Perfect Trees and Bit-reversal Permutations

March 1999

RALF HINZE

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

Abstract

A famous algorithm is the Fast Fourier Transform, or 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

$$\mathit{brp}\ [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] \quad = \quad [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7] \ .$$

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* as the unique function which satisfies

$$\mathit{at}\ i \cdot \mathit{brp} \quad = \quad \mathit{at}\ (\mathit{rev}\ i) \ , \tag{1}$$

for all $i \in \{0, \ldots, n-1\}$. The function *at* denotes list indexing and *rev* computes the bit-reversal of a natural number. Assuming that list indexing takes constant time and given a function *rev* that runs in $\Theta(k)$ time it is straightforward to implement *brp* such that it takes $\Theta(nk)$ time to permute a list of length $n = 2^k$. Some extra cleverness is necessary to make *brp* run in linear time, see (Cormen *et al.*, 1991, Problem 18.1). Now, the question is whether *brp* 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

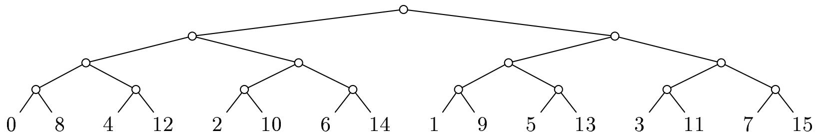

Fig. 1. The bit reversal permutation of the list $[0\,..\,15]$.

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

For a start, 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 length.

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

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

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 Fig. 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 $\Delta^4\ \mathit{Int}$. 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.

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

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 $(\!|{-}|\!)_n$, takes a function of type $\Delta a \to a$ and replaces each node in

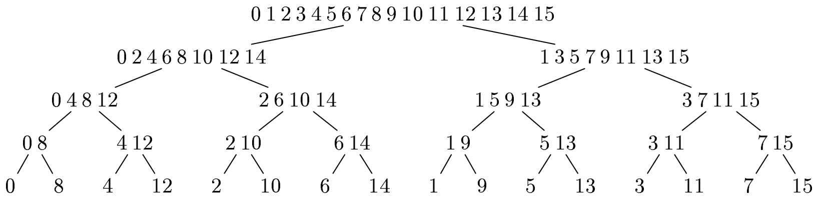

Fig. 2. Constructing the bit reversal permutation of $[0\,..\,15]$.

its input with this function.

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

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.

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

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

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

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

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

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

The first two laws show that ranked cata- and anamorphisms can be expressed as compositions of mapping functions. The third and the fourth law 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* 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 which avoids the use of an indexing operation works by building a perfect tree and then flattening it into a list. Either during the first or during the second phase the elements are shuffled in order to obtain the desired bit-reversal permutation. Fig. 2 illustrates the build of a perfect tree which has the bit-reversal permutation of the input list as 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 halve, and the results are finally combined. Here and in what follows we assume that the input list has 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 *halve*, partitions a list according to the most significant bit of the indices; the second, called *divide*, according to the least significant bit.

$$\begin{array}{lcl} \mathit{halve}\,[\,a_0,\ldots,a_{i-1}\,] & = & ([\,a_0,\ldots,a_{j-1}\,],[\,a_j,\ldots,a_{i-1}\,])\ \textbf{where}\ j = i \,/\, 2 \\ \mathit{divide}\,[\,a_0,a_1,a_2,a_3,\ldots\,] & = & ([\,a_0,a_2,\ldots\,],[\,a_1,a_3,\ldots\,]) \end{array}$$

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

$$\begin{array}{lcl} \mathit{build}_k, \mathit{shuffle}_k & :: & [\,a\,] \to \Delta^k\ a \\ \mathit{build}_k & = & \Delta^k\ \mathit{unwrap} \cdot [\![\mathit{halve}]\!]_k \\ \mathit{shuffle}_k & = & \Delta^k\ \mathit{unwrap} \cdot [\![\mathit{divide}]\!]_k \end{array}$$

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 build_k and $\mathit{shuffle}_k$ may be considered as *radix trees*: $\mathit{build}_k\ x$ represents the finite map $i \mapsto \mathit{at}\ i\ x$ while $\mathit{shuffle}_k\ x$ represents $i \mapsto \mathit{at}\ (\mathit{rev}\ i)\ x$.

