Program optimisation, naturally

Richard Bird
Programming Research Group
Oxford University

Jeremy Gibbons
School of Computing and Mathematical Sciences
Oxford Brookes University

Geraint Jones
Programming Research Group
Oxford University

July 1999

Abstract

It is well-known that each polymorphic function satisfies a certain equational law, called a *naturality* condition. Such laws are part and parcel of the basic toolkit for improving the efficiency of functional programs. More rarely, some polymorphic functions also possess a *higher-order* naturality property. One example is the operation *unzip* that takes lists of pairs to pairs of lists. Surprisingly, this property can be invoked to improve the asymptotic efficiency of some divide-and-conquer algorithms from $O(n \log n)$ to $O(n)$. The improved algorithms make use of polymorphic recursion, and can be expressed neatly using nested datatypes, so they also serve as evidence of the practical utility of these two concepts.

1 Introduction

One of the first things that any functional programmer learns is how to turn the naive quadratic algorithm for reversing a list into a linear-time algorithm by using an accumulating parameter. In this paper we derive another, quite different linear-time algorithm for reversing a list. The derivation relies on a *higher-order naturality* [4] property of the function *unzip*, which turns a list of pairs into a pair of lists in the obvious way. The final program uses *polymorphic recursion* [8], and can be expressed rather neatly using *nested datatypes* [1]. Although neither feature is an essential component of the fast algorithm, they do simplify it, providing more evidence of the practical utility of these two concepts.

Hinze [3] presents another application of these techniques to the problem of computing the bit-reversal permutation of a list. We describe this problem and its solution briefly in Section 6. Indeed, it was Hinze's application that inspired us to write this paper in the first place. Whether or not the technique has wider application remains to be seen.

2 Reversing a list

A divide-and-conquer algorithm for reversing a list can be constructed from the observation that the reverse of the concatenation of two lists is the concatenation, in the opposite order, of their reverses. For simplicity in what follows, we restrict attention to *powerlists* [7], a powerlist being a list with length a power of two. Then the function rev_k, which reverses a list of length 2^k, is defined inductively by

$$\begin{array}{lcl} rev_0\,[a] & = & [a] \\ rev_{k+1}\,(x +\!\!+\, y) & = & rev_k\; y +\!\!+\, rev_k\; x \end{array}$$

It is assumed that both x and y have length 2^k in the second equation. In a pointfree style, we have:

$$\begin{array}{lcl} rev_0 & = & id \\ rev_{k+1} & = & cat \circ swap \circ pair\; rev_k \circ uncat \end{array}$$

Here, $cat :: Pair\;[a] \to [a]$ concatenates two lists of the same length to give a list of twice the length, and *uncat* is its inverse; *pair* is the action of the pairing functor on functions; and $swap :: Pair\; a \to Pair\; a$ swaps the components of a pair. On a non-singleton list, this algorithm involves two recursive calls on lists of half the length; therefore the time taken to reverse a list of length n is $O(n \log n)$. If we interpret *uncat* as dividing a list of non-unit but otherwise arbitrary length into two, then the equation

$$rev \;=\; cat \circ swap \circ pair\; rev \circ uncat$$

remains valid. Consequently, the algorithm can be adapted to handle lists of arbitrary length.

Here is another, less obvious, divide-and-conquer characterisation of reverse. Instead of using concatenation and its inverse to combine and split the list, we use 'riffle' (what Misra [7] calls 'tie') and its inverse. Informally, $unriffle :: [a] \to Pair\;[a]$ applied to a non-singleton list puts the even-numbered elements (counting from zero) in the first component of its result, and the odd-numbered elements in the second component; *riffle* is the inverse operation. Again we restrict attention to powerlists. If one considers the indices of the elements of the list in binary notation, then it becomes clear that *riffle* is a dual to *uncat*: the former partitions on the least significant bit, the latter on the most significant.

