Performance evaluation of various functional programming styles

Jan Pustelnik

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

LambdaDays 2015, 2015-02-26
Outline

1. Introduction
   - General
   - Benchmarking
   - Code

2. Sorting
   - MergeSort
   - QuickSort
   - HeapSort
   - Correctness

3. Summary
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++
Windows 7 machine. The same for all the tests.
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.
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 code!

Sure, please go to https://github.com/gosubpl for the Source.
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
All programs used as examples are either Public Domain, MIT or Creative Commons by attribution.

Top-Down mergesort in Haskell

1. \texttt{mergesort} :: (a \to a \to \textbf{Bool}) \to [a] \to [a]
2. \texttt{mergesort \ pred \ []} = []
3. \texttt{mergesort \ pred \ [x]} = [x]
4. \texttt{mergesort \ pred \ xs} = \texttt{merge \ pred \ (mergesort \ pred \ xs1) \ (mergesort \ pred \ xs2)}
   \hspace{1em} where
5. \hspace{1em} (xs1, xs2) = \texttt{split \ xs}

Jan Pustelnik
Performance evaluation of various functional programming styles
Top-Down mergesort in Haskell, cntd.

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

5 merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
6 merge pred xs [] = xs
7 merge pred [] ys = ys
8 merge pred (x:xs) (y:ys)
9     | pred x y = x: merge pred xs (y:ys)
10    | 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
```

Jan Pustelnik
Performance evaluation of various functional programming styles
You can also do it in C++

It even has two built-in functions for merging: \texttt{std::merge} and \texttt{std::inplace_merge}

```cpp
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.

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

```scala
def msort[T](less: (T, T) => Boolean) (xs: List[T]): 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
        }
    }
}
```

Jan Pustelnik
Performance evaluation of various functional programming styles
And the performance crown goes to . . .

All times in the table below are in milliseconds

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

The table above gives us some point of reference – no thrills, really.
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!
Haskell Functional – http://en.literateprograms.org/ but almost identical programs can be found in [2] or [3]


C++ – could not find this anywhere, had to code it, but it was easy, doh!

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

```scala
1  def sortQuickTraditional(xs: Array[Int]): Array[Int] = {
2    def swap(i: Int, j: Int) {
3      val t = xs(i)
4      xs(i) = xs(j)
5      xs(j) = t
6    }
```

Jan Pustelnik

Performance evaluation of various functional programming styles
Standard mutable quicksort in Scala – contd.

```scala
1 def sort1(l: Int, r: Int) {
2   val pivot = xs((l + r) / 2)
3   var i = l
4   var j = r
5   while (i <= j) {
6     while (xs(i) < pivot) i += 1
7     while (xs(j) > pivot) j -= 1
8     if (i <= j) {
9       swap(i, j)
10      i += 1
11      j -= 1
12     }
13   }
14   if (l < j) sort1(l, j)
15   if (j < r) sort1(i, r)
16 }
17 sort1(0, xs.length - 1)
18 xs
```
One has to use the ST monad.

```haskell
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)
```

Jan Pustelnik
Performance evaluation of various functional programming styles
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.

```haskell
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 h i
  i <- sloop piv lo hi
  swap i h
  myqsort a lo (i-1)
  myqsort a (i+1) hi
  | otherwise = return ()
```

Jan Pustelnik

Performance evaluation of various functional programming styles
def sortQuickMonadicST(x: Array[Int]): Array[Int] = {
  def invert(x: (Int, Int)): (Int, Int) = (x._2, x._1)
  def identify(x: Int, y: Int): Int = y
  var arrLen = x.length
  var xiz = x.zipWithIndex
  var xizi = (xiz map invert).toList

type ForallST[A] = Forall[({type λ[S] = ST[S, A]})# λ]
  def noop[S] = ST[S, Unit]()
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 ()
  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)
} yield l
ST monad mutable quicksort Scala – contd.

```scala
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.

1 qsort1 [] = []
2 qsort1 (p:xs) = qsort1 lesser ++ [p] ++ qsort1 greater
3     where
4       lesser = [ y | y <- xs, y < p ]
5       greater = [ y | y <- xs, y >= p ]
Pure functional quicksort in Scala

As easy as in Haskell :) You can also replace Array with ArrayBuffer or Vector¹

```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 <)))
  }
}
```

¹Vector is immutable and should behave like an Array plus have nice amortised complexities.
Maybe even easier than in Haskell and maybe not. But maybe faster?

```cpp
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.

```cpp
vector<int> fql = funqsor(lesser);
vector<int> fqg = funqsor(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

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

```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
}
```

Jan Pustelnik
Performance evaluation of various functional programming styles
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

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

```haskell
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 }
```

Jan Pustelnik
Performance evaluation of various functional programming styles
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 consing – 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