Zippy Tabulations of Recursive Functions

Richard S. Bird

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
bird@comlab.ox.ac.uk

zippy: (adjective) 1. bright, fresh or lively. 2. speedy
Oxford Compact Dictionary

Abstract. This paper is devoted to the statement and proof of a theorem showing how recursive definitions whose associated call graphs satisfy certain shape conditions can be converted systematically into efficient bottom-up tabulation schemes. The increase in efficiency can be dramatic, typically transforming an exponential time algorithm into one that takes only quadratic time. The proof of the theorem relies heavily on the theory of zips developed by Roland Backhouse and Paul Hoogendijk.

1 Introduction

From one point of view all recursive functions look alike: if the input is simple the function value is computed directly; if the input is not simple, it is decomposed into simpler instances on which the function is computed recursively, and the various results are then combined to give the final value. In a divide and conquer algorithm, such as mergesort, the recursive instances do not overlap so the top-down computation is efficient. By contrast, in a dynamic programming algorithm the recursive instances do overlap, and the top-down computation is potentially very inefficient because the same subproblem may be solved many times.

The two main methods for improving matters are *memoization* and *tabulation*. In memoization the top-down structure of the computation is preserved but computed results are remembered and stored in a memo table for subsequent retrieval. In tabulation the computation switches to a bottom-up scheme in which the simple problems are solved first and then solutions to larger problems are placed in a table level by level.

The problem with tabulation is that the programmer has to think carefully about the structure of the table being built and the best way to store the entries to make them easily accessible for the next level. This is particularly true of the pure functional programmer who eschews arrays and relies on lists and trees instead. Wouldn't it be pleasant if the bottom-up tabulation algorithm could be derived automatically, or at least systematically from the top-down recursion?

The wish is not grantable in general. For example, [8] shows that choosing a good table design for a given decomposition is an NP-hard problem. However, it turns out that for one particular class of recursive functions, namely

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 92–109, 2008.

those involving decompositions which satisfy a rather severe shape constraint, a bottom-up tabulation scheme can indeed be derived automatically. The proof that everything works is the main contribution of the present paper. The proof is interesting because it depends heavily on the theory of zips developed by Backhouse and Hoogendijk, see [1,2,6,7]. Our main theorem can also be viewed as an extension and formalisation of some of bottom-up schemes described in [3].

2 Top-Down Computation

Let us begin by assembling the ingredients for describing a generic top-down recursion. The first thing to say is that although we use mostly Haskell notation to describe recursive functions, we do not assume a lazy semantics. Indeed for the theorem and proof to come our setting is a relational semantics based on allegories with certain properties, see [4]. But the allegorical undergrowth need not impede our progress.

The first assumption is that the input is an element of some instance of a *polymorphic* data type $L\,a$ with no empty structures. For concreteness think of $L\,a$ as being nonempty lists of elements of type a, and the input as an element of $L\,Integer$, a list of integers. For technical reasons, explained later on, we disallow data types that contain empty elements. We use single capital letters for the names of polymorphic types, and the same letter for the map function over the data type. So if $f :: a \to b$, then $L\,f :: L\,a \to L\,b$. In a word, L is a functor.

The next ingredient is a total function $sg :: L\,a \to Bool$ that determines whether the input is sufficiently simple for the result to be returned directly. The identifier sg connotes 'singleton', which is often the base case of a recursion though other interpretations are possible.

The companion to sg is a partial function $ex :: L\,a \to a$, total on arguments that satisfy sg, that extracts the singleton element. Both sg and ex are polymorphic, which means they satisfy the properties that $sg \cdot L\,f = sg$ and $ex \cdot L\,f = f \cdot ex$ for any total function f. In two words, sg and ex are natural transformations. In general, a polymorphic function $\alpha :: F\,a \to G\,a$ satisfies $\alpha \cdot F\,f = G\,f \cdot \alpha$ for any total function f. In particular, $ex :: L\,a \to I\,a$, where I is the identity functor, and $sg :: L\,a \to K\,a$, where $K\,a = Bool$ is the constant functor that returns $Bool$. Moreover, $K\,f$ is the identity function on $Bool$. We will exploit these naturality assumptions later on.

