J. Functional Programming **10** (3): 305–317, May 2000. Printed in the United Kingdom
© 2000 Cambridge University Press

FUNCTIONAL PEARL

Perfect trees and bit-reversal permutations

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

Abstract

One well known algorithm is the Fast Fourier Transform (FFT). An efficient iterative version of the FFT algorithm performs as a first step a bit-reversal permutation of the input list. The bit-reversal permutation swaps elements whose indices have binary representations that are the reverse of each other. Using an amortized approach, this operation can be made to run in linear time on a random-access machine. An intriguing question is whether a linear-time implementation is also feasible on a pointer machine, that is, in a purely functional setting. We show that the answer to this question is in the affirmative. In deriving a solution, we employ several advanced programming language concepts such as nested datatypes, associated fold and unfold operators, rank-2 types and polymorphic recursion.

1 Introduction

A *bit-reversal permutation* operates on lists whose length is $n = 2^k$ for some natural number k, and swaps elements whose indices have binary representations that are the reverse of each other. The bit-reversal permutation of a list of length $8 = 2^3$, for instance, is given by

$$brp_3\ [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \quad = \quad [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7].$$

In this case, the elements at positions $1 = (001)_2$ and $4 = (100)_2$ and the elements at positions $3 = (011)_2$ and $6 = (110)_2$ are swapped. Formally, we may define brp_k as the unique function that satisfies

$$at\ i \cdot brp_k \quad = \quad at\ (rev_k\ i), \tag{1}$$

for all $i \in \{0, \ldots, n-1\}$. The function *at* denotes list indexing and rev_k computes the bit-reversal of a natural number. Assuming that list indexing takes a constant time, and given a function rev_k that runs in $\Theta(k)$ time, it is straightforward to implement brp_k such that it takes $\Theta(nk)$ time to permute a list of length $n = 2^k$. Some extra cleverness is necessary to make brp_k run in linear time – see Cormen *et al.* (1991, Problem 18.1). Now, the question is whether brp_k can be implemented to run in linear time *without* assuming a constant time indexing function. Again, it is straightforward to design an implementation that takes $\Theta(nk)$ time. The main idea is to represent the

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

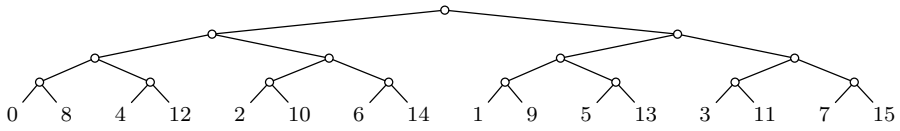

Fig. 1. The bit reversal permutation of the list $[0..15]$.

input list by a perfectly balanced, binary leaf tree (Dielissen and Kaldewaij, 1995), and to use tree instead of list indexing. In the rest of this Functional Pearl, we show how to develop this idea into a linear-time implementation.

To begin, let us assume that the length of the input list is fixed and known in advance. The algorithmic part of the solution will be developed under this assumption. Once the algorithmic details have been settled, we discuss the extensions necessary to make the program work for inputs of unknown lengths.

2 Perfect trees

This section introduces perfectly balanced, binary leaf trees – perfect trees for short – and recursion operators for folding and unfolding them. To represent perfect trees we employ the simplest scheme conceivable, namely, pairs of pairs of ... of elements. Formally, a *perfect tree of rank n* is an element of $\Delta^n a$, where Δ is given by

$$\mathbf{type}\ \Delta\ a \quad = \quad a \times a,$$

and F^n is defined by $F^0\ a = a$ and $F^{n+1}\ a = F^n\ (F\ a)$. Members of Δa are also called *nodes*. The tree depicted in figure 1, for instance, is represented by the term

$$((((0,8),(4,12)),((2,10),(6,14))),(((1,9),(5,13)),((3,11),(7,15))))$$

of type $\Delta^4\ Int$. To manipulate trees we will make frequent use of the *mapping function* on nodes defined by

$$\begin{array}{lcl} \Delta & :: & (a \to b) \to (\Delta\ a \to \Delta\ b) \\ \Delta\ \varphi\ (a_0, a_1) & = & (\varphi\ a_0, \varphi\ a_1). \end{array}$$

Following common practice, we use the same name both for the type constructor and for the corresponding map on functions. Accordingly, the mapping function for perfect trees of rank n is given by Δ^n, where $f^0\ a = a$ and $f^{n+1}\ a = f^n\ (f\ a)$. The combination of type constructor and mapping function is often referred to as a *functor*. Every mapping function satisfies the following so-called *functor laws*, which will prove useful in the calculations to follow.

$$\begin{array}{rcl} \Delta\ id & = & id \\ \Delta\ (\varphi \cdot \psi) & = & \Delta\ \varphi \cdot \Delta\ \psi. \end{array}$$

Now, to build and to flatten perfect trees we employ variants of recursion schemes widely known as cata- and anamorphisms (Meijer *et al.*, 1991). The catamorphism on Δ^n, denoted $([\text{-}])_n$, takes a function of type $\Delta a \to a$ and replaces each node in its

