*J. Functional Programming* **1** (1): 1–000, July 1998. Printed in the United Kingdom



# FUNCTIONAL PEARL

## *A Pointless Derivation of Radix Sort*

Jeremy Gibbons
*School of Computing and Mathematical Sciences, Oxford Brookes University*
*Gipsy Lane, Headington, Oxford OX3 0BP, UK.*
*Email:* `jgibbons@brookes.ac.uk`

**Abstract**

This paper is about point-free (or 'pointless') calculations — calculations performed at the level of function composition instead of that of function application. We address this topic with the help of an example, namely calculating the *radix-sort* algorithm from a more obvious specification of sorting. The message that we hope to send is that point-free calculations are sometimes surprisingly simpler than the corresponding point-wise calculations.

## 1 Introduction

This paper concerns algorithms for sorting a list of items into *lexical order*. Given is a list of (total) functions `ds`, each of which extracts a 'field' of information from an item. Two items `x` and `y` are *lexically ordered with respect to* `ds` if the lists of fields of `x` and `y` extracted by the functions `ds` are lexicographically ordered:

```
> ordered :: Ord b => [a->b] -> a -> a -> Bool
> ordered [] x y = True
> ordered (d:ds) x y = d x < d y ||
>                       (d x == d y) && ordered ds x y
```

Note that the fields themselves have to permit an ordering. For simplicity, we suppose that all fields are of the same type; later on, we will also assume that the field type is *bounded*, with minimum and maximum values `minBound` and `maxBound`, and enumeratable. For example, the items might be three-digit natural numbers, and the three 'fields' the hundreds, tens and units digits.

We claim that the following two-phase *tree-sort* algorithm 'obviously' sorts a list into lexical order. The first phase constructs a tree from the list; the second phase flattens the tree back to a list. In the first phase the list is partitioned into 'buckets' according to the most significant field. A tree is grown recursively in each of these buckets according to the remaining fields. Partitioning stops when there are no more fields. The second phase flattens the resulting tree of lists into one long list in the natural order.

What is less obvious is that the list can also be sorted by starting with the *least* significant field. The list is partitioned into buckets on each field, starting with the least significant; in between partitioning steps, the buckets are simply concatenated into one long list again. The crucial requirement is that the partitioning operation must be *stable*, in a sense made precise later. This second method is the well-known *distribution-sort* (Knuth, 1973) or *radix-sort* (Cormen *et al.*, 1990) algorithm.

In this paper, we derive radix-sort from tree-sort as an exercise in calculating programs from specifications. However, the main lesson we have learnt from this exercise is a methodological one. Our calculations got completely bogged down using the natural definitions of the various functions concerned. In attempting to rephrase the definitions in a form acceptable to Bird's program calculator (Bird, 1998) — in particular, eliminating as many variables as possible and performing point-free (or 'pointless') calculations at the level of function composition instead of point-wise calculations at the level of application — suddenly the calculations became almost trivial. This is the point of pointless calculations: when you travel light — discarding variables that do not contribute to the calculation — you can sometimes step lightly across the surface of the quagmire.

The remainder of the paper is structured as follows. In Section 2 we present a program for the 'obvious' tree-sort. Section 3 briefly presents some standard theory. The main calculation is in Section 4. In Section 5 we formalize and prove the stability property needed in the calculation. Section 6 concludes.

## 2 The program for tree-sort

The tree type we will use throughout the paper is

```
> data Tree a = Leaf a | Node [Tree a]
```

The first phase of tree-sort partitions the list into buckets, according to the most significant field—one bucket for each value in the range `minBound..maxBound`. A tree is grown recursively in each of these buckets according to the remaining fields. Thus, we have

```
> mktree :: (Bounded b, Enum b) => [a->b] -> [a] -> Tree [a]
> mktree [] xs     = Leaf xs
> mktree (d:ds) xs = Node (map (mktree ds) (ptn d xs))
```

Here, `ptn d xs` partitions the list `xs` into buckets according to the field extracted by `d`:

```
> ptn :: (Bounded b, Enum b) => (a->b) -> [a] -> [[a]]
> ptn d xs = [ filter ((m==).d) xs | m <- rng ]

> rng :: (Bounded a, Enum a) => [a]
> rng = [minBound..maxBound]
```

For brevity, we have introduced the variable `rng` for the range of field values. If the range contains $r$ values, and the list `ds` has $n$ elements, then `mktree ds xs` constructs an $r$-ary tree of depth $n$ (for any `xs`).

