Performance evaluation of various functional programming styles

Jan Pustelnik

GFT Poland,
ul. Sterlinga 8a, Lodz, Poland

LambdaDays 2015, 2015-02-26

Outline

Intro

This talk is devoted to analysis of how various functional programming styles impact the performance of running programs. The examples presented will be mainly in Haskell, Scala and (!) C++

Hardware

Windows 7 machine. The same for all the tests.

Software – Scala

Scala build config for JMH by Konrad Malawski – `https://github.com/ktoso/sbt-jmh`.
Scala 2.10.4, Java 1.7.0_71 and JMH 1.5.

Software – Haskell

Criterion library by Bryan O'Sullivan –
`http://www.serpentine.com/criterion/tutorial.html`
GHC 7.8.3, code was compiled with `-O` option with the standard backend.

Software – C++

Wall clock time, MinGW 4.8.1, GCC version 4.6.3 with -O3 option and run on Windows (yes, it might make a difference).

All fine, but give me the c0de!

Sure, please go to `https://github.com/gosubpl` for the *Source*.

MergeSort

Mergesort is probably the oldest sorting algorithm for computers. Attributed by Knuth to von Neumann around 1945.

Pros:

- Is obviously correct
- Has a nice functional implementation, in fact difficult to implement with mutation
- Works well with lists, tapes, other sequential data structures/media
- $O(n \times log(n))$ asymptotic – best known for a sorting algorithm that uses comparisons

Cons:

- Maybe slower than quicksort, that will be discovered some fifteen years later

Sources for examples

All programs used as examples are either Public Domain, MIT or Creative Commons by attribution.

- Haskell - `http://en.literateprograms.org/`
- C++ - look for inspiration at `http://rosettacode.org/wiki/Sorting_algorithms`
- Scala - *Programming in Scala* example improved by Daniel Sobral – `http://stackoverflow.com/questions/2201472/merge-sort-from-programming-scala-causes-stack-overflo`

Top-Down mergesort in Haskell

```
mergesort :: (a -> a -> Bool) -> [a] -> [a]
mergesort pred []   = []
mergesort pred [x]  = [x]
mergesort pred xs = merge pred (mergesort pred xs1) (
    mergesort pred xs2)
  where
    (xs1,xs2) = split xs
```

Top-Down mergesort in Haskell, cntd.

```haskell
split :: [a] -> ([a],[a])
split xs = go xs xs where
  go (x:xs) (_:_:zs) = (x:us,vs) where (us,vs)=go xs zs
  go    xs    _      = ([],xs)

merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
merge pred xs []         = xs
merge pred [] ys         = ys
merge pred (x:xs) (y:ys)
     | pred x y = x: merge pred xs (y:ys)
     | otherwise = y: merge pred (x:xs) ys
```

Bottom-Up mergesort in Haskell

```haskell
mergesort pred [] = []
mergesort pred xs = go [[x] | x <- xs]
  where
    go xs@(_:_:_)  = go (pairs xs)
    go [xs]        = xs
    pairs (x:y:xs) = merge pred x y : pairs xs
    pairs xs       = xs
```

You can also do it in C++

It even has two built-in functions for merging: `std::merge` and `std::inplace_merge`

```
template<typename RandomAccessIterator, typename Order>
 void mergesort(RandomAccessIterator first,
     RandomAccessIterator last, Order order)
{
  if (last - first > 1)
  {
    RandomAccessIterator middle = first + (last - first
        ) / 2;
    mergesort(first, middle, order);
    mergesort(middle, last, order);
    std::inplace_merge(first, middle, last, order);
  }
}
```

You can also do it in C++ – contd.

```
template<typename RandomAccessIterator>
 void mergesort(RandomAccessIterator first,
     RandomAccessIterator last)
{
  mergesort(first, last, std::less<typename std::
      iterator_traits<RandomAccessIterator>::value_type
      >());
}
```

And in Scala

```
def msort[T](less: (T, T) => Boolean)(xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T], acc: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys.reverse ::: acc
      case (_, Nil) => xs.reverse ::: acc
      case (x :: xs1, y :: ys1) =>
        if (less(x, y)) merge(xs1, ys, x :: acc)
        else merge(xs, ys1, y :: acc)
    }
  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
    merge(msort(less)(ys), msort(less)(zs), Nil).reverse
  }
}
```

And the performance crown goes to . . .

All times in the table below are in milliseconds

Program / Input size	20k	200k	500k	1m	2m
Haskell Bottom-Up	28	562.2	1948	4580	–
Haskell Top-Down	33.5	645.6	2225	5213	–
C++	6.9	–	173.4	355.6	748.8
Scala	9.5	–	411	1035.5	–

The table above gives us some point of reference – no thrills, really.