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

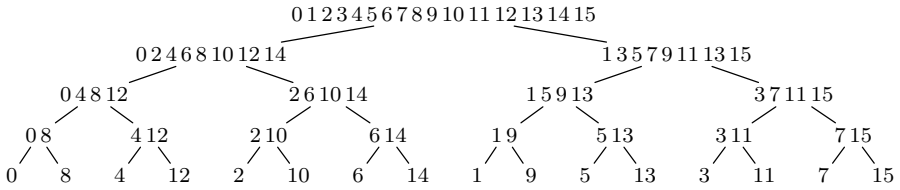

Fig. 2. Constructing the bit reversal permutation of [0..15].

input with this function:

$$\begin{array}{lcl} (\!|\text{-}|\!)_n & :: & (\Delta\ a \to a) \to (\Delta^n\ a \to a) \\ (\!|\varphi|\!)_0 & = & id \\ (\!|\varphi|\!)_{n+1} & = & \varphi \cdot \Delta\ (\!|\varphi|\!)_n. \end{array}$$

Since the recursion operator is indexed by the rank of its input, we should rather speak of a *ranked catamorphism*. The converse of a ranked catamorphism is a *ranked anamorphism*, denoted $[\!|\text{-}|\!]_n$, which takes a function of type $a \to \Delta a$ and builds a perfect tree from a given seed of type a:

$$\begin{array}{lcl} [\!|\text{-}|\!]_n & :: & (a \to \Delta\ a) \to (a \to \Delta^n\ a) \\ [\!|\psi|\!]_0 & = & id \\ [\!|\psi|\!]_{n+1} & = & \Delta\ [\!|\psi|\!]_n \cdot \psi. \end{array}$$

Ranked cata- and anamorphisms satisfy a variety of properties. We will make use of the following laws:

$$(\!|\varphi|\!)_n = \Delta^0\ \varphi \cdot \ldots \cdot \Delta^{n-1}\ \varphi \tag{2}$$

$$[\!|\psi|\!]_n = \Delta^{n-1}\ \psi \cdot \ldots \cdot \Delta^0\ \psi \tag{3}$$

$$(\!|\varphi|\!)_n \cdot [\!|\psi|\!]_n = id \iff \varphi \cdot \psi = id \tag{4}$$

$$[\!|\psi|\!]_n \cdot (\!|\varphi|\!)_n = id \iff \psi \cdot \varphi = id. \tag{5}$$

The first two laws show that ranked cata- and anamorphisms can be expressed as compositions of mapping functions. The third and fourth laws state that ranked cata- and anamorphisms are inverse to each other if the base functions are.

3 Two recursive solutions

Recall the main idea of implementing brp_k sketched in the introduction: the input list is transformed into a perfect tree, which is then repeatedly indexed to build the bit-reversal permutation. An alternative approach that avoids the use of an indexing operation works by building a perfect tree, and then flattening it into a list. During either the first or second phases, the elements are shuffled to obtain the desired bit-reversal permutation. Figure 2 illustrates the building of a perfect tree that has the bit-reversal permutation of the input list as a frontier.

Building a perfect tree is probably a matter of routine: the input list is split into two equal halves; trees are built recursively for each half, and the results are finally combined. Here, and in what follows, we assume that the input list has a

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

length $n = 2^k$. Now, there are essentially two methods for splitting a list of length 2^k into two equal halves. The first, called *uncat*, partitions a list according to the most significant bit of the indices; the second, called *uninterleave*, partitions a list according to the least significant bit:

$$\begin{array}{lcl} \mathit{uncat}\ [a_0,\ldots,a_{m-1}] & = & ([a_0,\ldots,a_{m/2-1}],[a_{m/2},\ldots,a_{m-1}]) \\ \mathit{uninterleave}\ [a_0,a_1,a_2,a_3,\ldots] & = & ([a_0,a_2,\ldots],[a_1,a_3,\ldots]). \end{array}$$

Both functions have natural inverses, termed *cat* and *interleave*, i.e. $\mathit{cat} \cdot \mathit{uncat} = \mathit{id}$ and $\mathit{interleave} \cdot \mathit{uninterleave} = \mathit{id}$. Since we consider only lists of length 2^k, the dual properties $\mathit{uncat} \cdot \mathit{cat} = \mathit{id}$ and $\mathit{uninterleave} \cdot \mathit{interleave} = \mathit{id}$ hold, as well. Building upon *uncat* and *uninterleave* we obtain two functions for constructing a perfect tree of rank k. The first, $\mathit{unflatten}_k$, builds a tree that has the input list as a frontier, while the second, $\mathit{unshuffle}_k$, builds a tree that has the bit-reversal permutation as a frontier:

$$\begin{array}{lcl} \mathit{unflatten}_k, \mathit{unshuffle}_k & :: & [a] \to \Delta^k\ a \\ \mathit{unflatten}_k & = & \Delta^k\ \mathit{unwrap} \cdot (\!|\mathit{uncat}|\!)_k \\ \mathit{unshuffle}_k & = & \Delta^k\ \mathit{unwrap} \cdot (\!|\mathit{uninterleave}|\!)_k \end{array}$$

The function *unwrap* is given by $\mathit{unwrap}\ [a] = a$; we will also require its converse, *wrap*, which is accordingly defined by $\mathit{wrap}\ a = [a]$. As an aside, note that the trees generated by $\mathit{unflatten}_k$ and $\mathit{unshuffle}_k$ may be considered as *radix trees*: $\mathit{unflatten}_k\ x$ represents the finite map $i \mapsto \mathit{at}\ i\ x$, while $\mathit{unshuffle}_k\ x$ represents $i \mapsto \mathit{at}\ (\mathit{rev}_k\ i)\ x$.