The second phase is to flatten the tree of lists into one long list, which will be lexically ordered according to the fields:

```
> flatten :: Tree [a] -> [a]
> flatten = foldt id concat
```

where `foldt` is the fold on our trees:

```
> foldt :: (a->b) -> ([b]->b) -> Tree a -> b
> foldt f g (Leaf x) = f x
> foldt f g (Node ts) = g (map (foldt f g) ts)
```

Combining the two phases, we have

```
> treesort :: (Bounded b, Enum b) => [a->b] -> [a] -> [a]
> treesort ds xs = flatten (mktree ds xs)
```

## 3 Theory

The theory of datatypes (Malcolm, 1990; Meijer *et al.*, 1991) provides many useful properties of folds (and their dual, unfolds (Gibbons & Jones, 1998)) over datatypes. In particular, the fold over a datatype enjoys a *universal property*, stating that the fold is the unique solution to a certain equation. Thus, the task of showing that a certain function `h` is equal to a given function expressed a fold (which would otherwise require an inductive proof) is reduced to showing that `h` is a solution to the equation characterizing the fold (which typically does not require induction).

For example, the universal property for the standard `foldr` on lists states, for strict `h`, that `h = foldr f e` precisely if

```
h [] = e
h (a:as) = f a (h as)
```

A consequence of this is the *fusion law* for `foldr`, giving conditions under which the composition of a function with a `foldr` is again a `foldr`: for strict `h`,

```
h . foldr f e = foldr f' e'
```

if and only if `e' = h e` and

```
h (f a (foldr f e x)) = f' a (h (foldr f e x))
```

The latter property follows in turn from the stronger condition

```
h (f a b) = f' a (h b)
```

The fold for trees enjoys the universal property, for strict `h`, that `h = foldt f g` if and only if

```
h . Leaf = f
h . Node = g . map h
```

The corresponding fusion law states that

```
h . foldt f g = foldt f' g'
```

provided that `h` is strict and

```
h . f = f'
h . g = g' . map h
```

## 4 Using folds

Unfortunately, none of the properties of fold discussed in Section 3 seem to apply to our program for tree-sort: the only fold is in the function `flatten`, and in general nothing can be said about a fold *after* another function. However, we can write `mktree` as a `foldr` over the list of field functions by eliminating the second parameter, which does not contribute to the equations. We have

```
mktree [] = Leaf
mktree (d:ds) = Node . map (mktree ds) . ptn d
```

and so

```
mktree = foldr fm Leaf
  where fm d g = Node . map g . ptn d
```

Now, both `flatten` and `mktree` are expressed using folds. Unfortunately, `treesort` is not expressed as the composition of another function with a `foldr`: we have

```
treesort ds = flatten . mktree ds
```

but it is `mktree`, not `mktree ds`, that is the fold. We can get around this problem by defining a synonym for composition:

```
> comp :: (b->c) -> (a->b) -> (a->c)
> comp f g = f . g
```

We now obtain

```
treesort ds = comp flatten (mktree ds)
```

or equivalently

```
treesort = comp flatten . mktree
```

(Of course, we could have written simply `(flatten .) . mktree`, but partially applied infix compositions are quite confusing to use.)

Now the fusion law is applicable, and we obtain

```
treesort = foldr ft et
```

if and only if

```
et = comp flatten Leaf
```

and

```
ft d (comp flatten (mktree ds)) = comp flatten (fm d (mktree ds))
```

For the first of these we get

```
   comp flatten Leaf
=    { definition of comp }
   flatten . Leaf
=    { definition of flatten }
   id
```

so we let `et` be `id`. For the second we get

```
   comp flatten (fm d (mktree ds))
=    { definitions of comp, fm }
   flatten . Node . map (mktree ds) . ptn d
=    { definition of flatten }
   concat . map flatten . map (mktree ds) . ptn d
=    { definition of treesort }
   concat . map (treesort ds) . ptn d
=    { claim (see Section 5):
         map (treesort ds) . ptn d = ptn d . treesort ds }
   concat . ptn d . treesort ds
=    { definition of treesort }
   concat . ptn d . comp flatten (mktree ds)
```

so we let `ft d g` be `concat . ptn d . g`. We have shown that

```
treesort = radixsort
```

where

```
> radixsort :: (Bounded b, Enum b) => [a->b] -> [a] -> [a]
> radixsort = foldr ft id
>   where ft d g = concat . ptn d . g
```

As the name suggests, this is the well-known radix-sort algorithm. The advantage of radix-sort over tree-sort is that it does not require a stack. Indeed, radix-sort was used to sort punched cards in the early days of computing: card 'sorting' machines could perform the `ptn d` stage on one column of a punched card, and all the operator had to do was `concat` the resulting piles of cards into one big pile and repeat the process for the remaining columns. Using tree-sort would have entailed keeping a 'stack' of many partially-sorted piles of cards.

