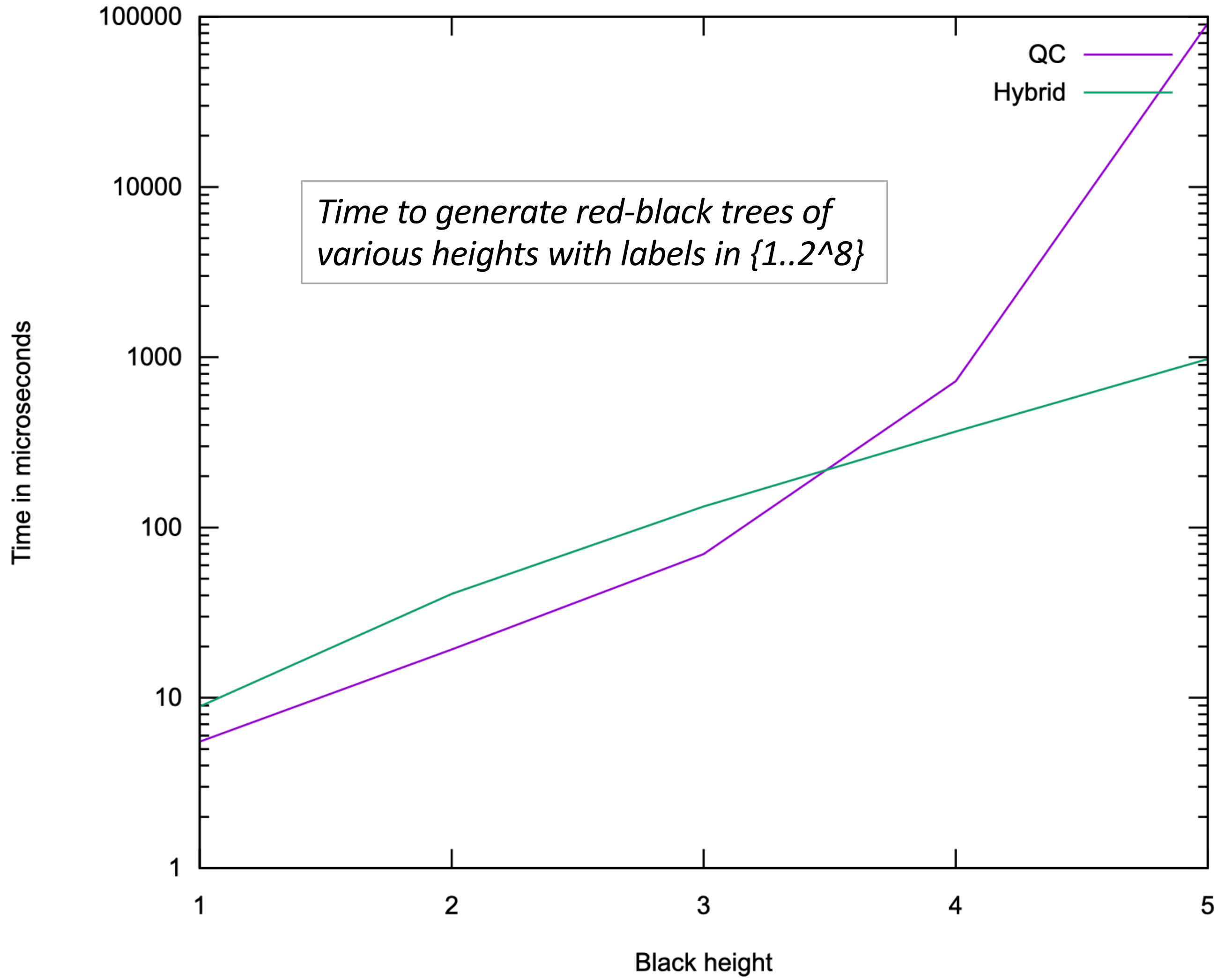
```
genRBT :: Color -> Int -> Int -> Gen (Tree Int)

genRBT R lo hi = return Empty
genRBT B lo hi
  | hi - lo <= 1 = return Empty
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     elements [Empty, Node R x Empty Empty]
genRBT f bh c lo hi
  | hi - lo <= 1 = empty
  | otherwise = do
     x <- choose (lo + 1, hi - 1)
     c' <- if c == R then return B else elements [B, R]
     let bh' = if c' == B then bh - 1 else bh
     guard (x - lo >= 2 ^ bh')
     guard (hi - x >= 2 ^ bh')
     l <- genRBT f bh' c' lo x
     r <- genRBT f bh' c' x hi
     return $ Node c' x l r
```