From $\mathit{unflatten}_k$ and $\mathit{unshuffle}_k$, we can easily derive two functions for flattening a tree. The derivation of $\mathit{unflatten}_k$'s inverse proceeds as follows:

$$\begin{array}{ll} & \mathit{flatten}_k \cdot \mathit{unflatten}_k = \mathit{id} \\ \Longleftarrow & \quad \{\ \text{definition } \mathit{unflatten}_k\ \} \\ & \mathit{flatten}_k \cdot \Delta^k\ \mathit{unwrap} \cdot (\!|\mathit{uncat}|\!)_k = \mathit{id} \\ \Longleftarrow & \quad \{\ \mathit{cat} \cdot \mathit{uncat} = \mathit{id} \text{ and (4)}\ \} \\ & \mathit{flatten}_k \cdot \Delta^k\ \mathit{unwrap} = (\!|\mathit{cat}|\!)_k \\ \Longleftarrow & \quad \{\ \Delta \text{ functor and } \mathit{wrap} \cdot \mathit{unwrap} = \mathit{id}\ \} \\ & \mathit{flatten}_k = (\!|\mathit{cat}|\!)_k \cdot \Delta^k\ \mathit{wrap}. \end{array}$$

The derivation of $\mathit{unshuffle}_k$'s inverse proceeds in an analogous fashion. To summarize,

$$\begin{array}{lcl} \mathit{flatten}_k, \mathit{shuffle}_k & :: & \Delta^k\ a \to [a] \\ \mathit{flatten}_k & = & (\!|\mathit{cat}|\!)_k \cdot \Delta^k\ \mathit{wrap} \\ \mathit{shuffle}_k & = & (\!|\mathit{interleave}|\!)_k \cdot \Delta^k\ \mathit{wrap}. \end{array}$$

Now, by composing $\mathit{unshuffle}_k$ with $\mathit{flatten}_k$ or $\mathit{unflatten}_k$ with $\mathit{shuffle}_k$, we obtain two $\Theta(nk)$ time implementations of brp_k:

$$\mathit{brp}_k = (\!|\mathit{cat}|\!)_k \cdot (\!|\mathit{uninterleave}|\!)_k = (\!|\mathit{interleave}|\!)_k \cdot (\!|\mathit{uncat}|\!)_k.$$

The proof that brp_k satisfies the specification (1) is left as an exercise to the reader. Note that both cata- and both anamorphisms take $\Theta(nk)$ time. It is well known that the running time of $\mathit{unflatten}_k$ can be improved to $\Theta(n)$ using a technique

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

called *tupling* (Bird, 1998). The dual technique termed *accumulation* may be used to improve the complexity of $flatten_k$. However, the overall gain is only a constant factor, since $unshuffle_k$ and $shuffle_k$ are not amenable to these techniques. The key to a linear-time implementation of brp_k is to build and to flatten perfect trees *iteratively*.

4 Two iterative solutions

Rather than introducing the iterative versions in a single big 'eureka' step, we try to derive them from the recursive functions defined in the previous section. In fact, we present two different derivations: the first is based on algorithmic considerations, while the second, which is more elegant but also more abstract, rests upon the so-called *naturality* of brp_k.

4.1 A derivation based on algorithmic considerations

Since flattening a tree is simpler than building one, we start by improving $flatten_k$ and its colleague $shuffle_k$. To this end we try to express $flatten_{i+1}$ in terms of $flatten_i$:

$$flatten_{i+1} \quad = \quad step \cdot flatten_i. \tag{6}$$

It is not entirely obvious that this approach works. However, if it works, then the iterative variant of $flatten_k$ is given by $step^k \cdot wrap$ (note that $flatten_0 = wrap$). Now, the function *step* has type $[\Delta a] \to [a]$, i.e. it transforms a list of pairs of elements into a list of elements. A moment's reflection reveals that *step* takes the list $[(a_0, b_0), (a_1, b_1), \ldots]$ to $[a_0, b_0, a_1, b_1, \ldots]$. Thus, it can be defined by $interleave \cdot unzip$, where *unzip* is given by

$$\begin{array}{lcl} unzip & :: & [\Delta\, a] \to \Delta\, [a] \\ unzip & = & list\ fst\ \triangle\ list\ snd. \end{array}$$

Here *list* denotes the mapping function on lists and $(\triangle)$ is given by $(\varphi_0 \triangle \varphi_1)\ a = (\varphi_0\ a, \varphi_1\ a)$. In the sequel we also require *unzip*'s inverse, denoted *zip*. The reason for defining *step* in terms of *unzip* is simply to make the symmetry between $flatten_k$ and $shuffle_k$ explicit (see below). The crucial property of $step = interleave \cdot unzip$ is that it distributes over *cat*, i.e.