## 5 Stability

We are left with the task of proving

```
map (treesort ds) . ptn d = ptn d . treesort ds
```

Informally, this condition states that `ptn d` is stable: partitioning a sorted sequence yields a collection of sorted buckets. Straightforward calculation* shows that this condition follows from

```
filter p . flatten = flatten . mapt (filter p)
```

and

```
mapt (filter p) . mktree ds = mktree ds . filter p
```

Here, `mapt` is map over trees, which is a fold:

```
> mapt :: (a->b) -> Tree a -> Tree b
> mapt f = foldt (Leaf . f) Node
```

The first of these new obligations is easy to discharge, given the following *fold-map fusion* law, a special case of fusion:

```
foldt f g . mapt h = foldt (f . h) g
```

The remaining proof obligation is to show that

```
mapt (filter p) . mktree ds = mktree ds . filter p
```

We would like to use fusion, the most powerful tool at our disposal, and `mktree` is the most obvious fold here on which to use it. However, neither side of the equation is in the correct form of some function composed with `mktree`. Fortunately, the trick of discarding idle variables works as well here as it did in Section 4.

The left-hand side is equal to

```
(comp (mapt (filter p)) . mktree) ds
```

and so the idle `ds` can easily be discarded. Fusion states that

```
comp (mapt (filter p)) . mktree = foldr f e
```

provided that

```
comp (mapt (filter p)) Leaf = e
```

and

```
comp (mapt (filter p)) (fm d g) = f d (comp (mapt (filter p)) g)
```

Straightforward calculations conclude that `e` should be `Leaf . filter p` and `f` should be just `fm`, and we obtain

```
comp (mapt (filter p)) . mktree = foldr fm (Leaf . filter p)
```

On the right-hand side of the remaining proof obligation we have the expression `mktree ds . filter p` and we need to discard the idle `ds`. To this end we introduce the function `after` (so we can write composition partially applied to the other argument):

* The calculations omitted from the body of the paper are included in Appendix A.

```
> after :: (a->b) -> (b->c) -> (a->c)
> after f g = g . f
```

We can now write the right-hand side of the remaining proof obligation as

```
(after (filter p) . mktree) ds
```

Another straightforward application of fusion shows that

```
after (filter p) . mktree = foldr fm (Leaf . filter p)
```

provided that

```
ptn d . filter p = map (filter p) . ptn d
```

The proof of this final property relies essentially on the fact that two filters (of total predicates) commute with each other:

```
    map (filter p) (ptn d xs)
=     { definition of ptn }
    map (filter p) [ filter ((m==).d) xs | m <- rng ]
=     { definition of map }
    [ filter p (filter ((m==).d) xs) | m <- rng ]
=     { filters (of total predicates) commute with each other }
    [ filter ((m==).d) (filter p xs) | m <- rng ]
=     { definition of ptn }
    ptn d (filter p xs)
```

Note that the property only holds for total predicates `p`. Fortunately, in our case the predicates are all of the form `(m==).d`, and so are total when `d` is.

## 6 Conclusions

We expected the calculation of radix-sort from tree-sort to be a simple exercise. However, our first attempts became bogged down in a morass of bound variables and cascading proof obligations. Judicious use of partially applied function compositions (of the form `comp f` and `after f`) eliminated the awkward variables, and at the same time simplified the proofs to linear calculations, making the problem tractable.

The program as it stands is inefficient, because `ptn d xs` performs many traversals of the list `xs`, one for each bucket. A more efficient approach is to perform a single traversal, constructing all the buckets at once:

```
ptn d xs = foldr (fp d) empties xs
  where empties = [ [] | m <- rng ]
        fp d x buckets    = at (fromEnum (d x)) (x:) buckets
        at 0 f (x:xs)     = f x : xs
        at (n+1) f (x:xs) = x : at n f xs
```

However, this more efficient version is more difficult to manipulate, so throughout this paper we have stuck with the unoptimized version.

**Acknowledgements**

Thanks are due to Richard Bird, who threw all my variables away, and to Patrick Djojo Surjo, whose questions inspired this investigation in the first place.

## References

Bird, Richard S. (1998). *Introduction to functional programming using Haskell.* Prentice-Hall.

Cormen, Thomas H., Leiserson, Charles E., & Rivest, Ronald L. (1990). *Introduction to algorithms.* MIT Press.

Gibbons, Jeremy, & Jones, Geraint. (1998). The under-appreciated unfold. *ACM International Conference on Functional Programming.*

Knuth, Donald E. (1973). *The art of computer programming, volume 3: Sorting and searching.* Addison-Wesley.

