QuickerCheck

Robert Krook

2000!

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in whic hthe tool w as successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

1. INTRODUCTION

T esting is by far the most commonly used approach to ensuring softw are qualit y. It is also v ery labour intensive, accoun tingfor up to 50% of the cost of softw are dev elopment. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-c hec ks, it usually works'), in practice it is still a major part of functional program development.

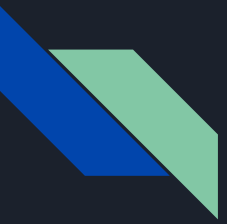
The cost of testing motivates efforts to automate it, wholly

monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically chec kable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as chec kable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], whic hcompetes surprisingly favourably with systematic methods in practice. Ho wever, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reuseable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform dis-


PROPERTY-BASED TESTING
ANY OTHER TESTING
imgflip.com

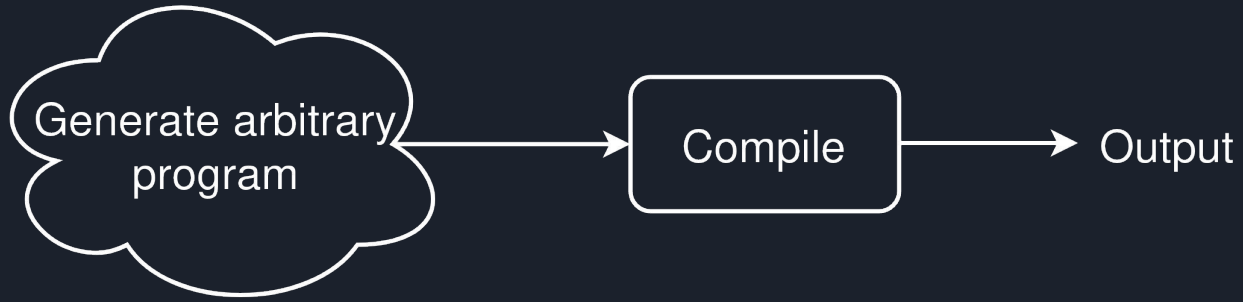


QuickCheck

Not just for simple lists and trees!

Can test larger systems, e.g compilers

QuickCheck



QuickCheck

```
        *v1 = v12;
    }
}
void proc3(i8 *v0, i16 v1, i8 v2)
{
    _STEP_;
    {
        // allocate reference;

        i16 *v3 = (i16 *) malloc(sizeof(i16));
        cell *new_cell = create_cell(++max_haddr_index, (void *) v3, I16);

        insert_cell(new_cell);
        *v3 = safe_mul_funI16(safe_mul_funI16(-15334, -14299), safe_mul_funI16(v18, 11811));
    }
    {
        i32 v4 = safe_add_funI32(856653378, -1107741951);

        {
            i32 v5 = -600099798;

            *v16 = safe_add_funI16(safe_add_funI16(30755, *v10), safe_mul_funI16(*v10, v18));
        }
        {
            i16 v6 = safe_add_funI16(safe_add_funI16(*v23, 9469), safe_mul_funI16(-3633, -26275));

            v3 = -718112451;
        }
        v2 = -19;
        v18 = safe_add_funI16(safe_add_funI16(31809, v18), safe_mul_funI16(v18, *v23));
        *v5 = safe_mul_funI16(v24, *v25);
        v21 = v17;
    }
    {
        // allocate reference;

        i32 *v5 = (i32 *) malloc(sizeof(i32));
        cell *new_cell = create_cell(++max_haddr_index, (void *) v5, I32);

        insert_cell(new_cell);
        *v5 = -94 == -66;
    }
}
void proc4(i32 v0, i16 *v1)
{
    _STEP_;
    {
```

QuickCheck

Arbitrary programs

- May be thousands of lines long
- Use global state
- Invoke procedures
- Use pointers
- Might not terminate!

QuickerCheck

instead of `quickCheck p`, `quickCheckPar p`.

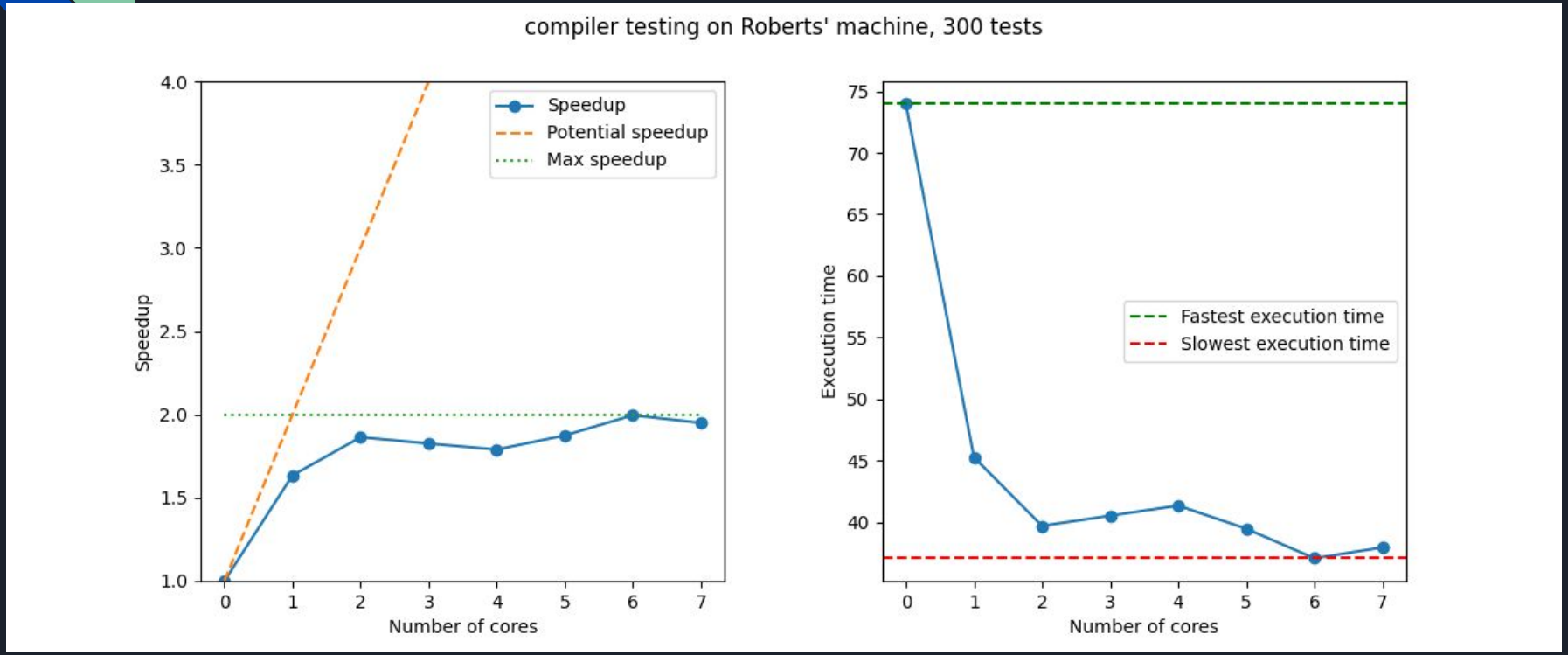
ghc-options: -threaded -rtsopts -feager-blackholing