$$step \cdot cat \quad = \quad cat \cdot \Delta\, step \tag{7}$$

$$step \cdot (\!|cat|\!)_i \quad = \quad (\!|cat|\!)_i \cdot \Delta^i\ step. \tag{8}$$

Now, to prove (6) we reason

$$\begin{array}{lcl} flatten_{i+1} & = & \{\ \text{definition } flatten_k\ \} \\ & & (\!|cat|\!)_{i+1} \cdot \Delta^{i+1}\ wrap \\ & = & \{\ (2)\ \} \\ & & (\!|cat|\!)_i \cdot \Delta^i\ cat \cdot \Delta^{i+1}\ wrap \\ & = & \{\ \Delta \text{ functor}\ \} \end{array}$$

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & (\!|cat|\!)_i \cdot \Delta^i \ (cat \cdot \Delta\ wrap) \\
= & \quad \{\ cat \cdot \Delta\ wrap = step \cdot wrap\ \} \\
 & (\!|cat|\!)_i \cdot \Delta^i \ (step \cdot wrap) \\
= & \quad \{\ \Delta \text{ functor}\ \} \\
 & (\!|cat|\!)_i \cdot \Delta^i\ step \cdot \Delta^i\ wrap \\
= & \quad \{\ (8)\ \} \\
 & step \cdot (\!|cat|\!)_i \cdot \Delta^i\ wrap \\
= & \quad \{\ \text{definition } flatten_k\ \} \\
 & step \cdot flatten_i .
\end{array}
$$

The derivation for $shuffle_k$ proceeds in an analogous fashion. It suffices, in fact, to interchange the rôles of *cat* and *interleave*. To summarize,

$$
\begin{array}{rcl}
flatten_k & = & (interleave \cdot unzip)^k \cdot wrap \\
shuffle_k & = & (cat \cdot unzip)^k \cdot wrap .
\end{array}
$$

Given these equations, it is almost trivial to derive iterative definitions for $unflatten_k$ and $unshuffle_k$. We get

$$
\begin{array}{rcl}
unflatten_k & = & unwrap \cdot (zip \cdot uninterleave)^k \\
unshuffle_k & = & unwrap \cdot (zip \cdot uncat)^k .
\end{array}
$$

Both $zip \cdot uninterleave$ and $zip \cdot uncat$ take time proportional to the size of the input list. Since the length of the list is halved in each step, we have a total running time of $2^k + 2^{k-1} + \cdots + 2 + 1 = \Theta(n)$. Putting things together, we obtain two linear-time implementations of brp_k:

$$
brp_k = (interleave \cdot unzip)^k \cdot (zip \cdot uncat)^k = (cat \cdot unzip)^k \cdot (zip \cdot uninterleave)^k .
$$

4.2 A derivation based on naturality

The bit-reversal permutation satisfies a very fundamental property:

$$
list\ h \cdot brp_k \quad = \quad brp_k \cdot list\ h . \tag{9}
$$

This so-called *naturality law* holds for every polymorphic function of type $[a] \to [a]$ – see Wadler (1989). Basically, (9) captures the intuitive property that a polymorphic list-processing function does not depend in any way upon the nature of the list elements. All such a function can possibly do is to rearrange the input list. Thus, applying h to each element of the input list and then rearranging yields the same result as rearranging and then applying h to each element.

Building upon the naturality law, we can give an alternative, more elegant derivation of the linear-time brp_k implementations. To this end let us unfold the *first* recursive solution:

$$
\begin{array}{rcl}
brp_{k+1} & = & \{\ \text{first definition of } brp_k \text{ in section 3}\ \} \\
 & & (\!|cat|\!)_{k+1} \cdot [\!|uninterleave|\!]_{k+1}
\end{array}
$$

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{ \text{ definition } (\![\text{-}]\!)_k \text{ and } [\![\text{-}]\!]_k \ \} \\
 & \quad cat \cdot \Delta\, (\![cat]\!)_k \cdot \Delta\, [\![uninterleave]\!]_k \cdot uninterleave \\
= & \{ \ \Delta \text{ functor and definition } brp_k \ \} \\
 & \quad cat \cdot \Delta\, brp_k \cdot uninterleave.
\end{array}
$$

Note that the *second* iterative solution also depends upon *cat* and *uninterleave*. Unfolding its definition, we obtain

$$
\begin{array}{lll}
brp_{k+1} & = & \{ \text{ second definition of } brp_k \text{ in section 4.1 } \} \\
 & & \quad (cat \cdot unzip)^{k+1} \cdot (zip \cdot uninterleave)^{k+1} \\
 & = & \{ \text{ definition } \text{-}^k \text{ and definition } brp_k \ \} \\
 & & \quad cat \cdot unzip \cdot brp_k \cdot zip \cdot uninterleave.
\end{array}
$$

Now, in order to join the loose ends we require

$$
unzip \cdot brp_k \quad = \quad \Delta\, brp_k \cdot unzip.
$$

So, unzipping a list of pairs and then independently rearranging the two output lists should yield the same result as rearranging a list of pairs and then unzipping. In fact, this proves to be true for every polymorphic function of type $[a] \rightarrow [a]$. Here is a simple calculational proof.