Assuming the definitions of *riffle* and *unriffle*, the algorithm is

$$
\begin{array}{lcl}
\mathit{rev}_0 & = & \mathit{id} \\
\mathit{rev}_{k+1} & = & \mathit{riffle} \circ \mathit{swap} \circ \mathit{pair}\ \mathit{rev}_k \circ \mathit{unriffle}
\end{array}
$$

For example,

$$
\begin{array}{cl}
 & \mathit{rev}_3\,[0,1,2,3,4,5,6,7] \\
= & \quad \{\ \text{definition of } \mathit{rev}\ \} \\
 & \mathit{riffle}\ (\mathit{swap}\ (\mathit{pair}\ \mathit{rev}_2\ (\mathit{unriffle}\,[0,1,2,3,4,5,6,7]))) \\
= & \quad \{\ \text{definition of } \mathit{unriffle}\ \} \\
 & \mathit{riffle}\ (\mathit{swap}\ (\mathit{pair}\ \mathit{rev}_2\ ([0,2,4,6],[1,3,5,7]))) \\
= & \quad \{\ \text{induction}\ \} \\
 & \mathit{riffle}\ (\mathit{swap}\ ([6,4,2,0],[7,5,3,1])) \\
= & \quad \{\ \text{definition of } \mathit{swap}\ \} \\
 & \mathit{riffle}\ ([7,5,3,1],[6,4,2,0]) \\
= & \quad \{\ \text{definition of } \mathit{riffle}\ \} \\
 & [7,6,5,4,3,2,1,0]
\end{array}
$$

The function *unriffle* can be built from two components. The first, $\mathit{group} :: [a] \to [\mathit{Pair}\ a]$, acts only on non-singleton powerlists; it groups the first two elements into a pair, the third and fourth elements into a second pair, and so on. For example,

$$\mathit{group}\,[0,1,2,3,4,5,6,7] \quad = \quad [(0,1),(2,3),(4,5),(6,7)]$$

The second component, $\mathit{unzip} :: [\mathit{Pair}\ a] \to \mathit{Pair}\,[a]$, teases apart a list of pairs into two lists, the first list consisting of the first components of each pair and the second list of the second components. For example,

$$\mathit{unzip}\,[(0,1),(2,3),(4,5),(6,7)] \quad = \quad ([0,2,4,6],[1,3,5,7])$$

These two functions, which are easy to implement, have inverses *ungroup* and *zip*, respectively. Now we can define

$$
\begin{array}{lcl}
\mathit{unriffle} & = & \mathit{unzip} \circ \mathit{group} \\
\mathit{riffle} & = & \mathit{ungroup} \circ \mathit{zip}
\end{array}
$$

Unlike the algorithm based on *cat* and *uncat*, the algorithm using *riffle* and *unriffle* has to be modified to handle lists of arbitrary length. The equation

$$\mathit{rev} \quad = \quad \mathit{riffle} \circ \mathit{swap} \circ \mathit{pair}\ \mathit{rev} \circ \mathit{unriffle}$$

holds only for lists of even length. For lists of odd length, the corresponding equation is

$$\mathit{rev} \quad = \quad \mathit{riffle} \circ \mathit{pair}\ \mathit{rev} \circ \mathit{unriffle}$$

in which the *swap* operation does not appear.

The second algorithm also takes $O(n \log n)$ steps to reverse a list of length n. However, as we shall see in the next section, we can exploit a law of *unzip* to combine the two recursive calls into a single call, giving a linear-time algorithm.

3 A law of unzip

The function $unzip :: [Pair\ a] \rightarrow Pair\ [a]$ is a *natural transformation*, which is to say that it satisfies the law

$$unzip \circ map\ (pair\ f) \quad = \quad pair\ (map\ f) \circ unzip$$

for all functions f. In words, since *unzip* does not inspect the elements of the input list, one can change these elements systematically either before or after unzipping the list, with exactly the same result in both cases. Wadler [10] has shown that *every* polymorphic function satisfies a naturality law whose form is deducible simply from its type. In the general case there are certain strictness conditions on the function f, although they do not appear in the law for *unzip*.