+RTS -N

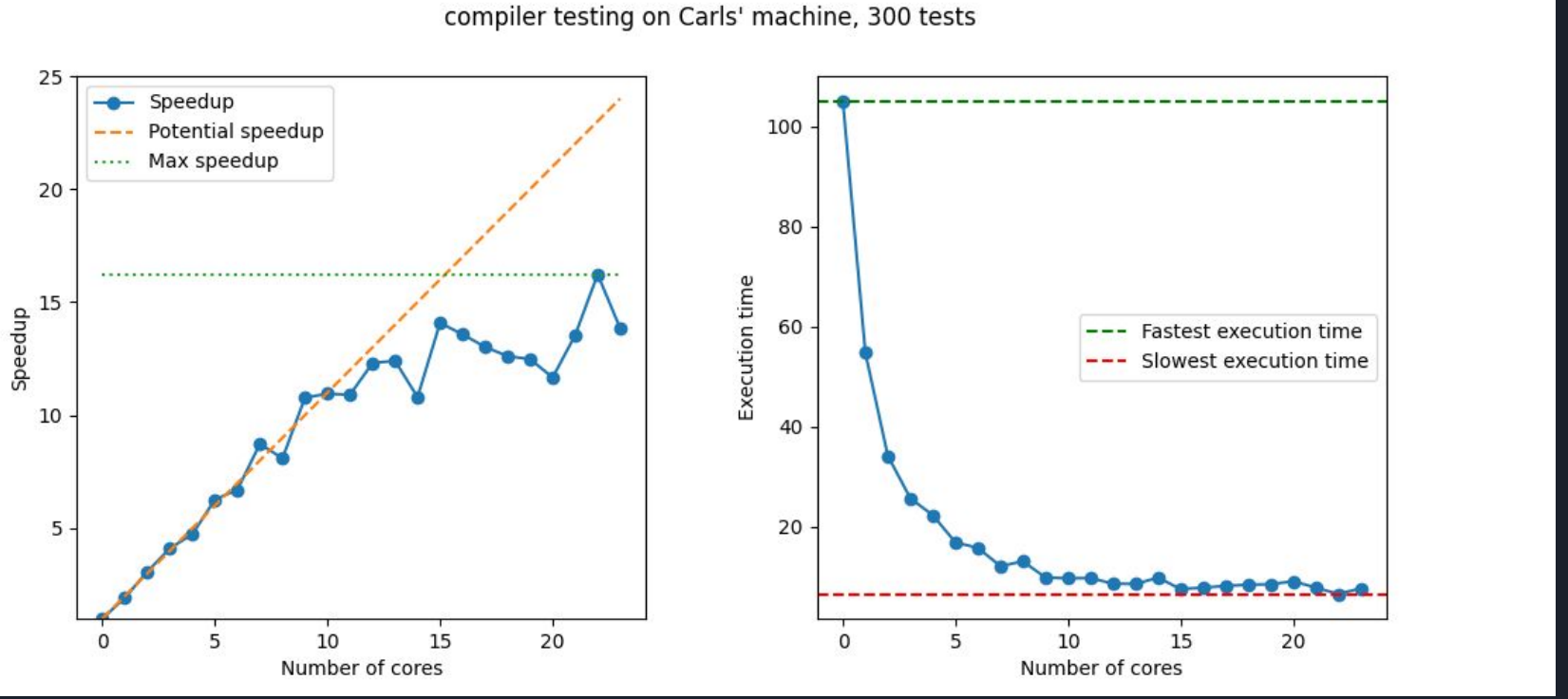
QuickerCheck

```
> cabal run …
Up to date
+++ OK, passed 300 tests
 tester 0: 39 tests
 tester 1: 38 tests
 tester 2: 39 tests
 tester 3: 39 tests
 tester 4: 34 tests
 tester 5: 37 tests
 tester 6: 38 tests
 tester 7: 36 tests
```

QuickCheck


compiler testing on Roberts' machine, 300 tests
Speedup
Potential speedup
Max speedup
Speedup
Number of cores
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
1
2
3
4
5
6
7
Fastest execution time
Slowest execution time
Execution time
Number of cores
75
70
65
60
55
50
45
40

QuickCheck


compiler testing on Carls' machine, 300 tests
Speedup
Potential speedup
Max speedup
Speedup
Number of cores
Fastest execution time
Slowest execution time
Execution time
Number of cores

QuickCheck

```
module Set (
    Set          -- abstract type of Sets
    , empty      -- :: Set a
    , singleton  -- :: a -> Set a
    , size       -- :: Set a -> Int
    , elem       -- :: Eq a => a -> Set a -> Bool
    , insert     -- :: Eq a => a -> Set a -> Set a
) where
```