$$
\begin{array}{lll}
unzip \cdot \varphi & = & \{ \text{ definition } unzip \ \} \\
 & & \quad (list\ fst \ \triangle\ list\ snd) \cdot \varphi \\
 & = & \{ \ (\varphi_0 \ \triangle\ \varphi_1) \cdot \varphi = (\varphi_0 \cdot \varphi) \ \triangle\ (\varphi_1 \cdot \varphi) \ \} \\
 & & \quad (list\ fst \cdot \varphi) \ \triangle\ (list\ snd \cdot \varphi) \\
 & = & \{ \ \varphi \text{ satisfies } list\ h \cdot \varphi = \varphi \cdot list\ h \ \} \\
 & & \quad (\varphi \cdot list\ fst) \ \triangle\ (\varphi \cdot list\ snd) \\
 & = & \{ \ (\varphi \cdot \varphi_0) \ \triangle\ (\varphi \cdot \varphi_1) = \Delta\, \varphi \cdot (\varphi_0 \ \triangle\ \varphi_1) \ \} \\
 & & \quad \Delta\, \varphi \cdot (list\ fst \ \triangle\ list\ snd) \\
 & = & \{ \text{ definition } unzip \ \} \\
 & & \quad \Delta\, \varphi \cdot unzip
\end{array}
$$

Using an analogous argument we can also give an alternative derivation of the *first* iterative solution.

5 A Haskell program

Up to now we have assumed that the length of the input list is fixed and known in advance. Let us finally get rid of this assumption. For concreteness, the final program will be given in the functional programming language Haskell 98 (Peyton Jones and Hughes,1999). The main reason for choosing Haskell is that we require a fairly advanced type system.

We must first seek a suitable datatype for representing perfect trees. Since the type should encompass perfect trees of arbitrary rank, we are, in fact, looking for

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

a representation of $\Delta^0 + \Delta^1 + \Delta^2 + \cdots$. Here, '+' denotes the disjoint sum raised to the level of functors, $(F_0 + F_1)\ a = F_0\ a + F_1\ a$. Recall that F^n is given by $F^0\ a = a$ and $F^{n+1}\ a = F^n\ (F\ a)$. Alternatively, we may define $F^0 = Id$ and $F^{n+1} = F^n \cdot F$ where Id is the identity functor, $Id\ a = a$, and '$\cdot$' denotes functor composition, $(F \cdot G)\ a = F\ (G\ a)$. Now, using the fact that functor composition distributes leftward through sums, $(G_0 + G_1) \cdot F = G_0 \cdot F + G_1 \cdot F$, we obtain

$$\Delta^0 + \Delta^1 + \Delta^2 + \cdots \quad = \quad Id + (\Delta^0 + \Delta^1 + \Delta^2 + \cdots) \cdot \Delta.$$

Replacing $\Delta^0 + \Delta^1 + \Delta^2 + \cdots$ by an unknown we arrive at the following fixpoint equation for perfect trees:

$$Perfect \quad = \quad Id + Perfect \cdot \Delta.$$

Rewriting the functor equation in an applicative style and introducing constructor names yields the desired Haskell datatype definition:

$$\textbf{data}\ Perfect\ a \quad = \quad Zero\ a \mid Succ\ (Perfect\ (\Delta\ a)).$$

This definition is somewhat unusual in that the recursive component, $Perfect\ (\Delta\ a)$, is not identical to the left-hand side of the equation. The type recursion is nested which is why datatype definitions with this property are called *nested datatypes* (Birdland Meertens, 1998). Abbreviating the constructor names to their first letter the tree of figure 1 is represented by the following term:

$$\begin{aligned} S\ (S\ (S\ (S\ (Z\ ((((0,8),(4,12)),((2,10),(6,14))),\\ (((1,9),(5,13)),((3,11),(7,15)))))))). \end{aligned}$$

Note that the 'prefix' $S^n\ Z$ encodes the tree's rank in unary representation.

It is interesting to contrast *Perfect* to the 'usual' definition of binary leaf trees, which in fact corresponds to the following fixpoint equation:

$$Tree \quad = \quad Id + \Delta \cdot Tree.$$

Clearly, *Tree* is not identical to *Perfect*, the formal reason being that functor composition does not distribute rightward through sums. In general, we only have $F \cdot (G_0 + G_1) \supseteq F \cdot G_0 + F \cdot G_1$. Here is the Haskell datatype corresponding to the functor equation above:

$$\textbf{data}\ Tree\ a \quad = \quad Leaf\ a \mid Fork\ (\Delta\ (Tree\ a)).$$

Encoded as an element of *Tree Int*, the tree of figure 1 reads

$$\begin{aligned} &F\ (F\ (F\ (F\ (L\ 0, L\ 8), F\ (L\ 4, L\ 12)), F\ (F\ (L\ 2, L\ 10), F\ (L\ 6, L\ 14))),\\ &\quad F\ (F\ (F\ (L\ 1, L\ 9), F\ (L\ 5, L\ 13)), F\ (F\ (L\ 3, L\ 11), F\ (L\ 7, L\ 15)))). \end{aligned}$$

Comparing the two expressions, it is fairly obvious that the first representation is more concise than the second. If we estimate the space usage of an k-ary constructor at $k + 1$ cells, we have that a perfect tree of rank n consumes $(2^n - 1)3 + (n + 1)2$ cells with the first and $(2^n - 1)3 + 2^n 2$ with the second representation.[1]

[1] We even assume that $F\ (\ell, r)$ occupies only three cells.