However, *unzip* enjoys another naturality property. This states that, for any natural transformation $phi :: [a] \rightarrow [a]$, we have

$$pair\ phi \circ unzip \quad = \quad unzip \circ phi$$

Formally, *unzip* is a *higher-order natural transformation* [4]. Informally, since *phi* is a natural transformation, it does the same thing to any list. Therefore, unzipping a list of pairs and doing *phi* to each list does the same to corresponding elements of the lists, and so is equivalent to doing *phi* once to the list of pairs and then unzipping.

To prove this law we need to give some definitions. We define the type of pairs and accompanying operations as follows:

$$
\begin{array}{lcl}
\mathsf{type}\ Pair\ a & = & (a, a) \\[4pt]
pair & :: & (a \rightarrow b) \rightarrow Pair\ a \rightarrow Pair\ b \\
pair\ f\ (a, a') & = & (f\ a, f\ a') \\[4pt]
fst & :: & Pair\ a \rightarrow a \\
fst\ (a, a') & = & a \\[4pt]
snd & :: & Pair\ a \rightarrow a \\
snd\ (a, a') & = & a' \\[4pt]
fork & :: & Pair\ (a \rightarrow b) \rightarrow a \rightarrow Pair\ b \\
fork\ (f, g)\ a & = & (f\ a, g\ a) \\[4pt]
swap & :: & Pair\ a \rightarrow Pair\ a \\
swap & = & fork\ (snd, fst)
\end{array}
$$

The function *unzip* is defined by the equation

$$unzip \quad = \quad fork\ (map\ fst, map\ snd)$$

We can now reason:

$$
\begin{array}{ll}
 & pair\ f \circ unzip \\
= & \qquad \{\ \text{definition of } unzip\ \} \\
 & pair\ f \circ fork\ (map\ fst, map\ snd)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{ pair-fork fusion } \} \\
&\quad \mathit{fork}\ (f \circ \mathit{map}\ \mathit{fst}, f \circ \mathit{map}\ \mathit{snd}) \\
&= \quad \{ \text{ naturality of } f \ \} \\
&\quad \mathit{fork}\ (\mathit{map}\ \mathit{fst} \circ f, \mathit{map}\ \mathit{snd} \circ f) \\
&= \quad \{ \text{ fork distributes over composition } \} \\
&\quad \mathit{fork}\ (\mathit{map}\ \mathit{fst}, \mathit{map}\ \mathit{snd}) \circ f \\
&= \quad \{ \text{ definition of } \mathit{unzip} \ \} \\
&\quad \mathit{unzip} \circ f
\end{aligned}
$$

In fact, with a little bit more notation, we can state a stronger result. For an arbitrary functor F, define unzip_F by

$$
\begin{aligned}
\mathit{unzip}_F &:: F\ (\mathit{Pair}\ a) \to \mathit{Pair}\ (F\ a) \\
\mathit{unzip}_F &= \mathit{fork}\ (F\ \mathit{fst}, F\ \mathit{snd})
\end{aligned}
$$

Then the higher-order naturality condition is that, for any natural transformation $f :: F\ a \to G\ a$, we have

$$
\mathit{unzip}_G \circ f \quad = \quad \mathit{pair}\ f \circ \mathit{unzip}_F
$$

The proof is almost identical. However, we do not need the extra generality for this paper.

4 Applying the law

Returning to the problem of reversing a list, we see that

$$
\begin{aligned}
&\quad \mathit{riffle} \circ \mathit{swap} \circ \mathit{pair}\ \mathit{rev}_k \circ \mathit{unriffle} \\
&= \quad \{ \text{ definition of } \mathit{unriffle} \ \} \\
&\quad \mathit{riffle} \circ \mathit{swap} \circ \mathit{pair}\ \mathit{rev}_k \circ \mathit{unzip} \circ \mathit{group} \\
&= \quad \{ \text{ higher-order naturality } (\mathit{rev} \text{ is natural}) \ \} \\
&\quad \mathit{riffle} \circ \mathit{swap} \circ \mathit{unzip} \circ \mathit{rev}_k \circ \mathit{group}
\end{aligned}
$$

Hence

$$
\begin{aligned}
\mathit{rev}_0 &= \mathit{id} \\
\mathit{rev}_{k+1} &= \mathit{riffle} \circ \mathit{swap} \circ \mathit{unzip} \circ \mathit{rev}_k \circ \mathit{group}
\end{aligned}
$$

in which there is but a single recursive call on a list of half the length, giving a linear-time algorithm.