Next is a partial function $dc :: L\,a \to F\,(L\,a)$ which is total over non-singleton structures. In words, dc returns an F-structure of L-structures. The identifier dc connotes 'decompositions'. The function dc is, by assumption, another natural transformation so we have $F\,(L\,f) \cdot dc = dc \cdot L\,f$ for any total function f.

Everything is now in place for describing a top-down computation:

$$\begin{aligned} td &:: (a \to b) \to (F\,b \to b) \to L\,a \to b \\ td\,f\,g &= (sg \to f \cdot ex, g \cdot F\,(td\,f\,g) \cdot dc) \end{aligned}$$

The expression on the right is a McCarthy conditional $(p \to f, g)$. Applied to x the conditional returns $f\,x$ if $p\,x$ and $g\,x$ otherwise. The computation of td

reads: if the input is a singleton, extract the element and apply f; otherwise decompose the input into further instances using dc, apply $td\, f\, g$ recursively to each of these instances and collect the sub-results into one result by applying g.

Before going on to formulate a bottom-up tabulation scheme let us first consider some instructive examples of top-down computations.

3 Examples

Bear in mind that the main restriction on td is that the function dc should be polymorphic. That rules out quicksort as an example because dc has to inspect the elements in the argument list. There are many examples of recursive top-down computations with polymorphic decomposition functions, and we will look only at examples where the function $dc :: L\, a \to F\, (L\, a)$ is *layered* in the sense that dc produces an F-structure of L-structures in which all the L-structures have the same shape. This restriction turns out to be central in the results to come.

First, consider mergesort restricted to power lists. By definition a power list is a list whose length is a power of two. Here $L\, a$ is the type of power lists and $F = P$, where $P\, a = (a, a)$ so P is the data type of pairs. The function sg is a test for singletons and ex extracts a singleton value. The definition of dc is

$$dc\, xs = (take\ n\ xs, drop\ n\ xs) \quad \textbf{where}\ n = length\ xs\ \textbf{div}\ 2$$

Applied to a power list, dc returns a pair of power lists of the same length, so is layered. The function $td\ wrap\ merge$ sorts a power list of elements from any ordered type, where $wrap\ x = [x]$ and $merge$ merges a pair of ordered power lists. The call graph associated with dc is a perfect binary tree. The subproblems do not overlap and there is no sharing of subtrees.

Second, consider a recursion based on the decomposition function

$$\begin{aligned} dc &:: L\, a \to P\, (L\, a) \\ dc &= fork\, (init, tail) \end{aligned}$$

where $init$ and $tail$ respectively drop the last and first elements of a list of length at least two. Here L is the data type of nonempty lists and P is again the data type of pairs. The useful function $fork$ is defined by

$$\begin{aligned} fork\ \ &:: (a \to b, a \to c) \to a \to (b, c) \\ fork\, (f, g)\, x &= (f\, x, g\, x) \end{aligned}$$

The functions sg and ex are as for mergesort. The function dc returns a pair of lists of the same length, so is layered. The call graph of dc is pictured for an input list of length 5 in Figure 1. Observe that this graph is also a perfect binary tree except that subtrees are shared. A tree with shared nodes was called a *nexus* in [3] and we will henceforth adopt this terminology. Evaluating $td\, f\, g$ takes exponential time assuming f and g take constant time.

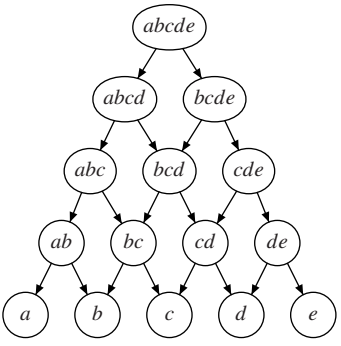

Fig. 1. A nexus

Third, consider a problem about matrices . Each $(n{+}1) \times (n{+}1)$ matrix has four $n \times n$ corner matrices. For example, the matrix

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

has four corner matrices

$$\begin{pmatrix} a & b & c \\ e & f & g \\ i & j & k \end{pmatrix} \begin{pmatrix} b & c & d \\ f & g & h \\ j & k & l \end{pmatrix} \begin{pmatrix} e & f & g \\ i & j & k \\ m & n & o \end{pmatrix} \begin{pmatrix} f & g & h \\ j & k & l \\ n & o & p \end{pmatrix}$$