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

There is one further difference. Since Haskell is a non-strict language, *Tree a* comprises finite as well as partial and infinite trees. By contrast, *Perfect a* only accommodates finite trees.[2] Given this, and the fact that the nested datatype is more space economical, we are led to conclude that *Perfect a* is the datatype of choice when only perfectly balanced trees are required.

Next we tackle the question how to define recursion schemes for folding and unfolding perfect trees. The presentation largely follows the approach taken by Meijer and Hutton (1995), but as we shall see, at a higher level of abstraction. We must first recast recursive type definitions as fixed points of so-called *base functors*. Here is the base functor corresponding to *Perfect*:

$$\textbf{data}\ \mathit{Base}\ \mathit{perfect}\ a \quad = \quad \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{perfect}\ (\Delta\ a)).$$

The base functor is obtained by replacing the recursive occurrence of *Perfect* by a type variable. The type *Perfect* can now be defined as the fixpoint of this functor:

$$\textbf{newtype}\ \mathit{Perfect}\ a \quad = \quad \textbf{in}\ (\mathit{Base}\ \mathit{Perfect}\ a).$$

The constructor **in** and its inverse **out** given by $\textbf{out}\ (\textbf{in}\ a) = a$ establish an isomorphism between the functors *Perfect* and *Base Perfect*. Note that *Base* is not really a functor but a higher-order functor as it takes type constructors to type constructors, i.e. functors to functors. Its associated mapping function is even more unusual, since it takes polymorphic functions of type $\forall a.t\ a \to u\ a$ to polymorphic functions of type $\forall b.\mathit{Base}\ t\ b \to \mathit{Base}\ u\ b$.

$$\begin{array}{lcl}
\textbf{base} & :: & (\forall a.t\ a \to u\ a) \to (\forall b.\mathit{Base}\ t\ b \to \mathit{Base}\ u\ b) \\
\textbf{base}\ \varphi & = & \mathit{Zero} \bigtriangledown \mathit{Succ} \cdot \varphi \\
(f \bigtriangledown g)\ (\mathit{Zero}\ a) & = & f\ a \\
(f \bigtriangledown g)\ (\mathit{Succ}\ t) & = & g\ t\ .
\end{array}$$

Note that the parameter φ is applied as a function of type $t\ (\Delta\ a) \to u\ (\Delta\ a)$, which explains why it must be polymorphic. The type of **base** is a so-called *rank-2 type* (McCracken, 1984), which is not legal Haskell 98. A suitable extension, however, has been implemented in GHC (Peyton Jones, 1998) and in Hugs 98 (Jones and Peterson, 1999), both of which accept the definition if we change the type signature to $(\forall a.t\ a \to u\ a) \to \mathit{Base}\ t\ b \to \mathit{Base}\ u\ b$. The definition of cata- and anamorphisms is now entirely straightforward except perhaps for the types:

$$\begin{array}{lcl}
(\!|\text{-}|\!) & :: & (\forall a.\mathit{Base}\ t\ a \to t\ a) \to (\forall b.\mathit{Perfect}\ b \to t\ b) \\
(\!|\varphi|\!) & = & \varphi \cdot \textbf{base}\ (\!|\varphi|\!) \cdot \textbf{out} \\
[\!|\text{-}|\!] & :: & (\forall a.t\ a \to \mathit{Base}\ t\ a) \to (\forall b.t\ b \to \mathit{Perfect}\ b) \\
[\!|\psi|\!] & = & \textbf{in} \cdot \textbf{base}\ [\!|\psi|\!] \cdot \psi.
\end{array}$$

Both $(\!|\text{-}|\!)$ and $[\!|\text{-}|\!]$ map polymorphic functions to polymorphic functions. Catamorphisms on perfect trees usually take the form $(\!|f \bigtriangledown g|\!)$ with $f :: a \to t\ a$ and

[2] Of course, *Perfect a* also contains partial elements such as $\mathit{Succ}\ \bot$ and the infinite element $\textbf{let}\ t = \mathit{Succ}\ t\ \textbf{in}\ t$ but these elements hardly qualify as trees.

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

$g :: t\ (\Delta\ a) \to t\ a$, which we will abbreviate to $(\!|f,g|\!)$. Anamorphisms are typically written as $[\!(p \to Zero \cdot f, Succ \cdot g)\!]$ with $p :: t\ a \to Bool$, $f :: t\ a \to a$, and $g :: t\ a \to t\ (\Delta\ a)$. The expression $(p \to f, g)$, McCarthy's conditional form, is given by

$$(p \to f, g)\ a \quad = \quad \textbf{if}\ p\ a\ \textbf{then}\ f\ a\ \textbf{else}\ g\ a.$$

For better readability we abbreviate the unwieldy $[\!(p \to Zero \cdot f, Succ \cdot g)\!]$ to $[\!|p,f,g|\!]$.