QuickCheck

```
size empty =  0

forall x . size (singleton x) =  1

forall set .
 forall x .
   not (x `elem` set) => size (insert x set) = size set +   1

forall set .
 forall x .
   x `elem` set => size (insert x set) = size set

forall set .
 forall x .
   elem x (insert x set)
```

QuickCheck

```
prop_size_empty :: Bool
prop_size_empty = size empty ==0

prop_size_singleton :: Int -> Bool
prop_size_singleton x = size (singleton x) ==1

prop_size_insert_elem :: Set Int -> Int -> Property
prop_size_insert_elem set x =
 not (x `Set.elem` set) ==>
   size (insert x set) == size set +1

prop_size_insert :: Set Int -> Int -> Property
prop_size_insert set x =
 x `Set.elem` set ==>
   size (insert x set) == size set

prop_elem :: Set Int -> Int -> Bool
prop_elem set x = Set.elem x (insert x set)
```

QuickCheck

```
ghci> quickCheck $ prop_size_empty
+++ OK, passed 1 test

ghci> quickCheck $ prop_size_singleton
+++ OK, passed 100 test

ghci> quickCheck $ prop_size_insert_elem
+++ OK, passed 100 test; 17 discarded

ghci> quickCheck $ prop_size_insert
+++ OK, passed 100 test; 450 discarded

ghci> quickCheck $ prop_elem
+++ OK, passed 100 test
```

QuickCheck

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf deriving (Show, Eq)

genTree :: Arbitrary a => Gen (Tree a)
genTree = frequency [ (1, return Leaf)
                    , (2, do elem  <- arbitrary
                             left  <- genTree
                             right <- genTree
                             return $ Node elem left right)
                    ]
```

QuickCheck

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf deriving (Show, Eq)

genTree :: Arbitrary a => Gen (Tree a)
genTree = sized sizedTree

sizedTree :: Arbitrary a => Int -> Gen (Tree a)
sizedTree 0 = return Leaf
sizedTree n = do
   elem  <- arbitrary
   left  <- sizedTree (n `div` 2)
   right <- sizedTree (n `div` 2)
   return $ Node elem left right
```

QuickCheck

```
Node 10 (Node (-24) (Node (-30) (Node (-20) (Node (-26) (Node 11 (Node (-1) Leaf
Leaf) (Node (-23) Leaf Leaf)) (Node (-29) (Node (-12) Leaf Leaf) (Node (-24)
Leaf Leaf))) (Node 16 (Node (-4) (Node 26 Leaf Leaf) (Node (-16) Leaf Leaf))
(Node (-28) (Node 17 Leaf Leaf) (Node (-29) Leaf Leaf)))) (Node (-2) (Node 4
(Node 26 (Node (-14) Leaf Leaf) (Node 23 Leaf Leaf)) (Node (-25) (Node 25 Leaf
Leaf) (Node (-22) Leaf Leaf))) (Node (-18) (Node 16 (Node (-11) Leaf Leaf) (Node
(-21) Leaf Leaf)) (Node 27 (Node (-17) Leaf Leaf) (Node (-9) Leaf Leaf)))))
(Node (-8) (Node (-19) (Node 13 (Node 0 (Node (-4) Leaf Leaf) (Node 19 Leaf
Leaf)) (Node (-25) (Node (-22) Leaf Leaf) (Node (-1) Leaf Leaf))) (Node 23 (Node
(-24) (Node 17 Leaf Leaf) (Node 1 Leaf Leaf)) (Node 7 (Node (-24) Leaf Leaf)
(Node 5 Leaf Leaf)))) (Node 12 (Node 6 (Node 2 (Node 8 Leaf Leaf) (Node 3 Leaf
Leaf)) (Node (-6) (Node 4 Leaf Leaf) (Node 10 Leaf Leaf))) (Node 12 (Node 1
(Node (-3) Leaf Leaf) (Node (-8) Leaf Leaf)) (Node (-5) (Node (-22) Leaf Leaf)
(Node 24 Leaf Leaf))))))) (Node (-6) (Node 6 (Node 11 (Node 23 (Node (-14) (Node
(-27) Leaf L …
```

QuickCheck

```
shrinkTree :: Tree a -> [Tree a]
shrinkTree Leaf = []
shrinkTree (Node _ l r) = [l,r]
```

```
Node 0 Leaf Leaf
```

The testing loop

```
data State = MkState
 { terminal                 :: Terminal
 , maxSuccessTests          :: Int
 , maxDiscardedRatio        :: Int
 , coverageConfidence       :: Maybe Confidence
 , computeSize              :: Int -> Int -> Int
 , numTotMaxShrinks         :: !Int
 , numSuccessTests          :: !Int
 , numDiscardedTests        :: !Int
 , numRecentlyDiscardedTests :: !Int
 , stlabels                   :: !(Map [String] Int)
 , stclasses                  :: !(Map String Int)
 , sttables                   :: !(Map String (Map String Int))
 , strequiredCoverage         :: !(Map (Maybe String, String) Double)
 , expected                 :: !Bool
 , randomSeed               :: !QCGen

 , numSuccessShrinks        :: !Int
 , numTryShrinks            :: !Int
 , numTotTryShrinks         :: !Int
 }
```