We can simplify the algorithm a little further, since

$$
\begin{aligned}
&\quad \mathit{riffle} \circ \mathit{swap} \circ \mathit{unzip} \\
&= \quad \{ \text{ definition of } \mathit{riffle} \ \} \\
&\quad \mathit{ungroup} \circ \mathit{zip} \circ \mathit{swap} \circ \mathit{unzip}
\end{aligned}
$$

$$= \quad \{ \text{ since } swap \circ unzip = unzip \circ map\ swap \ \}$$
$$ungroup \circ zip \circ unzip \circ map\ swap$$
$$= \quad \{ \ zip \text{ and } unzip \text{ are inverses } \}$$
$$ungroup \circ map\ swap$$

Hence we can define

$$\begin{array}{lcl} rev_0 & = & id \\ rev_{k+1} & = & ungroup \circ map\ swap \circ rev_k \circ group \end{array}$$

Note that there is something peculiar going on with the types here. On the left-hand side of the second equation above, rev_{k+1} is acting on a list of elements of type A for some A. However, on the right-hand side, rev_k is acting on a list of *pairs* of elements of type A. This is an instance of *polymorphic recursion* [8]. The latest versions of Haskell allow polymorphic recursion like this, but they cannot infer the types of polymorphically recursive functions because the type inference problem is undecidable. As a consequence, the types need to be given by the programmer and merely checked by Haskell.

5 Nested datatypes

The above definitions can be implemented perfectly well in Haskell using ordinary lists. However, there is then no way to enforce the constraint that the lists should have length a power of two. Such 'structure invariants' can often be expressed using *nested datatypes* [1].

In particular, we can define the type of powerlists by

$$\mathsf{data}\ Plist\ a \ = \ Zero\ a \mid Succ\ (Plist\ (Pair\ a))$$

For example, the powerlist of naturals from 0 to 7 is represented by the expression

$$Succ\ (Succ\ (Succ\ (Zero\ (((0,1),(2,3)),((4,5),(6,7))))))$$

Notice how the constructors in the above expression encode in Peano numerals the 'depth' of the list. The type $Plist\ a$ is a nested datatype because the occurrence of $Plist$ on the right-hand side of the type definition has a different argument to the defining occurrence on the left-hand side.

The map operation on powerlists is given by

$$\begin{array}{lcl} plist & :: & (a \to b) \to Plist\ a \to Plist\ b \\ plist\ f\ (Zero\ a) & = & Zero\ (f\ a) \\ plist\ f\ (Succ\ x) & = & Succ\ (plist\ (pair\ f)\ x) \end{array}$$

This is another instance of polymorphic recursion, because the occurrence of $plist$ on the right-hand side of the second equation is at a different type to the defining occurrence on the left-hand side.

The other functions required for reversing a list are as follows.

$$
\begin{array}{lcl}
\mathit{group} & :: & \mathit{Plist}\ a \to \mathit{Plist}\ (\mathit{Pair}\ a) \\
\mathit{group}\ (\mathit{Succ}\ x) & = & x \\[1ex]
\mathit{ungroup} & :: & \mathit{Plist}\ (\mathit{Pair}\ a) \to \mathit{Plist}\ a \\
\mathit{ungroup}\ x & = & \mathit{Succ}\ x \\[1ex]
\mathit{cat} & :: & \mathit{Pair}\ (\mathit{Plist}\ a) \to \mathit{Plist}\ a \\
\mathit{cat}\ (\mathit{Zero}\ a, \mathit{Zero}\ b) & = & \mathit{Succ}\ (\mathit{Zero}\ (a, b)) \\
\mathit{cat}\ (\mathit{Succ}\ x, \mathit{Succ}\ y) & = & \mathit{Succ}\ (\mathit{cat}\ (x, y)) \\[1ex]
\mathit{unzip} & :: & \mathit{Plist}\ (\mathit{Pair}\ a) \to \mathit{Pair}\ (\mathit{Plist}\ a) \\
\mathit{unzip} & = & \mathit{fork}\ (\mathit{plist}\ \mathit{fst}, \mathit{plist}\ \mathit{snd})
\end{array}
$$