QuickSort

QuickSort is quick. Discovered by Hoare around 1960. Proven to work around 1969. In the meantime Hoare was busy inventing Hoare logic to use it to prove that quicksort is not only quick but also sorts.

QuickSort – contd.

Pros:

- Is quick
- Apart from imperative mutating implementation has a nice functional one . . .

Cons:

- Who knows what it does
- Demands mutation!
- Does not work well with lists, tapes, other sequential data structures/media – requires random access data structure
- $O(n \times log(n))$ asymptotic – but $O(n^2)$ pessimistic
- . . . but unfortunately that functional implementation is not quick!

As previously, credits

- Haskell Functional – http://en.literateprograms.org/ but almost identical programs can be found in [2] or [3]
- Haskell Imperative – http://stackoverflow.com/questions/11481675/using-vectors-for-performance-improvement-in-haskell
- C++ – could not find this anywhere, had to code it, but it was easy, doh!
- Scala – *Scala by Example* by Martin Odersky, except for the ST monadic example – this is from https://github.com/fpinscala/fpinscala/blob/master/exercises/src/main/scala/fpinscala/localeffects/LocalEffects.scala (solutions to *Functional programming in Scala* by Paul Chiusano and Rúnar Bjarnason).

Standard mutable quicksort in Scala

Standard, *ugly* and i-am-not-quite-sure-it-is-working implementation of the quicksort algorithm in a language that supports direct mutation – Scala

```
def sortQuickTraditional(xs: Array[Int]): Array[Int]
    = {
  def swap(i: Int, j: Int) {
    val t = xs(i)
    xs(i) = xs(j)
    xs(j) = t
  }
```

Standard mutable quicksort in Scala – contd.

```
    def sort1(l: Int, r: Int) {
      val pivot = xs((l + r) / 2)
      var i = l
      var j = r
      while (i <= j) {
        while (xs(i) < pivot) i += 1
        while (xs(j) > pivot) j -= 1
        if (i <= j) {
          swap(i, j)
          i += 1
          j -= 1
        }
      }
      if (l < j) sort1(l, j)
      if (j < r) sort1(i, r)
    }
    sort1(0, xs.length - 1)
    xs
  }
```

Standard mutable quicksort in Haskell

One has to use the ST monad.

```
stuquick :: [Int] -> [Int]
stuquick [] = []
stuquick xs = runST (do
    let !len = length xs
    arr <- newListArray (0,len-1) xs
    myqsort arr 0 (len-1)
    let pick acc i
            | i < 0     = return acc
            | otherwise = do
                !v <- unsafeRead arr i
                pick (v:acc) (i-1)
    pick [] (len-1))
```

Standard mutable quicksort in Haskell – contd.

```
myqsort :: STUArray s Int Int -> Int -> Int -> ST s ()
myqsort a lo hi
    | lo < hi    = do
        let lscan p h i
                | i < h = do
                    v <- unsafeRead a i
                    if p < v then return i else lscan p
                        h (i+1)
                | otherwise = return i
            rscan p l i
                | l < i = do
                    v <- unsafeRead a i
                    if v < p then return i else rscan p
                        l (i-1)
                | otherwise = return i
```

Standard mutable quicksort in Haskell – contd.

```
            swap i j = do
                v <- unsafeRead a i
                unsafeRead a j >>= unsafeWrite a i
                unsafeWrite a j v
            sloop p l h
                | l < h = do
                    l1 <- lscan p h l
                    h1 <- rscan p l1 h
                    if (l1 < h1) then (swap l1 h1 >>
                        sloop p l1 h1) else return l1
                | otherwise = return l
        piv <- unsafeRead a hi
        i <- sloop piv lo hi
        swap i hi
        myqsort a lo (i-1)
        myqsort a (i+1) hi
    | otherwise = return ()
```

ST monad mutable quicksort Scala

```
def sortQuickMonadicST(xi: Array[Int]): Array[Int] = {
  def invert(x: (Int, Int)): (Int, Int) = (x._2, x._1)
  def identify(x: Int, y: Int): Int = y
  val arrLen = xi.length
  val xiz = xi.zipWithIndex
  val xizi = (xiz map invert).toList

  type ForallST[A] = Forall[({type λ[S] = ST[S, A]})#λ]
  def noop[S] = ST[S, Unit](())
```