The testing loop

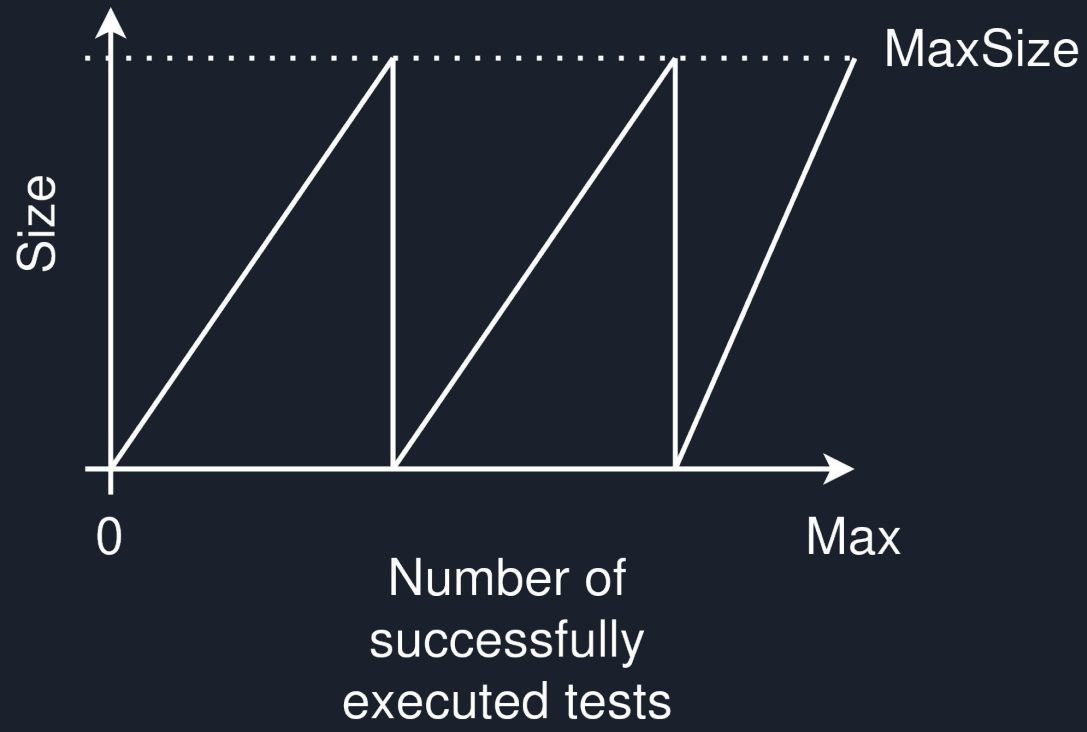
```
testloop :: State -> Property -> IO ()
testloop st prop = do
   (st', seed) <- fetchSeed st
   size        <- computeSize st
   result      <- runProperty prop seed size
   printTheStatistics
   inspectResultAndMaybeRecurse result st’
```

How to parallelize this?

One problem: size computation!

```
prop_list :: [Int] -> Bool
prop_list xs = not (null xs) ==> <the property>

data Gen a = Gen (Int -> Seed -> a)

listOf :: Gen a -> Gen [a]
listOf gen = sized $ \n ->
 do k <- chooseInt (0,n)
    vectorOf k gen
```

One problem: size computation!



First try

```
size :: Int -> Int
size numsuccess = numsuccess `mod` <max size>

size 0 => []
size 1 => [], forall x. [x]
size 2 => [], forall x. [x], forall x. forall y. [x,y]
...
```

One problem: size computation!

```
prop_list :: [Int] -> Bool
prop_list xs = not (null xs) ==> <the property>

size :: Int -> Int
size numsuccess = numsuccess `mod` <max size>
```

One problem: size computation!

```
prop_list :: [Int] -> Bool
prop_list xs = not (null xs) ==> <the property>

size :: Int -> Int -> Int
size numsuccess numrecentlydiscarded =
 (numsuccess + numrecentlydiscarded `div` 10) `mod` <max size>
```

One problem: size computation!

[t1 t2 t3 t4 t5 t6 t7 t8 t9 t10]

One problem: size computation!

[**t1** t2 **t3** t4 **t5** t6 t7 t8 **t9** t10]

[t1 **t2** t3 **t4** **t5** **t6** t7 t8 t9 **t10**]

[t1 t2 **t3** t4 t5 **t6** **t7** t8 t9 t10]

[t1 t2 t3 **t4** t5 t6 t7 **t8** t9 t10]

The testing loop

```
setup :: Int -> Property -> IO ()
setup numworkers prop = do
   let states = createStates numworkers
       testers = map (\st -> testloop st prop) states
   runEval $ parList rdeepseq testers
```

Should we create sparks?

The testing loop

```
setup :: Int -> Property -> IO ()
setup numworkers prop = do
   let states  = createStates numworkers
       parcomp =
         do ivars  = mapM (\st -> spawnP (testloop st prop)) states
            map get ivars
   runPar parcomp
```

Should we use the par monad?

Parallelising the testing loop

```
forkIO      :: IO a -> IO ThreadId
killThread :: ThreadId -> IO ()
throwTo     :: Exception e => ThreadId -> e -> IO ()

data MVar a
newEmptyMVar :: IO (MVar a)
putMVar       :: MVar a -> a -> IO ()
takeMVar      :: MVar a -> IO a
modifyMVar    :: MVar a -> (a -> IO (a, b)) -> IO b
```

Parallelising the testing loop

```
-- parallelism
, numConcurrent                 :: Int
, numSuccessOffset              :: !Int
, myId                          :: !Int
, signalGaveUp                  :: IO ()
, signalTerminating             :: IO ()
, signalFailureFound            :: State -> QCGen -> Result
                                -> [Rose Result] -> Int -> IO ()
, shouldUpdateAfterWithMaxSuccess  :: Bool
, stsizeStrategy                :: SizeStrategy
, testBudget                    :: IORef Int
, stealTests                    :: IO (Maybe Int)
, discardBudget                 :: IORef Int
, stealDiscards                 :: IO (Maybe Int)
```