Original QuickCheck

```
genRBT :: Int -> Color -> Int -> Int -> Gen (Maybe (Tree Int))

genRBT 0 R lo hi = return $ Just Empty
genRBT 0 B lo hi
  | hi - lo <= 1 = return $ Just Empty
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     elements [Just Empty, Just (Node R x Empty Empty)]
genRBT bh c lo hi
  | hi - lo <= 1 = return Nothing
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     c' <- if c == R then return B else elements [B, R]
     let bh' = if c' == B then bh - 1 else bh
     if (not (x - lo >= 2 ^ bh' && hi - x >= 2 ^ bh')) then
       return Nothing
     else do
       ml <- genRBT bh' c' lo x
       mr <- genRBT bh' c' x hi
       case (ml, mr) of
         (Just l, Just r) -> return $ Just $ Node c' x l r
         _ -> return Nothing
```

Hybrid

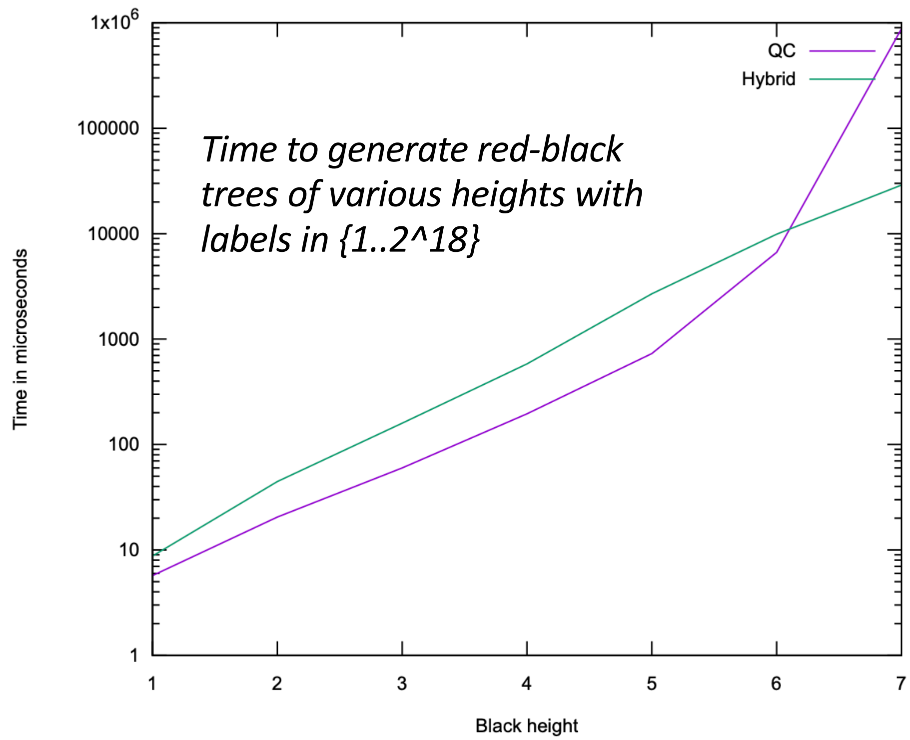
```
genRBT :: Color -> Int -> Int -> Gen (Tree Int)

genRBT R lo hi = return $ Empty
genRBT B lo hi
  | hi - lo <= 1 = return Empty
  | otherwise = do
     x <- choose (lo + 1 , hi - 1)
     elements [Empty, Node R x Empty Empty]
genRBT f bh c lo hi
  | hi - lo <= 1 = empty
  | otherwise = do
     x <- randomOrder (lo + 1, hi - 1)
     c' <- if c == R then return B else elements [B, R]
     let bh' = if c' == B then bh - 1 else bh
     guard (x - lo >= 2 ^ bh')
     guard (hi - x >= 2 ^ bh')
     l <- genRBT f bh' c' lo x
     r <- genRBT f bh' c' x hi
     return $ Node c' x l r
```



Time to generate red-black trees of various heights with labels in {1..2^8}
QC
Hybrid
Time in microseconds
100000
10000
1000
100
10
1
1
2
3
4
5
Black height

Criticisms

- Constrained range of labels
  - But: look…
- Hybrid monad helps only for large-ish black heights
  - But: random testing experts tell us to generate structures much larger than the minimal counterexample
- We already know how to generate random red-black trees
  - Generate a random list and insert its elements into a tree one by one
  - But: Can all well-formed red-black trees be generated in this way?