ST monad mutable quicksort Scala – contd.

```
def swap[S](a: STArray[S, Int], i: Int, j: Int): ST[
    S, Unit] = for {
  x <- a.read(i)
  y <- a.read(j)
  _ <- a.write(i, y)
  _ <- a.write(j, x)
} yield ()
```

ST monad mutable quicksort Scala – contd.

```
def partition[S](a: STArray[S, Int], l: Int, r: Int,
    pivot: Int): ST[S, Int] = for {
  vp <- a.read(pivot)
  _ <- swap(a, pivot, r)
  j <- newVar(l)
  _ <- (l until r).foldLeft(noop[S])((s, i) => for
      {
    _ <- s
    vi <- a.read(i)
    _ <- if (vi < vp) (for {
      vj <- j.read
      _ <- swap(a, i, vj)
      _ <- j.write(vj + 1)
    } yield ())
    else noop[S]
  } yield ())
  x <- j.read
  _ <- swap(a, x, r)
```

ST monad mutable quicksort Scala – contd.

```
def qs[S](a: STArray[S, Int], l: Int, r: Int): ST[S, Unit] = if (l < r) for {
  pi <- partition(a, l, r, l + (r - l) / 2)
  _ <- qs(a, l, pi - 1)
  _ <- qs(a, pi + 1, r)
} yield ()
else noop[S]

def e1[S] = for {
  arr <- newArr[S, Int](arrLen, 0)
  _ <- arr.fill(identify, xizi)
  _ <- qs(arr, 0, arr.size - 1)
  sorted <- arr.freeze
} yield sorted
```

ST monad mutable quicksort Scala – contd.

```
    runST(new ForallST[ImmutableArray[Int]] {
      def apply[S] = e1[S]
    }).toArray
  }
```

It might have not been the easiest to comprehend pieces of code, but maybe it runs fast? Who knows . . .

Pure functional quicksort in Haskell

Short, simple and easy to understand.

```
qsort1 []     = []
qsort1 (p:xs) = qsort1 lesser ++ [p] ++ qsort1 greater
    where
        lesser  = [ y | y <- xs, y < p ]
        greater = [ y | y <- xs, y >= p ]
```

Pure functional quicksort in Scala

As easy as in Haskell :) You can also replace `Array` with `ArrayBuffer` or `Vector`[1]

```scala
def sortFunctional(xs: Array[Int]): Array[Int] = {
    if (xs.length <= 1) xs
    else {
      val pivot = xs(xs.length / 2)
      Array.concat(sortFunctional(xs filter (pivot >)),
           xs filter (pivot ==), sortFunctional(xs
          filter (pivot <)))
    }
  }
```

[1] `Vector` is immutable and should behave like an `Array` plus have nice amortised complexities.

Pure functional quicksort in C++

Maybe even easier than in Haskell and maybe not. But maybe faster?

```
vector<int> funqsort(vector<int> v) {
  if (v.size() > 1) {
    int pivot = v[0];
  vector<int> lesser; vector<int> greater;
  std::copy_if(v.begin()+1, v.end(), std::back_inserter
      (lesser), std::bind2nd(std::less<int>(), pivot));

  std::copy_if(v.begin()+1, v.end(), std::back_inserter
      (greater), std::not1(std::bind2nd(std::less<int
      >(), pivot)));

  vector<int> result;
```

Pure functional quicksort in C++ – contd.

```
    vector<int> fql = funqsort(lesser);
    vector<int> fqg = funqsort(greater);
  std::copy(fql.begin(), fql.end(), std::back_inserter(
      result));

  result.push_back(pivot);

  std::copy(fqg.begin(), fqg.end(), std::back_inserter(
      result));

  return result;
  } else { return v; }
}
```

Ok, so tell me about the performance . . .

All times in milliseconds

Program / Input size	20k	200k	500k	1m	2m
MergeSort Scala	9.5	–	411	1035.5	–
Haskell Imperative	4.2	61.5	161.2	347.7	–
Haskell Functional	18.9	360.4	1072	2747	–
Haskell Fnct. saving	6.9	31.8	515.4	1280	–
StdLibSort C++	1.0	–	29.1	58.4	117.1
C++ Functional	44.5	–	1626	3751	8881
Scala Imperative	1.7	–	54.3	112.7	231.8
Scala Imp. Vector	17.1	–	816.5	1790	4648
Scala Functional	33.46	–	1058	2192	4726
Scala Funct. Vector	22.9	–	865.7	1981	4375
Scala ST Monad	138.3	–	5535	13428	35290

HeapSort

HeapSort is very easy to implement if you have a Heap or PriorityQueue data structure at hand. Benchmarking it gives you a chance to compare performance of various PriorityQueue implementations.