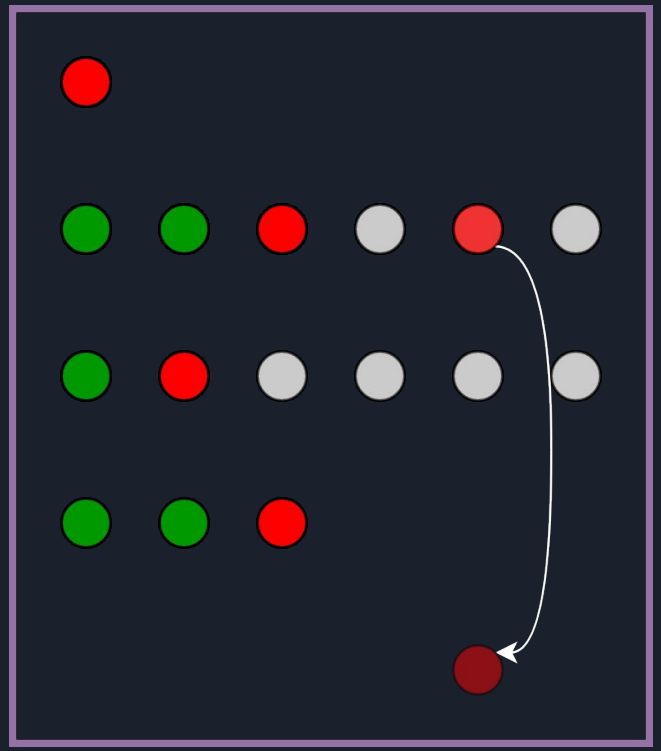
Parallelising the testing loop

```
quickCheckInternal :: Testable a => Args -> a -> IO ()
quickCheckInternal a p = do
 rnd <- newStdGen
 let initialSeeds = < split rnd n times >

 testbudgets    <- < create MVars >
 discardbudgets <- < create MVars >
 states         <- < create MVars >

 signal     <- < create MVars >

 < initialize the MVars >

 tids <- mapM (\vst -> forkIO $ testLoop vst True (property p)) states

 s <- takeMVar signal
 mapM_ killThread tids

 sts <- mapM readMVar states
 < print result >
```

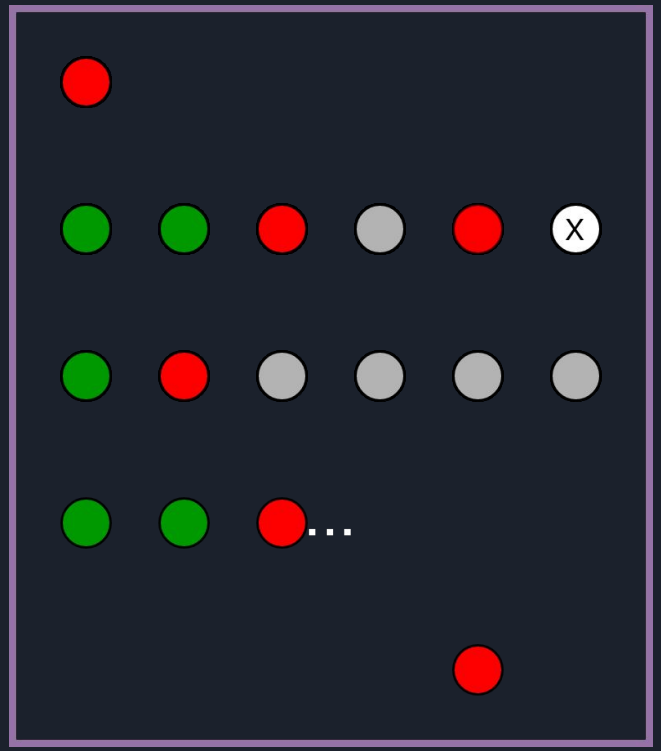
Parallelising the testing loop

```
testLoop :: MVar State -> Bool -> Property -> IO ()
testLoop vst False f = do
  if < we should run one more >
    then testLoop vst True f
    else signalTerminating st
testLoop vst True f = do
  (s1,s2) <- < split the seed from the state >
  res <- < run the property with the seed and a size >
 case ok res of
   Discarded -> do
     if < we can discard more >
       then testLoop vst True f
       else signalGaveUp st'
   Ok  -> testLoop vst False f
   Falsified -> signalFailureFound
```

Shrinking



Parshrink



Parshrink - how are workers restarted?

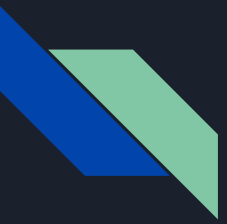
Ideal to start workers only once, less chance of resource misuse

Parshrink - how are workers restarted?

```
interruptShrinkers :: [ThreadId] -> IO ()
interruptShrinkers tids = mapM_ (\tid -> throwTo tid QCInterrupt) tids
```

Parshrink - how are workers restarted?

```
spawnWorkers num jobs stats signal =
  sequence $ replicate num $ forkIO $ defHandler $ worker jobs stats signal
  where
    defHandler :: IO () -> IO ()
    defHandler ioa = do
      r <- try ioa -- :: Exception e => IO a -> IO (Either e a)
      case r of
        Right a -> pure a
        Left QCInterrupt -> defHandler ioa
```



Graceful

Currently experimenting/designing a graceful combinator. Still slightly fluid exactly what it will look like. Mention the problem it is intended to solve (cleanup after unexpected termination) and potential designs.

Have a (seemingly) working combinator, but it clutters the code and leads to code duplication.

Verse

Confluent rulesets

The Verse Calculus: a Core Calculus for Functional Logic Programming

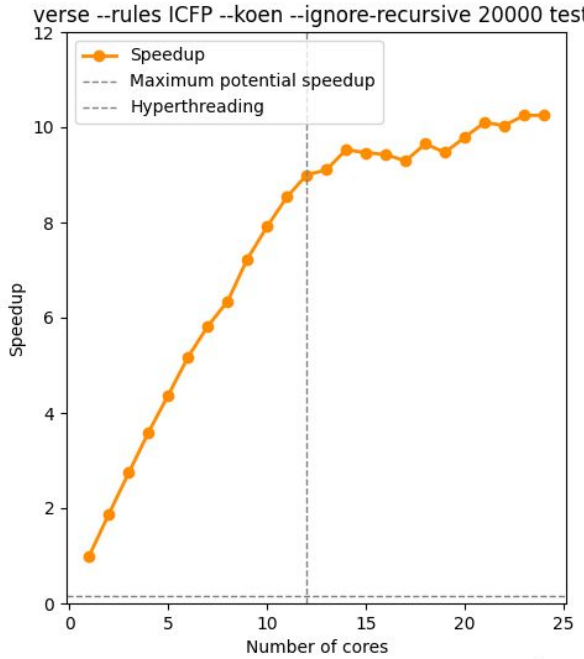
LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER
KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
GUY STEELE, Oracle Labs, USA
TIM SWEENEY, Epic Games, USA