A more realistic example…

Case Study: IFC

- Setup
  - A tiny RISC instruction set with built-in dynamic information-flow monitoring
  - Correctness property: *Noninterference*
    - "Secret data does not flow to publicly accessible locations"
    - I.e. Low-indistinguishable states remain low-indistinguishable after the machine steps

- Experimental procedure
  - Systematically inject bugs into the IFC monitor
  - Generate pairs of initial machine states with identical public parts
    - For each, step the machine by executing the instruction at the current PC and check whether the resulting machine states still have identical public parts
  - For each injected bug, measure how long it takes to find a pair of machine states that demonstrate it
  - Compare MTTF for two generation strategies…

Original

```
gen_states :: Gen (Machine, Machine)
gen_states = do
  m1_init <- gen_machine
  m2_init <- gen_indist m1_init
  instr <- gen_valid_instr m1_init
  m1 <- store_instr m1_init instr
  m2 <- store_instr m2_init instr
  return  (m1, m2)
```

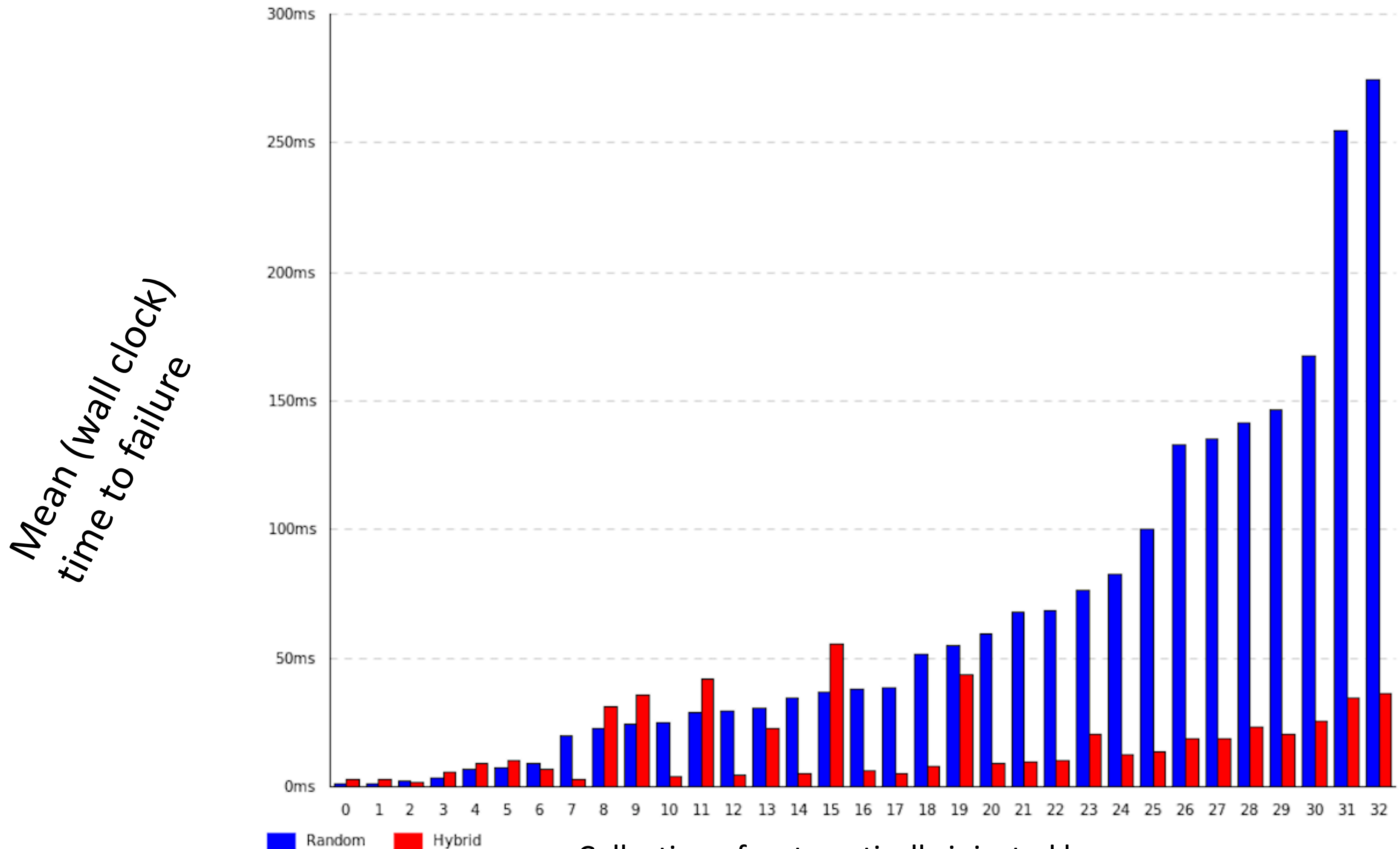
Nb.: This is "Haskell pseudocode." Actual implementation is in Coq using QuickChick.