Pros:

- Very simple to implement if you have a PriorityQueue
- $O(n \times log(n))$ asymptotic

Cons:

- If your library supports PriorityQueues, probably it has a standard sorting algorithm too
- Slower than QuickSort and not stable like MergeSort

Sources for examples

- Haskell – `Data.Heap` from the `heap` package (leftist trees from Okasaki by Edward Kmett) and `Data.PQueue` from the `pqueue` package (binomial heaps).
- Scala – one imperative implementation using `mutable.PriorityQueue` from the standard library, and one purely functional implementation using `scalaz.Heap` that implements leftist trees (also by Edward Kmett)

Heap sorts in Haskell

```haskell
import qualified Data.Heap as DH -- from heap package
import qualified Data.PQueue.Min as DPQMin -- from
    pqueue package

-- heapsort
hsort :: [Int] -> [Int]
hsort xs = DH.toAscList (DH.fromList xs :: DH.MinHeap
    Int)

-- another heapsort
hpqsort :: [Int] -> [Int]
hpqsort xs = DPQMin.toAscList (DPQMin.fromList xs)
```

Mutable heap sort in Scala

```
// a bit mutable heapsort using the StdLib mutable.
    PriorityQueue
def sortHeapPQ(xs: Array[Int]): Array[Int] = {
  val ord = implicitly[Ordering[Int]].reverse
  val lst = ListBuffer[Int]()
  val pq: PriorityQueue[Int] = new PriorityQueue[Int
     ]()(ord) ++ xs
  while (pq.size > 0) {
    val elem = pq.dequeue()
    lst += elem
  }
  lst.toArray
}
```

Pure immutable heap sort in Scala

```scala
def sortHeapLeftist(xs: Array[Int]): Array[Int] = {
  import scalaz._
  import Scalaz._
  def poorMansHeapFold(h: Heap[Int]): List[Int] = {
    def heapFoldLeftAccum(accum: List[Int], h: Heap[
        Int]): List[Int] = {
      if (h.size == 0) {
        accum.reverse
      } else {
        val (head, tail) = h.uncons.get
        heapFoldLeftAccum(head :: accum, tail)
      }
    }
    heapFoldLeftAccum(Nil, h)
  }
  poorMansHeapFold(Heap.fromData(xs.toList)).toArray
}
```

But, I want the numbers!

All times in milliseconds

Program / Input size	20k	200k	500k	1m	2m
MergeSort Scala	9.5	–	411	1035.5	–
Haskell QSort Funct.	18.9	360.4	1072	2747	–
Scala QSort Imp.	1.7	–	54.3	112.7	231.8
Scala PriorityQueue	17.1	–	330.8	834.1	–
Scala ScalazHeap	135.5	–	6778	15363	–
Scala QuickST	138.3	–	5535	13428	35290
Haskell PQueue	24.8	506.4	1735	4108	–
Haskell Heap	40.3	740.3	2475	5606	–

QuickCheck

QuickSort is quick. Discovered by Hoare around 1960. Proven to work around 1969. In the meantime Hoare was busy inventing Hoare logic to use it to prove that quicksort is not only quick but also sorts. . .

QuickCheck to the rescue!

```
-- usage: quickCheckN 10000 prop_qsort_isOrdered

isOrdered (x1:x2:xs) = x1 <= x2 && isOrdered (x2:xs)
isOrdered _ = True

prop_qsort_isOrdered :: [Int] -> Bool
prop_qsort_isOrdered = isOrdered . qsort1

quickCheckN n = quickCheckWith \$ stdArgs { maxSuccess
    = n }
```

- Imperative constructs win in terms of performance
- Haskell seems to be faster than Scala, C++ for *pure functional* constructs
- Garbage is a problem in functional languages (in 1960's it was called *cons*ing – but who uses Lisps anymore
- Scalaz is a library, Haskell is a compiler
- Pragmatic functional languages accept that mutation can be a fact of life
- One should isolate mutation, Monads not perfect – imperative code even less readable
- If it has functional interface, I can simply test it with randomized tests, don't have to worry what's inside

Bibliography

- Okasaki C., *Purely functional data structures*, Cambridge University Press, 1999
- Rabhi F., Lapalme G., *Algorithms, A Functional Programming Approach*, Addison-Wesley, 1999
- Bird R., *Introduction to Functional Programming using Haskell*, Prentice-Hall, 1998
- Chusano P., Bjarnason R., *Functional Programming in Scala*, Manning, 2014
- Odersky, M., Spoon, L., Venners, B. *Programming in Scala, 2nd ed.*, Artima, 2011