Malcolm, Grant. (1990). Data structures and program transformation. *Science of computer programming*, **14**, 255–279.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of:* Hughes, John (ed), *LNCS 523: Functional Programming Languages and Computer Architecture.* Springer-Verlag.

## Appendix A: Calculations

This appendix contains the straightforward calculations omitted from the body of the paper.

The stability condition claimed in Section 4 and discussed in Section 5 was that

```
map (treesort ds) . ptn d = ptn d . treesort ds
```

This property is proved as follows.

```
   ptn d (treesort ds xs) = map (treesort ds) (ptn d xs)
≡    { definition of ptn }
   [ filter ((m==).d) (treesort ds xs) | m <- rng ] =
     map (treesort ds) [ filter ((m==).d) xs | m <- rng ]
≡    { definition of map }
   [ filter ((m==).d) (treesort ds xs) | m <- rng ] =
     [ treesort ds (filter ((m==).d) xs) | m <- rng ]
⇐    { Leibniz }
   filter p . treesort ds = treesort ds . filter p
≡    { definition of treesort }
   filter p . flatten . mktree ds = flatten . mktree ds . filter p
⇐    { introduce mapt }
   filter p . flatten = flatten . mapt (filter p)   and
   mapt (filter p) . mktree ds = mktree ds . filter p
```

thereby generating two new proof obligations.

For the first of the two new proof obligations we have

```
   filter p . flatten
=    { definition of flatten }
   filter p . foldt id concat
=    { fusion: filter p . concat = concat . map (filter p) }
   foldt (filter p) concat
=    { fold-map fusion }
   foldt id concat . mapt (filter p)
=    { definition of flatten }
   flatten . mapt (filter p)
```

In Section 5 we showed that

```
comp (mapt (filter p)) . mktree = foldr f e
```

provided that

```
comp (mapt (filter p)) Leaf = e
```

and

```
comp (mapt (filter p)) (fm d g) = f d (comp (mapt (filter p)) g)
```

For the first of these conditions we have

```
   comp (mapt (filter p)) Leaf
=    { definition of comp }
   mapt (filter p) . Leaf
=    { definition of mapt }
   Leaf . filter p
```

and so we should let `e` be `Leaf . filter p`; for the second we have

```
   comp (mapt (filter p)) (fm d g)
=    { definition of comp, fm }
   mapt (filter p) . Node . map g . ptn d
=    { definition of mapt }
   Node . map (mapt (filter p)) . map g . ptn d
=    { definition of fm }
   fm d (mapt (filter p) . g)
=    { definition of comp }
   fm d (comp (mapt (filter p)) g)
```

and so we should let `f` be `fm`.

The second of the two new proof obligations is now discharged as follows:

```
   mapt (filter p) . mktree ds = mktree ds . filter p
≡    { discard idle variable ds }
   comp (mapt (filter p)) . mktree = after (filter p) . mktree
≡    { fusion (above) }
   foldr fm (Leaf . filter p) = after (filter p) . mktree
⇐    { fusion again }
```

```
    after (filter p) Leaf = Leaf . filter p   and
    after (filter p) (fm d g) = fm d (after (filter p) g)
  ≡   { first requirement is trivial; definitions of after, fm }
    Node . map g . ptn d . filter p = Node . map (g . filter p) . ptn d
  ⇐   { Leibniz }
    ptn d . filter p = map (filter p) . ptn d
```

## Appendix B: Digits

This appendix includes Haskell code for a type `Digit`, and the necessary accoutrements to make it useful for radix-sorting.

```
> base = 10 :: Int

> newtype Digit = D Int   deriving (Eq,Ord,Show)

> instance Bounded Digit where
>   minBound = D 0
>   maxBound = D (base-1)

> instance Enum Digit where
>   toEnum a = D a
>   fromEnum (D a) = a
>   enumFrom d
>     | d == maxBound = [d]
>     | otherwise     = d : enumFrom (succ d)
>   enumFromThen d1 d2
>     | (next <= fromEnum (maxBound :: Digit)) &&
>       (next >= fromEnum (minBound :: Digit))
>                     = d1 : enumFromThen d2 (toEnum next)
>     | otherwise     = [d1,d2]
>       where next = fromEnum d2 + fromEnum d2 - fromEnum d1

> threedigits :: [ Int -> Digit ]
> threedigits = [ digit2, digit1, digit0 ] where
>   digit0 n = D (n `mod` base)
>   digit1 n = digit0 (n `div` base)
>   digit2 n = digit1 (n `div` base)

> exampleInput = [ 123, 432, 753, 53 ] :: [Int]
> specOutput = treesort threedigits exampleInput
> implOutput = radixsort threedigits exampleInput
```