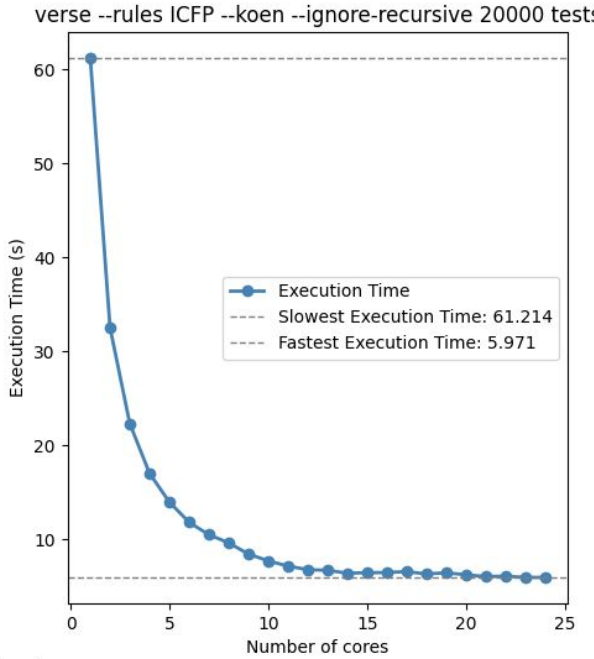
Functional logic languages have a rich literature, but it is tricky to give them a satisfying semantics. In this paper we describe the Verse calculus, $\mathcal{VC}$, a new core calculus for functional logic programming. Our main contribution is to equip $\mathcal{VC}$ with a small-step rewrite semantics, so that we can reason about a $\mathcal{VC}$ program in the same way as one does with lambda calculus; that is, by applying successive rewrites to it. We also show that the rewrite system is confluent.

Additional Key Words and Phrases: confluence, declarative programming, functional programming, lambda calculus, logic programming, rewrite rules, skew confluence, unification, Verse calculus, Verse language

1 INTRODUCTION

Functional logic programming languages add expressiveness to functional programming by introducing logical variables, equality constraints among those variables, and choice to allow multiple





Graph generated by ChatGPT

Gzip | GUnzip identity

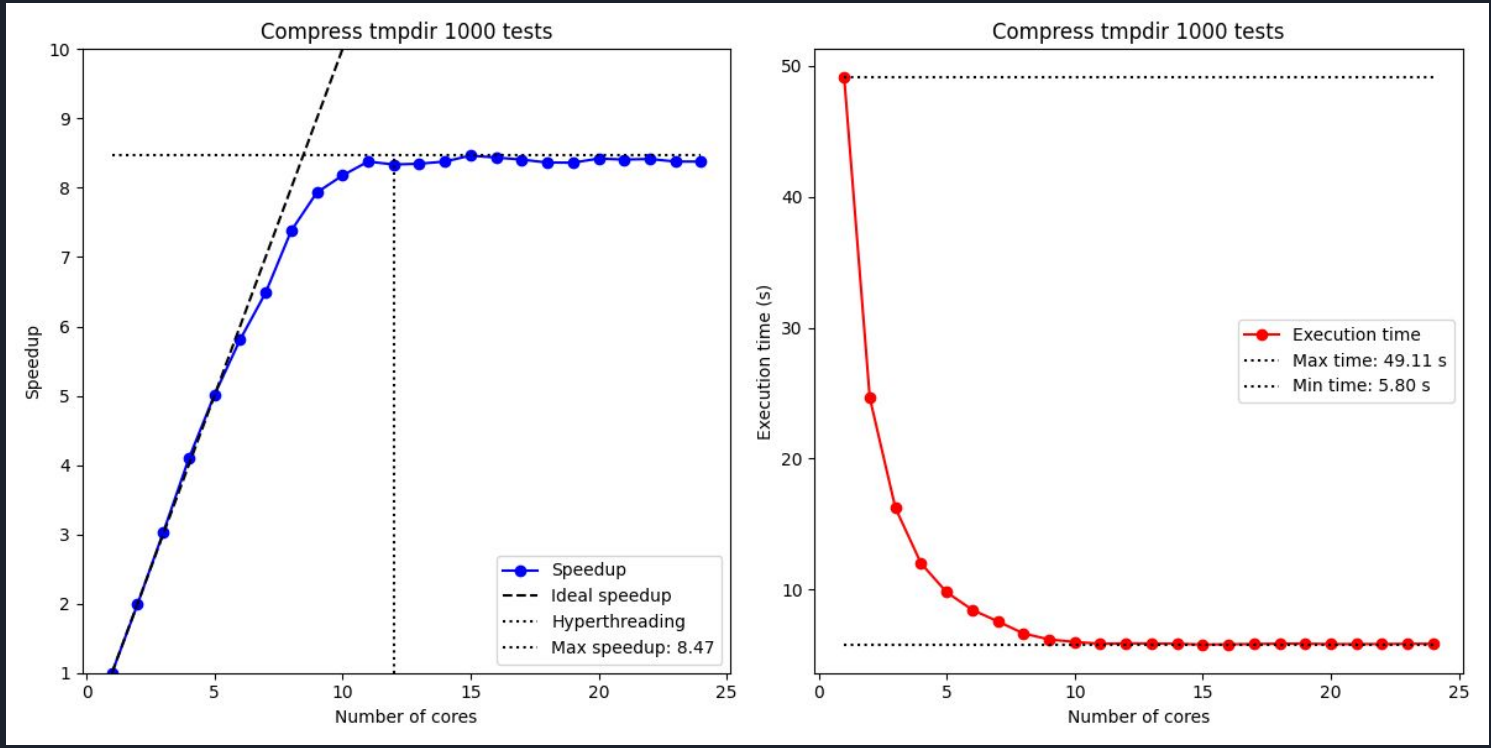
GZip followed by GUnzip = Identity?