Imagine a function defined on a matrix by splitting the matrix into its four corners, recursively computing the value of the function on these corners, and constructing a further value out of the four results. If the matrix is a singleton, then the value is computed directly. The type of a matrix is a list of lists, so $L\,a = [[a]]$. We can also take $F\,a = [a]$, though a special type of quadruples would serve equally well. We omit the definition of dc but it is clearly layered. The associated nexus is a perfect quaternary tree with sharing and the computation of $td\,f\,g$ takes 4^n steps on a $n \times n$ matrix, assuming f and g take constant time.

Finally, here is a problem in which the function dc is not polymorphic but can be converted into one that is. The value $comb\,(n, r)$ is the number of combinations of n objects taken r at a time. As a total function we can define

$$\begin{array}{lll} comb & :: P\ Integer \to Integer & \\ comb\,(n, r) & \mid\ n < r \vee r < 0 & = 0 \\ & \mid\ n = r \vee r = 0 & = 1 \\ & \mid\ n > r \wedge r > 0 & = comb\,(n{-}1, r) + comb\,(n{-}1, r{-}1) \end{array}$$

We set $comb\,(n, r) = 0$ if n and r do not satisfy $0 \le r \le n$. The associated decomposition, singleton test and extraction functions are not polymorphic but can be made so with a change of representation. Suppose (n, r) is represented by

a binary string of length $n+1$ containing a single 1 at position r if $0 \leq r \leq n$, and all 0s otherwise. With this representation $dc = fork\,(init, tail)$ because if xs represents (n, r), then $init\,xs$ represents $(n-1, r)$ and $tail\,xs$ represents $(n-1, r-1)$. Even better, at the expense of a little extra computation the functions sg and ex can be defined as for mergesort because a string of 0s has only singleton 0s below it, and the sum of such singletons is 0. In other words, we can base the computation of $comb$ on the nexus of Figure 1 and define

$$comb = td\,id\,(uncurry\,(+)) \cdot rep$$

where $rep :: P\,Int \to L\,Int$ installs the representation. This version is somewhat less efficient than the direct definition because of the unnecessary computations of 0, but both still take exponential time. And tabulation can reduce this to quadratic time as we will now see.

4 Bottom-Up Tabulation

Look again at Figure 1. The idea behind bottom-up tabulation is simply to replace each label x of a node n with $td\,f\,g\,x$. This is achieved by applying g to appropriate combinations of the labels of the nodes below n. The nexus takes the form of a labelled tree

$$\mathbf{data}\ N\,a = Leaf\,a \mid Node\,a\,(F\,(N\,a))$$

A node of a nexus consists of a label of type a and an F-structure of nexuses, where F is the same data type that appears in the type of dc.

The bottom-up algorithm takes the form

$$\begin{aligned} &bu \quad :: (a \to b) \to (F\,b \to b) \to L\,a \to b \\ &bu\,f\,g = label \cdot ex \cdot until\,sg\,(L\,(node\,g) \cdot cd) \cdot L\,(leaf\,f) \end{aligned}$$

where the standard function $until$ can be defined by

$$\begin{aligned} &until \quad :: (a \to Bool) \to (a \to a) \to a \to a \\ &until\,p\,f = (p \to id, until\,p\,f \cdot f) \end{aligned}$$

The computation of $bu\,f\,g$ begins with some L-structure of values. The first step is to apply $L\,(leaf\,f)$ and the result is an L-structure of nexuses, all of which are leaves; this gives the first level of the table. Then the function $L\,(node\,g) \cdot cd$ is applied repeatedly until the result is a singleton L-structure. From this singleton a single nexus is extracted and its label is returned as the final result of the computation.