Original

```
gen_states :: Gen (Machine, Machine)
gen_states = do
  m1_init <- gen_machine
  m2_init <- gen_indist m1_init
  instr <- gen_valid_instr m1_init
  m1 <- store_instr m1_init instr
  m2 <- store_instr m2_init instr
  return  (m1, m2)
```

With backtracking

```
gen_states =
  m1_init <- gen_machine
  m2_init <- gen_indist m1_init
  instr <- enum_valid_instr m1_init
  m1 <- store_instr m1_init instr
  m2 <- store_instr m2_init instr
  return (m1, m2)
```



Collection of systematically injected bugs
(ordered "easiest to hardest")

(pure enumeration is hopeless)

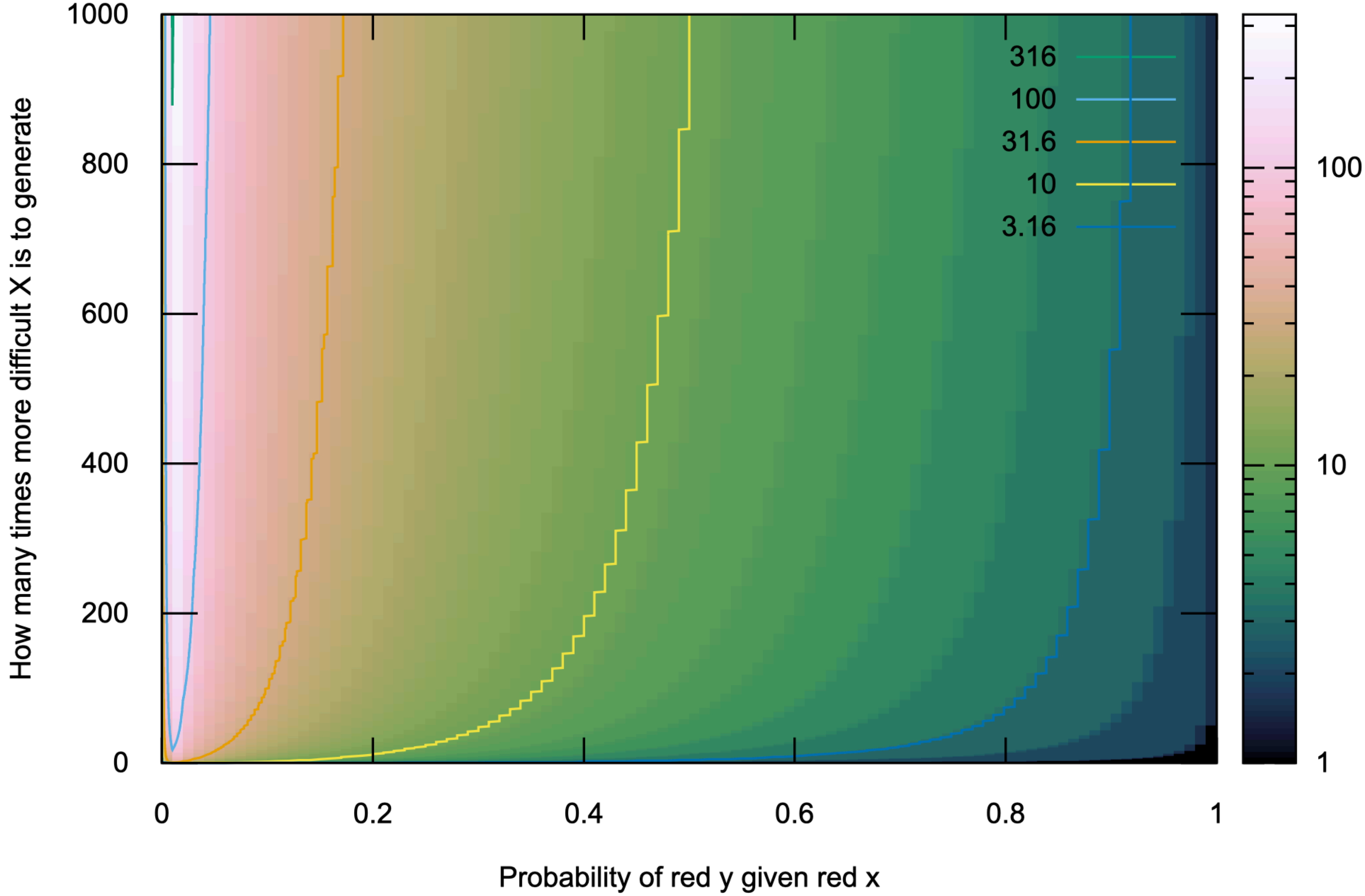
4x average speedup

A (simplified) analytical model

- Setup
  - Generate pairs (x,y), where generating y depends on x
  - Some pairs are red -- goal is to find one
  - For each x, assume that
    - either *no* (x,y) pairs are red (and we say x itself is black)
    - or else *some* (x,y) pair is red (and we say x is red)
  - Each x is red with 50% probability
  - When x is red, each (x,y) pair is red with equal probability

- Parameters
  - ratio between cost of generating an x and cost of generating a y
  - density of red ys (given a red x)

- Output
  - Optimal number of ys to generate for each generated x, to minimize expected time to generate a red (x,y)



1000
800
600
400
200
0
0
0.2
0.4
0.6
0.8
1
How many times more difficult X is to generate