Naive gzip-gunzip

```
naive :: String -> IO String
naive input = do
 writeFile "input.txt" input
 readProcess "gzip" ["input.txt"] ""
 readProcess "gunzip" ["input.txt.gz"] ""
 contents <- readProcess "cat" ["input.txt"] ""
 readProcess "rm" ["input.txt"] ""
 return contents
```

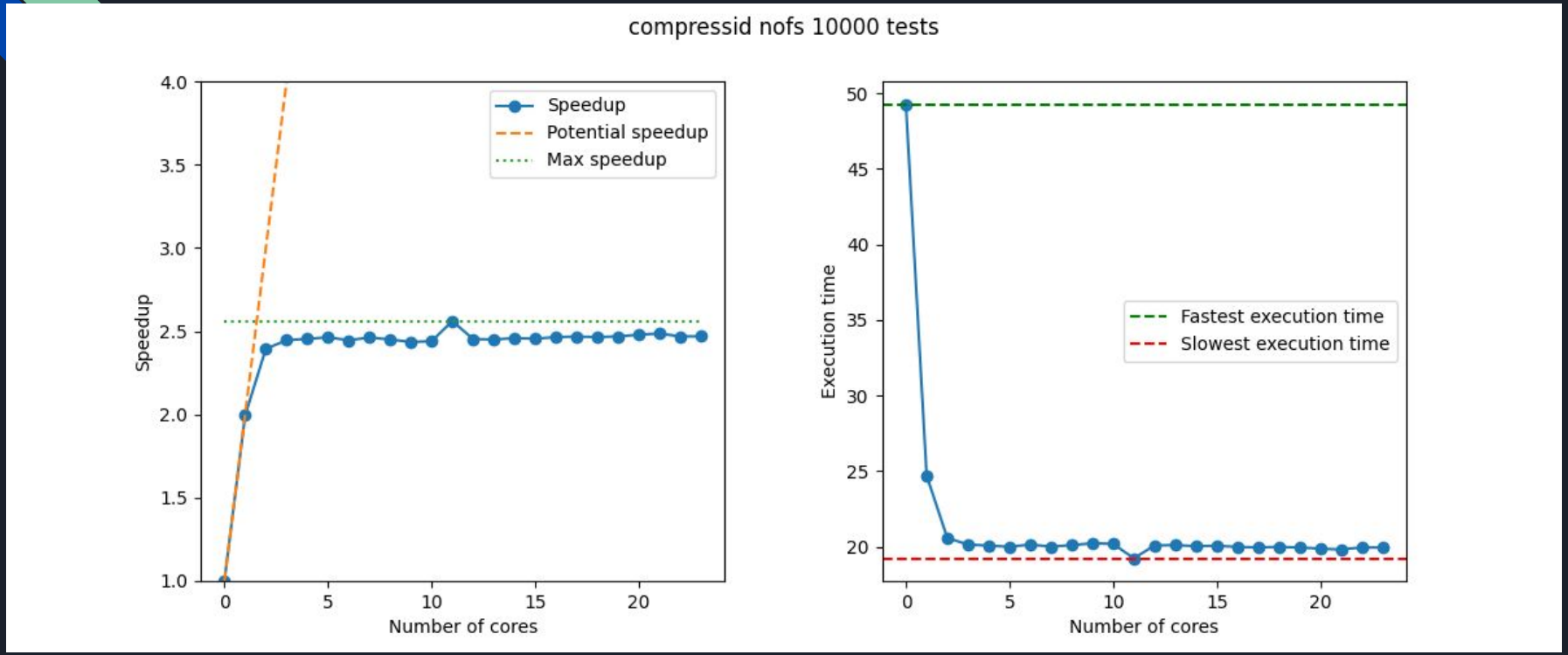
With temporary directory

```
withtmp :: FilePath -> String -> IO String
withtmp root input = do
 writeFile (root // "input.txt") input
 readProcess "gzip" [root // "input.txt"] ""
 readProcess "gunzip" [root // "input.txt.gz"] ""
 contents <- readProcess "cat" [root // "input.txt"] ""
 readProcess "rm" [root // "input.txt"] ""
 return contents
```