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

$$\begin{array}{ll} & \mathit{flatten}_k \cdot \mathit{build}_k = \mathit{id} \\ \Longleftarrow & \quad \{\ \text{definition } \mathit{build}_k\ \} \\ & \mathit{flatten}_k \cdot \Delta^k\ \mathit{unwrap} \cdot [\![\mathit{halve}]\!]_k = \mathit{id} \\ \Longleftarrow & \quad \{\ \mathit{cat} \cdot \mathit{halve} = \mathit{id} \text{ and } (4)\ \} \\ & \mathit{flatten}_k \cdot \Delta^k\ \mathit{unwrap} = (\![\mathit{cat}]\!)_k \\ \Longleftarrow & \quad \{\ \Delta \text{ functor and } \mathit{wrap} \cdot \mathit{unwrap} = \mathit{id}\ \} \\ & \mathit{flatten}_k = (\![\mathit{cat}]\!)_k \cdot \Delta^k\ \mathit{wrap} \end{array}$$

The derivation of $\mathit{shuffle}_k$'s inverse proceeds completely analogous. To summarize

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

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

$$\mathit{brp}_k = (\![\mathit{cat}]\!)_k \cdot [\![\mathit{divide}]\!]_k = (\![\mathit{interleave}]\!)_k \cdot [\![\mathit{halve}]\!]_k$$

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 $build_k$ can be improved to $\Theta(n)$ using a technique called *accumulation* (Bird, 1998). The dual technique termed *tupling* may be used to improve the complexity of $flatten_k$. However, the overall gain is only a constant factor since $shuffle_k$ and $unshuffle_k$ 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 will try as much as possible to derive them from the recursive functions defined in the previous section. Since flattening a tree is simpler than building one we start 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 $flatten_0 = wrap$). The function $step$ has type $[\Delta a] \to [a]$, ie 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 $[a_0, b_0, a_1, b_1, \ldots]$. It can be defined by $interleave \cdot unzip$ where $unzip$ is given by

$$\begin{array}{lcl} unzip & :: & [\Delta\ a] \to \Delta\ [a] \\ unzip\ [(a_0, b_0), (a_1, b_1), \ldots] & = & ([a_0, a_1, \ldots], [b_0, b_1, \ldots])\ . \end{array}$$

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 $unshuffle_k$ explicit, see below. The crucial property of $step = interleave \cdot unzip$ is that it distributes over cat, ie

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

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

Now, to prove (6) we reason

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

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

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

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

Given these equations it is almost trivial to derive iterative definitions for $build_k$ and $shuffle_k$. We get

$$
\begin{array}{lcl}
build_k & = & unwrap \cdot (zip \cdot divide)^k \\
shuffle_k & = & unwrap \cdot (zip \cdot halve)^k \ .
\end{array}
$$

Both $zip \cdot divide$ and $zip \cdot halve$ 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.

$$brp_k = (interleave \cdot unzip)^k \cdot (zip \cdot halve)^k = (cat \cdot unzip)^k \cdot (zip \cdot divide)^k$$

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 (Peyton Jones [editor] *et al.*, 1998). 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 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.

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

This definition is somewhat unusual in that the recursive component, $\mathit{Perfect}\ (\Delta\ a)$, 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* (Bird & Meertens, 1998). Abbreviating the constructor names to their first letter the tree of Fig. 1 is represented by the following term.

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

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 as a matter of fact corresponds to the following fixpoint equation.

$$\mathit{Bush} \quad = \quad \mathit{Id} + \Delta \cdot \mathit{Bush}$$

Clearly, *Bush* 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.

$$\textbf{data}\ \mathit{Bush}\ a \quad = \quad \mathit{Leaf}\ a \mid \mathit{Fork}\ (\Delta\ (\mathit{Bush}\ a))$$

Encoded as an element of *Bush Int* the tree of Fig. 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 one. 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]

There is one further difference. Since Haskell is a non-strict language, *Bush 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 lead 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 in (Meijer & Hutton, 1995), however, 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

[1] We assume that $F\ (\ell, r)$ occupies only 3 cells.
[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.

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, ie 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 $\forall b.\mathit{Base}\ t\ b \to \mathit{Base}\ u\ b$.

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

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. A suitable extension, however, has been implemented in GHC (Peyton Jones, 1998) and in Hugs 98 (Jones, 1998) 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 $(\!|f \triangledown g|\!)$ with $f :: a \to t\ a$ and $g :: t\ (\Delta\ a) \to t\ a$ which we will abbreviate to $(\!|f, g|\!)$. Anamorphisms are typically written as $[\!((p \to \mathit{Zero} \cdot f, \mathit{Succ} \cdot g))\!]$ with $p :: t\ a \to \mathit{Bool}$, $f :: t\ a \to a$, 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 $[\!((p \to \mathit{Zero} \cdot f, \mathit{Succ} \cdot g))\!]$ 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 $\mathit{flatten}_k$. Inspecting the types of $f :: a \to [a]$ and $g :: [\Delta a] \to [a]$ confirms this suspicion.