Probability of red y given red x
316
100
31.6
10
3.16
100
10
1

Discussion

We’d love to have more real-world examples!

Gory details

Review

The Random Generation Monad

Gen a = Int -> StdGen -> a

```
newtype Gen a = Gen { run :: Int -> StdGen -> a }

instance Monad Gen where
  return a    = Gen (\n r -> a)
  Gen m >>= k =
    Gen (\n r0 -> let (r1,r2) = split r0
                      Gen m'  = k (m n r1)
                  in m' n r2)

choose :: Random a => (a, a) -> Gen a

frequency :: [(Int, Gen a)] -> Gen a

suchThatMaybe :: Gen a -> (a -> Bool) -> Gen (Maybe a)
```

The Enumeration Monad

Gen a = Int -> [a]

```
newtype Gen a = Gen { run :: Int -> [a] }

instance Monad Gen where
  return a    = Gen (\n -> [a])
  Gen m >>= k =
    Gen (\n -> do x <- m n
                  run (k x) n)

enumerate :: [a] -> Gen a

allof :: [Gen a] -> Gen a

empty :: Gen a
```

aka msum
(nb: no weights!)

Failure!

The Hybrid Monad

Gen a = Int -> StdGen -> [a]

newtype Gen a = Gen { run :: Int -> StdGen -> [a] }

frequency :: [(Int, Gen a)] -> Gen a
allof :: [Gen a] -> Gen a

weightedAllof :: [(Int, Gen a)] -> Gen a

= frequency + allof

choose :: Random a => (a, a) -> Gen a
enumerate :: [a] -> Gen a

randomOrder :: Random a => (a, a) -> Gen a

= choose + enumerate

suchThatMaybe :: Gen a -> (a -> Bool) -> Gen (Maybe a)
empty :: Gen a

filterG :: (a -> Bool) -> Gen a -> Gen a

= suchThatMaybe - Maybe