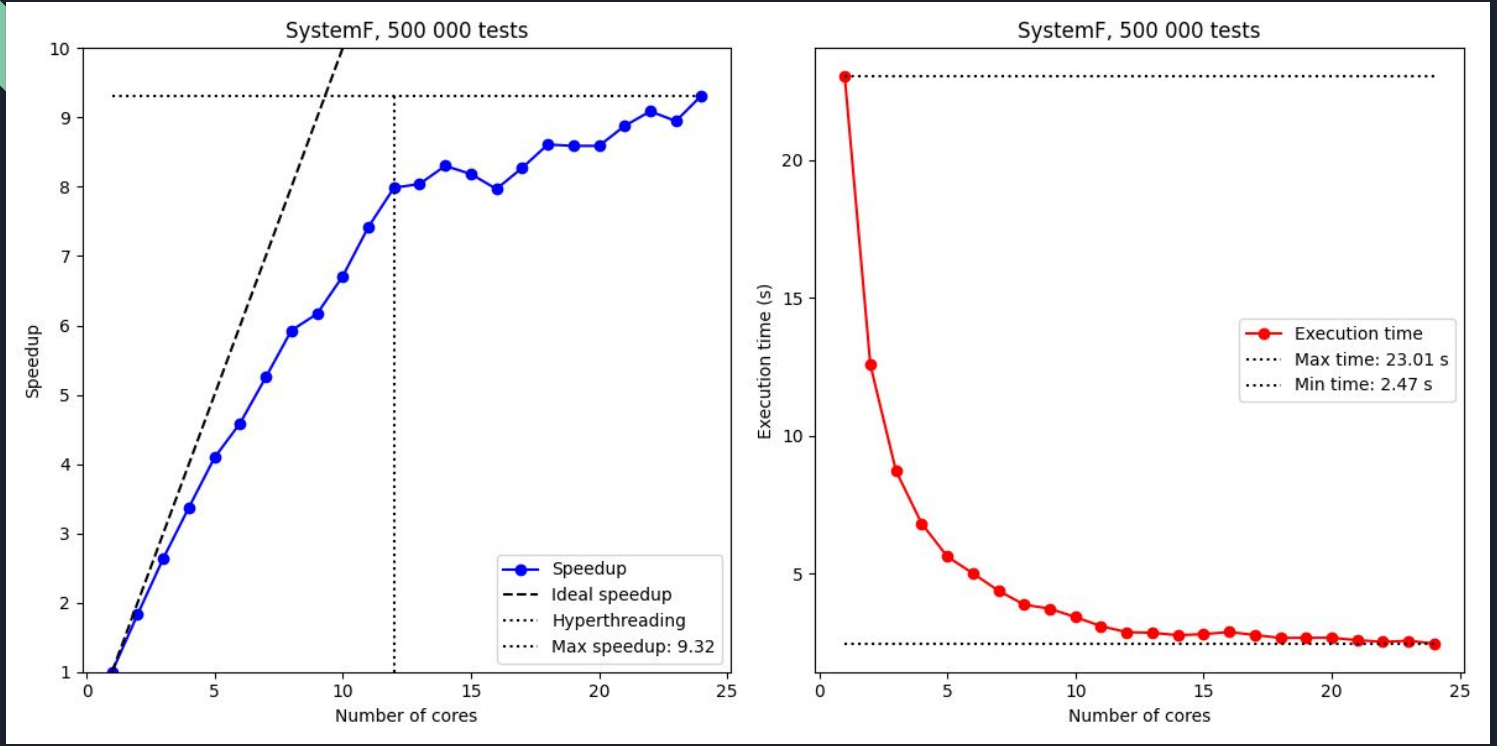

Compress tmpdir 1000 tests
Speedup
Number of cores
Speedup
Ideal speedup
Hyperthreading
Max speedup: 8.47
Compress tmpdir 1000 tests
Execution time (s)
Number of cores
Execution time
Max time: 49.11 s
Min time: 5.80 s

Without filesystem

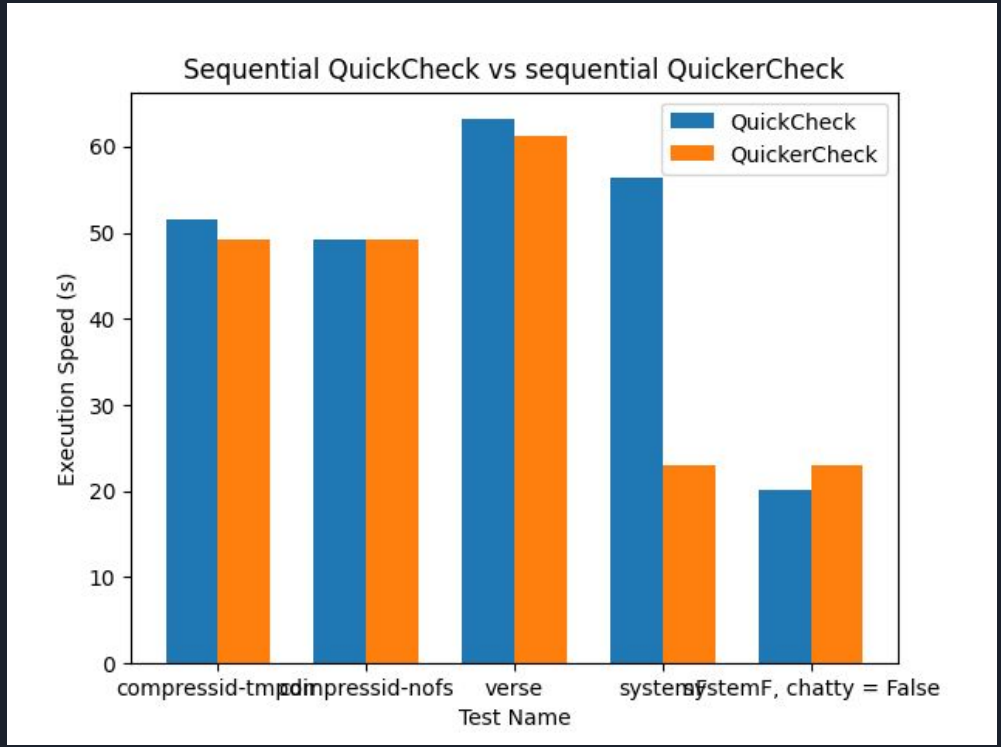
```
noFS :: String -> IO String
noFS input = do
 (Just hin, Just hout, _, _) <-
   createProcess
     (proc "gzip" [])
       { std_in = CreatePipe,
         std_out = CreatePipe
       }
 hPutStr hin input
 (_, Just hfin, _, _) <-
   createProcess
     (proc "gunzip" [])
       { std_in = UseHandle hout,
         std_out = CreatePipe
       }
 hClose hin
 hGetContents hfin
```


compressid nofs 10000 tests
Speedup
Potential speedup
Max speedup
Speedup
Number of cores
Fastest execution time
Slowest execution time
Execution time
Number of cores

SystemF


SystemF, 500 000 tests
Speedup
Number of cores
Speedup
Ideal speedup
Hyperthreading
Max speedup: 9.32
SystemF, 500 000 tests
Execution time (s)
Number of cores
Execution time
Max time: 23.01 s
Min time: 2.47 s

QuickCheck vs QuickerCheck



Links

https://github.com/Rewbert/quickcheck – my fork! pls hack, no judge

https://github.com/nick8325/quickcheck – actual QuickCheck

https://dl.acm.org/doi/abs/10.1145/351240.351266 – The 2000 paper (Mary wanted me to remind you that everything you see in lectures is examinable content, so perhaps taking a look at this paper is a good idea)

https://github.com/Rewbert/quickcheck-v1 – ‘Little’ QuickCheck!

Master thesis?

I am especially interested in the shrinking part. Is there some better way to select the next test candidate?