Here are the types and definitions of the ingredient functions. The function $leaf$ is defined by

$$\begin{aligned} &leaf \quad :: (a \to b) \to a \to N\,b \\ &leaf\,f = Leaf \cdot f \end{aligned}$$

The companion function *node* is defined by

$$\begin{aligned} &node \quad :: (F\ b \to b) \to F\ (N\ b) \to N\ b \\ &node\ g = uncurry\ Node \cdot fork\ (g \cdot F\ label, id) \end{aligned}$$

The function *label* extracts the label of a nexus:

$$\begin{aligned} &label\ &:: N\ b \to b \\ &label\ (Leaf\ x)\ &= x \\ &label\ (Node\ (x, fns))\ &= x \end{aligned}$$

Finally, the remaining function *cd* has type $cd :: L\ a \to L\ (F\ a)$. In particular, if $g :: F\ B \to B$ for some type B, then $L\ (node\ g) \cdot cd :: L\ (N\ B) \to L\ (N\ B)$. The functions $dc :: L\ a \to F\ (L\ a)$ and $cd :: L\ a \to L\ (F\ a)$ have dual types so we give them dual names.

One obvious definition of *cd*, though by no means the only one, is suggested by its type. The definition is

$$cd = zip(F, L) \cdot dc$$

where $zip(F, L) :: F\ (L\ a) \to L\ (F\ a)$ zips an F-structure of L-structures into an L-structure of F-structures. Zips are, in general, partial operations: $zip(F, L)$ is well-defined only when applied to an F-structure of L-structures in which all the L-structures have the same shape. For example, the transpose function $zip(L, L)$ on matrices, i.e. elements of $L\ (L\ a)$ where $L\ a = [a]$, is well-defined only if the matrix is a nonempty list of nonempty lists all of the same length. The restriction to nonempty lists is required because matrix transpose on an empty matrix is essentially a nondeterministic operation, returning an arbitrary list of empty lists (in Haskell this problem is resolved by having transpose return an infinite list of empty lists). When F and L contain no empty structures, both $zip(F, L)$ and $zip(L, F)$ are partial functions. This explains our restriction to data types without empty elements. Zips are required to have additional properties, described in Section 6.

Recall that *dc* is *layered* if *dc* returns an F-structure of L-structures all of the same shape; in such a case the domain of $zip(F, L) \cdot dc$ is that of *dc*.

Theorem 1. *$bu\ f\ g = td\ f\ g$ provided that dc is layered and cd is any function satisfying the following three conditions:*

$$\begin{aligned} sg \cdot cd &= sg \cdot zip(F, L) \cdot dc \\ ex \cdot cd &= F\ ex \cdot dc \\ dc \cdot cd &= F\ cd \cdot dc \end{aligned}$$

The second condition of Theorem 1 is required to hold only if *cd* returns a result that satisfies *sg*, so $ex \cdot cd$ is well-defined. Similarly, the third condition is required to hold only if *cd* returns a result not satisfying *sg*, so $dc \cdot cd$ is well-defined.

Corollary 1. *With $cd = zip(F, L) \cdot dc$ we have $bu\ f\ g = td\ f\ g$ provided that dc is layered and also satisfies the symmetry condition*

$$F\ dc \cdot dc = zip(F, F) \cdot F\ dc \cdot dc$$

Section 6 is devoted to the proofs of Theorem 1 and Corollary 1. First we revisit the examples in Section 3 to see what the symmetry condition entails.

5 Examples Revisited

Consider again the function dc arising in mergesort over power lists; as we have seen, dc is layered. Define $cd :: L\,a \to L\,(P\,a)$ by

$$cd = zip(P, L) \cdot fork\,(even, odd)$$

where $even$ applied to a power list returns the elements in even position; similarly for odd. The function $zip(P, L)$ zips a pair of power lists into a power list of pairs. For example,

$$cd\,[0\,..\,7] = [(0,1),(2,3),(4,5),(6,7)]$$

The three conditions of Theorem 1 are easy to check and we omit details. As a result the function *bu wrap merge* defines an iterative version of mergesort. There is no asymptotic increase in time efficiency because there is no sharing of subtrees.