Now for the utterly revolting part. How do we flatten a perfect tree of type $Perfect\ a$? The catamorphism $(\!|f,g|\!)$ takes a tree of the form $S^n(Z\ t)$ to $f^n(g\ t)$. It is immediate that the latter expression realizes a simple loop, which leads us to suspect that we must merely adapt the iterative variant of $flatten_k$. Inspecting the types of $f :: a \to [a]$ and $g :: [\Delta a] \to [a]$ confirms this suspicion:

$$\begin{array}{lcl} flatten, shuffle & :: & Perfect\ a \to [a] \\ flatten & = & (\!|wrap, interleave \cdot unzip|\!) \\ shuffle & = & (\!|wrap, cat \cdot unzip|\!). \end{array}$$

Loosely speaking, $Perfect\ a$ captures the recursion scheme of iterative tree algorithms. Building a perfect tree is, of course, also done iteratively:

$$\begin{array}{lcl} unflatten, unshuffle & :: & [a] \to Perfect\ a \\ unflatten & = & [\!|single, unwrap, zip \cdot uncat|\!] \\ unshuffle & = & [\!|single, unwrap, zip \cdot uninterleave|\!]. \end{array}$$

The function $single$, which tests a list for being a singleton, is defined by $single\ x = not\ (null\ x) \wedge null\ (tail\ x)$. The bit-reversal permutation can now be defined as the composition of an ana- and a catamorphism. The question naturally arises as to whether it is possible to remove the intermediate data structure built by the anamorphism and consumed by the catamorphism. Let us see what we can obtain by a little calculation. Setting

$$h \quad = \quad (\!|f,g|\!) \cdot [\!|p,f',g'|\!]$$

we argue

$$\begin{array}{lll} h & = & \{\ \text{specification}\ \} \\ & & (\!|f,g|\!) \cdot [\!|p,f',g'|\!] \\ & = & \{\ \text{definition } (\!|\text{-},\text{-}|\!) \text{ and } [\!|\text{-},\text{-},\text{-}|\!]\ \} \\ & & (f\ \triangledown\ g) \cdot \textbf{base}\ (\!|f,g|\!) \cdot \textbf{out} \cdot \textbf{in} \cdot \textbf{base}\ [\!|p,f',g'|\!] \cdot (p \to Z \cdot f', S \cdot g') \\ & = & \{\ \textbf{out} \cdot \textbf{in} = id,\ \textbf{base}\ \text{functor, and specification}\ \} \\ & & (f\ \triangledown\ g) \cdot \textbf{base}\ h \cdot (p \to Z \cdot f', S \cdot g') \\ & = & \{\ h \cdot (p \to f, g) = (p \to h \cdot f, h \cdot g) \Longleftarrow h\ \text{strict}\ \} \\ & & (p \to (f\ \triangledown\ g) \cdot \textbf{base}\ h \cdot Z \cdot f', (f\ \triangledown\ g) \cdot \textbf{base}\ h \cdot S \cdot g') \\ & = & \{\ \text{definition } \textbf{base},\ (f\ \triangledown\ g) \cdot Z = f, \text{ and } (f\ \triangledown\ g) \cdot S = g\ \} \\ & & (p \to f \cdot f', g \cdot h \cdot g'). \end{array}$$

Thus, we can express $(\!|f,g|\!) \cdot [\!|p,f',g'|\!]$ as the least fixed point of the recursion

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

equation $h = (p \rightarrow f \cdot f', g \cdot h \cdot g')$. It is interesting to take a closer look at h's typing: assuming the following types for the ingredient functions

$$\begin{array}{lcllcl} p & :: & t\ a \rightarrow \mathit{Bool} & & & \\ f' & :: & t\ a \rightarrow a & \quad f & :: & a \rightarrow u\ a \\ g' & :: & t\ a \rightarrow t\ (\Delta\ a) & \quad g & :: & u\ (\Delta\ a) \rightarrow u\ a \end{array}$$

we infer that h has type $t\ a \rightarrow u\ a$ while the recursive call is of type $t\ (\Delta a) \rightarrow u\ (\Delta a)$. In the ith level of recursion h has type $t\ (\Delta^i a) \rightarrow u\ (\Delta^i a)$. This means that h is a so-called *polymorphically recursive* function (Mycroft, 1984). It should be noted that the Hindley–Milner type system, which underlies most of today's functional programming languages, does not allow polymorphic recursion. Furthermore, a suitable extension of the type system has been shown to be undecidable (Henglein, 1993). Haskell allows polymorphic recursion only if an explicit type signature is provided for the respective function.

Now, by applying the fusion law to *flatten* · *unshuffle*, we obtain a surprisingly concise implementation of the bit-reversal permutation:

$$\begin{array}{lcl} \mathit{brp} & :: & [a] \rightarrow [a] \\ \mathit{brp} & = & (\mathit{single} \rightarrow \mathit{id}, \mathit{cat} \cdot \mathit{unzip} \cdot \mathit{brp} \cdot \mathit{zip} \cdot \mathit{uninterleave}). \end{array}$$

Note that *brp* accepts arbitrary non-empty lists. However, only the first $2^{\lfloor \log_2 n \rfloor}$ elements of the input list are actually used. The remaining elements are discarded by the invocations of *zip*.

