```
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

1) `treesort = comp flatten . mktree` = (comp flatten) ∘ (foldr fm L)

(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

foldr ft et ⇒

1)

treesort ds = flatten ∘ (mkTree ds)

<u>Via type analysis :</u>

treesort ds : [a] -> [a]

flatten : Tree [a] -> ?

mkTree ds : ? -> ?

(∘) flatten : ([a] -> Tree [a]) ->
[a] -> [a]

comp flatten = (∘) flatten

treesort : [a->b] -> [a] -> [a]

mkTree : [a->b] -> [a] -> Tree [a]

what is the type of (comp flatten) ∘ ?

(comp flatten) ∘ : ?

remember the type of (∘) :

->

$(\circ) : (B \to C) \to (A \to B) \to (A \to C)$

therefore

$f \circ : (A \to B) \to (A \to C)$

Notice that A is arbitrary but

$B = \text{dom } f$

$C = \text{codom } f$

In our case $f = \text{comp flatten} \Rightarrow$

$(\text{comp flatten}) \circ : A \to ([a] \to \text{Tree } [a]) \longrightarrow A \to ([a] \to [a])$

The task is now to find g such that

$(\text{comp flatten}) \circ g : [a \to b] \to [a] \to [a]$

this means

$g : [a \to b] \to [a] \to \text{Tree } [a]$!

$\longrightarrow$

We have only one function:

$$g = mkTree$$

Thus

$$treesort = (comp\ flatten) \circ mkTree$$

<u>Via equational reasoning:</u>

....

2)

```
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
```

foldt id concat

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)
```

3)

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

2) We have :

$$\text{fold } f' e' = h \circ \text{fold } f e$$

$$\equiv$$

$$e' = h\, e \quad \wedge$$

$$f' a\, (h\, (\text{fold } f e x)) = h\, (f\, a\, (\text{fold } f e x))$$

$\forall f', e', h, f, e$. Apply To

$$\text{foldr } ft\ et = (\text{comp flatten}) \circ \text{foldr } fm \text{ Leaf}$$

with $\underline{\text{foldr } fm \text{ Leaf} = \text{mkTree}}$:

i) $et = \text{comp flatten leaf}$

ii) $ft\ d\ (\text{comp flatten } (\text{mktree } ds))$

$$=$$

$\text{comp flatten } (fm\ d\ (\text{mkTree } ds))$

3)

comp flatten (fm d (mktree ds))

=

concat ∘ ptn d ∘ comp flatten (mktree ds)

---

comp flatten (fm d (mktree ds))

= { fm d g = Node ∘ map g ∘ ptn d }

comp flatten (Node ∘ map (mkTree ds) ∘ ptn d)

= { def. comp }

flatten ∘ (Node ∘ map (mkTree ds) ∘ ptn d)

= { flatten = foldt id concat }

(foldt id concat) ∘ . . . . .

= { foldt f g (Node ts) =

g (map (foldt f g) ts) }

→

$$\text{concat} \circ (\text{map}\ (\text{fold id concat}) \circ$$
$$\text{map}\ (\text{mkTree}\ ds) \circ \text{ptn}\ d)$$

$= \{$ def. flatten, Tree functor $\}$

$$\text{concat} \circ (\text{map}\ (\text{flatten} \circ \text{mkTree}\ ds) \circ$$
$$\text{ptn}\ d)$$

$= \{\ \text{treesort}\ ds = \text{flatten} \circ (\text{mkTree}\ ds)\ \}$

$$\text{concat} \circ (\text{map}\ (\text{treesort}\ ds) \circ \text{ptn}\ d)$$

$=$

....