Next, consider the function $dc = fork\,(init, tail)$. Here we aim for an application of Corollary 1, so we take $cd = zip(P, L) \cdot dc$. To appreciate that dc is symmetric, observe that the result of applying $P\,dc \cdot dc$ to the string "abcde" is

$$((\text{“abc”}, \text{“bcd”}), (\text{“bcd”}, \text{“cde”}))$$

Now, the definition of $zip(P, P)$, the zip function for pairs of pairs, is

$$\begin{array}{ll} zip(P,P) & :: P\,(P\,a) \to P\,(P\,a) \\ zip(P,P)\,((x,y),(u,v)) & = ((x,u),(y,v)) \end{array}$$

Applying $zip(P, P)$ to the above pair of pairs leaves it unchanged. So dc is indeed symmetric and Corollary 1 is applicable. The bottom-up algorithm $bu\,f\,g$ takes quadratic time assuming f and g take constant time.

The matrix example is similar. The function dc returns a list of matrices all of the same shape, and also satisfies the symmetry condition

$$zip(L, L) \cdot L\,dc \cdot dc = L\,dc \cdot dc$$

where $zip(L, L)$ is matrix transpose. For example, applying $L\,dc \cdot dc$ to the 4×4 matrix of Section 3 gives the symmetric matrix

$$\begin{pmatrix} \begin{pmatrix} a & b \\ e & f \end{pmatrix} & \begin{pmatrix} b & c \\ f & g \end{pmatrix} & \begin{pmatrix} e & f \\ i & j \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} \\ \begin{pmatrix} b & c \\ f & g \end{pmatrix} & \begin{pmatrix} c & d \\ g & h \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} g & h \\ k & l \end{pmatrix} \\ \begin{pmatrix} e & f \\ i & j \end{pmatrix} & \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} i & j \\ m & n \end{pmatrix} & \begin{pmatrix} j & k \\ n & o \end{pmatrix} \\ \begin{pmatrix} f & g \\ j & k \end{pmatrix} & \begin{pmatrix} g & h \\ k & l \end{pmatrix} & \begin{pmatrix} j & k \\ n & o \end{pmatrix} & \begin{pmatrix} k & l \\ o & p \end{pmatrix} \end{pmatrix}$$

Indeed the term symmetric was chosen in analogy with the idea of a symmetric matrix. It can now be appreciated that the condition that dc be layered means that the associated nexus is structured into levels with connections only between one level and the next, and the symmetry condition in the case $F\ a = [a]$ means that all nodes have the same number, n say, of children, and adjacent nodes at one level are connected to adjacent n tuples of nodes at the next level. So the symmetry condition is quite severe.

6 Proofs

Let us first assume Theorem 1 and show how Corollary 1 follows. The proof of Corollary 1 depends on various properties of zips, first formulated by Backhouse and Hoogendijk and recorded in a sequence of papers [1,2,6,7]. These properties are described in a calculus of relations rather than functions because of the need to reason about partial and nondeterministic operations.

In particular, as we said earlier, $zip(F, G) :: F\,(G\ a) \to G\,(F\ a)$ is a partial operation which is only well-defined on F-structures of G-structures all of the same shape. In such a case the result is a G-structure of F-structures all of the same shape. As well as being partial, zips can also be nondeterministic. For example, take $F\ a = Int$. In this case $zip(F, G)$ takes an integer n to an arbitrary G-structure of copies of n and so is a nondeterministic operation. A similar example is matrix transpose when the empty matrix is allowed. But if we exclude empty F-structures, an assumption captured by the restriction $F\,0 = 0$ where 0 is the empty relation, then $zip(F, G)$ is a deterministic operation, though still partial.

We will need four properties of zips. The first is that $zip(F, G)$ is a natural transformation from FG to GF. The second property is that $zip(F, I)$ is the identity function on F. The third property is the law of composition:

$$zip(F, GH) = G\,(zip(F, H)) \cdot zip(F, G)$$

Finally, zips enjoy a *higher-order naturality* property. Restricted to data types F that do not contain empty elements, this property states that if $f :: G\ a \to H\ a$ is a polymorphic function, then

$$f \cdot zip(F, G) = zip(F, H) \cdot F\,f \cdot zip(F, G)\triangleright$$

Here $zip(F, G)\triangleright$ is a *coreflexive* relation, a sub-relation of the identity function, which holds only elements in the domain of $zip(F, G)$. Coreflexive relations are also known as partial skips. In particular, dc is a layered function just in the case that $dc = zip(F, L)\triangleright \cdot dc$. It is proved in [6] that zips with these properties can be defined for all the data types one normally encounters in functional programming.