6 Final remarks

The nested datatype *Perfect* nicely incorporates the structural properties of perfectly balanced, binary leaf trees. Its definition essentially proceeds *bottom-up*: a perfect tree of rank $n + 1$ is defined as a perfect tree of rank n containing pairs of elements. Consequently, the recursion operators for folding and unfolding perfect trees capture *iterative* algorithms. By contrast, the regular datatype *Tree* proceeds in a *top-down* manner; its associated recursion operators capture *recursive* algorithms. Unsurprisingly, not every function on perfect trees can be expressed as an iteration. For that reason, a generalization of the fold operator has been proposed (Bird and Paterson, 1999), that allows us to implement iterative as well as recursive algorithms or even mixtures of both styles.

The bit-reversal permutation is only defined for lists of length $n = 2^k$. The construction of binary leaf trees, however, makes sense for lists of arbitrary length. In the general case, the recursive and the iterative versions of *unflatten* and *unshuffle* yield differently shaped trees. The recursive version constructs a leaf-oriented *Braun tree* (Braun and Rem, 1983), which is characterized by the following balance condition: each node *Fork* (ℓ, r) satisfies $\mathit{size}\ r \leqslant \mathit{size}\ \ell \leqslant \mathit{size}\ r + 1$. The iterative version yields a *leftist left-complete tree* (Dielissen and Kaldewaij, 1995), where the offsprings of the nodes on the right spine form a sequence of perfect trees of decreasing height. Both algorithms are mentioned in Bird (1997). The two techniques of constructing leaf trees are closely related to top-down and bottom-up versions of *merge sort*

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

(Paulson, 1996). In fact, the different merge sort implementations may be obtained by fusing *unflatten* with $([wrap, merge])$, where $([-,-])$ is the standard catamorphism for *Tree*. Interestingly, an input which provokes the worst-case for the respective merge sort is then constructed by applying $flatten \cdot unshuffle$ to an ordered list. This permutation has the effect that each application of *merge* must interleave its argument lists.

Acknowledgements

I am grateful to Richard Bird, Jeremy Gibbons and Geraint Jones for suggesting the 'higher-order' naturality law for *unzip*, on which the development in section 4.2 is based.

References

Bird, R. (1998) *Introduction to Functional Programming using Haskell*. 2nd ed. Prentice Hall.

Bird, R. and Meertens, L. (1998) Nested datatypes. In: Jeuring, J. (editor), *4th International Conference on Mathematics of Program Construction, MPC'98*, pp. 52–67. Marstrand, Sweden. *Lecture Notes in Computer Science* **1422**. Springer-Verlag.

Bird, R. and Paterson, R. (1999) Generalised folds for nested datatypes. *Formal Aspects of Computing*. **11**(2), 200–222, September 1999.

Bird, R. S. (1997) Functional Pearl: On building trees with minimum height. *J. Functional Programming*, **7**(4), 441–445.

Braun, W. and Rem, M. (1983) *A logarithmic implementation of flexible arrays*. Memorandum MR83/4, Eindhoven University of Technology.

Dielissen, V. J. and Kaldewaij, A. (1995) A simple, efficient, and flexible implementation of flexible arrays. *3rd International Conference on Mathematics of Program Construction, MPC'95. Lecture Notes in Computer Science* **947**, pp. 232–241. Springer-Verlag.

Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Trans. Programming Languages and Systems*, **15**(2), 253–289.

Jones, M. P. and Peterson, J. C. (1999) *Hugs 98 user manual*. Available from `http://www.haskell.org/hugs`.

McCracken, N. J. (1984) The typechecking of programs with implicit type structure. In: Kahn, G., MacQueen, D. B. and Plotkin, G. D. (editors), *Semantics of Data Types: International Symposium*, Sophia-Antipolis, France. *Lecture Notes in Computer Science* **173**, pp. 301–315. Springer-Verlag.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91*, pp. 124–144. Cambridge, MA, USA. *Lecture Notes in Computer Science* **523**. Springer-Verlag.

Meijer, E. and Hutton, G. (1995) Bananas in space: Extending fold and unfold to exponential types. *7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 International Conference on Functional Programming Languages and Computer Architecture, FPCA'95*, pp. 324–333. La Jolla, San Diego, CA, USA. ACM-Press.

Mycroft, A. (1984) Polymorphic type schemes and recursive definitions. In: Paul, M. and Robinet, B. (editors), *Proceedings of the International Symposium on Programming, 6th Colloquium*, pp. 217–228. Toulouse, France. *Lecture Notes in Computer Science* **167**. Springer-Verlag.

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press

Paulson, L. C. (1996) *ML for the Working Programmer. 2nd edn.* Cambridge University Press.

Peyton Jones, Simon. (1998) *Explicit quantification in Haskell.* URL: `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.

Peyton Jones, S. and Hughes, J. (editors) (1999) *Haskell 98—A non-strict, purely functional language.* URL: `http://www.haskell.org/definition`

Wadler, P. (1989) Theorems for free! *4th International Conference on Functional Programming Languages and Computer Architecture, FPCA'89*, pp. 347–359. London, UK. ACM-Press.

https://doi.org/10.1017/S0956796800003701 Published online by Cambridge University Press