This note contains appendices to the paper *A Pointless Derivation of Radixsort* by Jeremy Gibbons, in *Journal of Functional Programming* 9(3), pp. 339-346, May 1999.

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
```