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

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

$$\begin{array}{lcl} \mathit{build}, \mathit{shuffle} & :: & [a] \to \mathit{Perfect}\ a \\ \mathit{build} & = & [\!(\mathit{single}, \mathit{unwrap}, \mathit{zip} \cdot \mathit{halve})\!] \\ \mathit{shuffle} & = & [\!(\mathit{single}, \mathit{unwrap}, \mathit{zip} \cdot \mathit{divide})\!] \end{array}$$

The function *single*, which tests a list for being a singleton, is defined by $\mathit{single}\ x = \mathit{not}\ (\mathit{null}\ x) \wedge \mathit{null}\ (\mathit{tail}\ x)$. 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 build by the anamorphism and consumed by the catamorphism. Let's see what we can obtain by a little calculation. Setting

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

we argue

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

Thus, we can express $(\![f, g]\!) \cdot [\!(p, f', g')\!]$ as the least fixed point of the recursion equation $h = (p \to f \cdot f', g \cdot h \cdot g')$. It is interesting to take a closer look at h's typing: assuming the following types for the ingredient functions

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

we infer that h has type $t\ a \to u\ a$ while the recursive call is of type $t\ (\Delta a) \to u\ (\Delta a)$. In the i-th level of recursion h has type $t\ (\Delta^i a) \to u\ (\Delta^i a)$. 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* · *shuffle* we obtain a a surprisingly concise implementation of the bit-reversal permutation.

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

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 *Bush* 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 & Paterson, 1998) that allows 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 *build* and *shuffle* yield differently shaped trees. The recursive version constructs a *Braun tree* (Braun & Rem, 1983), which is characterized by the following balance condition: each node $Fork\ (\ell, r)$ satisfies $size\ r \leqslant size\ \ell \leqslant size\ r + 1$. The iterative version yields a *leftist left-complete tree* (Dielissen & 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* (Paulson, 1996). In fact, the different merge sort implementations may be obtained by fusing *build* with $(\![wrap, merge]\!)$ where $(\![\text{-}, \text{-}]\!)$ is the standard catamorphism for *Bush*. Interestingly, an input which provokes the worst-case for the respective merge sort is then constructed by applying *flatten* · *shuffle* to an ordered list. This permutation has the effect that each application of *merge* must interleave its argument lists.

References

Bird, Richard. (1998). *Introduction to functional programming using Haskell.* 2nd edn. London: Prentice Hall Europe.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J.

(ed), *Fourth international conference on mathematics of program construction, MPC'98, Marstrand, Sweden.* Lecture Notes in Computer Science, vol. 1422. Springer Verlag.

Bird, Richard, & Paterson, Ross. (1998). *Generalised folds for nested datatypes.* Submitted for publication.

Bird, Richard S. (1997). Functional Pearls: On building trees with minimum height. *Journal of functional programming*, **7**(4), 441–445.

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

Dielissen, Victor J., & Kaldewaij, Anne. (1995). A simple, efficient, and flexible implementation of flexible arrays. *Pages 232–241 of: Third international conference on mathematics of program construction (MPC'95).* Lecture Notes in Computer Science, vol. 947. Springer Verlag.

Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM transactions on programming languages and systems*, **15**(2), 253–289.

Jones, Mark. (1998). *A technical summary of the new features in Hugs 1.3c.* unpublished.

McCracken, Nancy Jean. (1984). The typechecking of programs with implicit type structure. *Pages 301–315 of:* Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of data types: International symposium, Sophia-Antipolis, France.* Lecture Notes in Computer Science, vol. 173. Berlin: Springer Verlag.

Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of: 5th ACM conference on functional programming languages and computer architecture, Cambridge, MA.* Lecture Notes in Computer Science, vol. 523. Springer Verlag.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Pages 324–333 of: Conf. record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 intl. conf. on functional programming languages and computer architecture, FPCA'95, La Jolla, San Diego, CA, USA.* ACM Press.

Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. Paul, M., & Robinet, B. (eds), *International symposium on programming, 6th colloquium Toulouse.* LNCS 167.

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 [editor], Simon, Hughes [editor], John, Augustsson, Lennart, Barton, Dave, Boutel, Brian, Burton, Warren, Fraser, Simon, Fasel, Joseph, Hammond, Kevin, Hinze, Ralf, Hudak, Paul, Johnsson, Thomas, Jones, Mark, Launchbury, John, Meijer, Erik, Peterson, John, Reid, Alastair, Runciman, Colin, & Wadler, Philip. 1998 (December). *Haskell 98 — A non-strict, purely functional language.*