The fast reverse algorithm on powerlists is now implemented by

$$
\begin{array}{lcl}
\mathit{rev} & :: & \mathit{Plist}\ a \to \mathit{Plist}\ a \\
\mathit{rev}\ (\mathit{Zero}\ a) & = & \mathit{Zero}\ a \\
\mathit{rev}\ (\mathit{Succ}\ x) & = & \mathit{ungroup}\ (\mathit{plist}\ \mathit{swap}\ (\mathit{rev}\ (\mathit{group}\ x)))
\end{array}
$$

6 Bit-reversal permutations

Hinze [3] presents another application of the ideas above, to the problem of a *bit-reversal permutation* of a powerlist. For each i, the element at position i moves to position j, where the binary representation of j is the reverse of the binary representation of i. For example, the bit-reverse permutation of the list of characters "abcdefgh" is the string "aecgbfdh"; "fallible" and "fillable" permute to each other, and "sixtieth" is a fixed point. Bit-reversal permutations are a feature of in-place implementations [6] of the Cooley-Tukey Fast Fourier Transform algorithm [2, 5].

One divide-and-conquer characterisation of the bit-reversal permutation is as follows:

$$
\begin{array}{lcl}
\mathit{brp}\ (\mathit{Zero}\ a) & = & \mathit{Zero}\ a \\
\mathit{brp}\ (\mathit{Succ}\ x) & = & \mathit{cat}\ (\mathit{pair}\ \mathit{brp}\ (\mathit{unzip}\ x))
\end{array}
$$

This algorithm takes $O(n \log n)$ steps; however, the same higher-order naturality law applies (because *brp* is a natural transformation), and so we also have that

$$
\begin{array}{lcl}
\mathit{brp}\ (\mathit{Zero}\ a) & = & \mathit{Zero}\ a \\
\mathit{brp}\ (\mathit{Succ}\ x) & = & \mathit{cat}\ (\mathit{unzip}\ (\mathit{brp}\ x))
\end{array}
$$

This implementation takes only linear time.

7 Conclusion

We have shown an $O(n \log n)$ divide-and-conquer algorithm for reversing a powerlist, and shown how to improve it to linear time using a higher-order naturality property of the function *unzip*. This linear-time algorithm relies on polymorphic recursion.

Powerlists can be defined as a nested datatype, enforcing the structure invariant; we have therefore also provided some examples of programs on nested datatypes.

This is the first instance we have seen of a higher-order naturality property leading to an asymptotic improvement in efficiency of an algorithm. There are many other divide-and-conquer algorithms with similar patterns of recursion, and we are keen to find other examples of such improvements. However, higher-order natural transformations are rather thin on the ground; indeed, we have not been able to think of any non-trivial functional examples apart from zips and unzips, which are the motivating examples from Hoogendijk's thesis.

References

[1] Richard S. Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *LNCS 1422: Proceedings of Mathematics of Program Construction*, pages 52–67, Marstrand, Sweden, June 1998. Springer-Verlag.

[2] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[3] Ralf Hinze. Perfect trees and bit-reversal permutations. Universität Bonn, 1999.

[4] Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[5] Geraint Jones. Deriving the fast Fourier algorithm by calculation. In Kei Davis and John Hughes, editors, *Functional Programming, Glasgow 1989*. Springer-Verlag, 1990.

[6] A. M. Macnaghten and C. A. R. Hoare. Fast Fourier Transform free from tears. *Computer Journal*, 20(1):78–83, 1977.

[7] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994. Also in [9].

[8] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *LNCS 167: International Symposium on Programming*, pages 217–228. Springer-Verlag, 1984.

[9] A. W. Roscoe, editor. *A Classical Mind: Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[10] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.