Setting $cd = zip(F, L) \cdot dc$ we can now verify the three conditions of Theorem 1. The first condition is immediate from the definition of cd. The second condition,

namely $ex \cdot cd = F\ ex \cdot dc$, is proved as follows:

$$
\begin{array}{ll}
 & ex \cdot cd \\
= & \quad \{\text{given definition of } cd\} \\
 & ex \cdot zip(F, L) \cdot dc \\
= & \quad \{\text{higher-order naturality of zips and } ex :: L\ a \to I\ a\} \\
 & zip(F, I) \cdot F\ ex \cdot zip(F, L) \triangleright \cdot dc \\
= & \quad \{\text{identity property of zips and assumption that } dc \text{ is layered}\} \\
 & F\ ex \cdot dc
\end{array}
$$

Finally, to show that $dc \cdot cd = F\ cd \cdot dc$ we reason:

$$
\begin{array}{ll}
 & dc \cdot zip(F, L) \cdot dc \\
= & \quad \{\text{higher-order naturality of zips and } dc :: L\ a \to F\ (L\ a)\} \\
 & zip(F, FL) \cdot F\ dc \cdot zip(F, L) \triangleright \cdot dc \\
= & \quad \{\text{since } dc \text{ is layered}\} \\
 & zip(F, FL) \cdot F\ dc \cdot dc \\
= & \quad \{\text{composition property of zips}\} \\
 & F\ zip(F, L) \cdot zip(F, F) \cdot F\ dc \cdot dc \\
= & \quad \{\text{assumption that } dc \text{ is symmetric}\} \\
 & F\ zip(F, L) \cdot F\ dc \cdot dc \\
= & \quad \{\text{functor composition}\} \\
 & F\ (zip(F, L) \cdot dc) \cdot dc
\end{array}
$$

This completes the proof of Corollary 1.

6.1 Proof of Theorem 1

Now we turn to the proof of Theorem 1. Proofs, like computations, can be top down or bottom up. We will proceed top down, collecting claims for subsequent subproofs. A number of steps involve the two basic laws of McCarthy conditionals, namely

$$
\begin{aligned}
(p \to f, g) \cdot h &= (p \cdot h \to f \cdot h, g \cdot h) \\
h \cdot (p \to f, g) &= (p \to h \cdot f, h \cdot g)
\end{aligned}
$$

Use of these laws is signalled with the hint ‘conditionals’. Observe that by applying the first law of conditionals twice, we have that

$$(p \to f, g) \cdot h = (p \to f', g') \cdot h$$

whenever $f \cdot h = f' \cdot h$ and $g \cdot h = g' \cdot h$. Finally, frequent use is made of the functor law $F(f \cdot g) = F\ f \cdot F\ g$ without explicit mention.

The proof of Theorem 1 is by induction. We show that $bu_n = td_n$ for all n, where bu_0 and td_0 are each the empty relation (i.e., the everywhere undefined function) and

$$\begin{aligned} bu_{n+1} &= label \cdot ex \cdot un_{n+1} \cdot L\,(leaf\ f) \\ un_{n+1} &= (sg \rightarrow id, un_n \cdot L\,(node\ g) \cdot cd) \\ td_{n+1} &= (sg \rightarrow f \cdot ex, g \cdot F\ td_n \cdot dc) \end{aligned}$$

The function un abbreviates $until\ sg\,(L\,(node\ g) \cdot cd)$. The base case is immediate and the induction step is:

$$\begin{aligned} & bu_{n+1} \\ = \quad & \{\text{definition}\} \\ & label \cdot ex \cdot (sg \rightarrow id, un_n \cdot L\,(node\ g) \cdot cd) \cdot L\,(leaf\ f) \\ = \quad & \{\text{conditionals and } sg \cdot L\,(leaf\ f) = sg\} \\ & (sg \rightarrow label \cdot ex \cdot L\,(leaf\ f), label \cdot ex \cdot un_n \cdot L\,(node\ g) \cdot cd \cdot L\,(leaf\ f)) \\ = \quad & \{\text{claim: see (1) below}\} \\ & (sg \rightarrow label \cdot ex \cdot L\,(leaf\ f), label \cdot node\ g \cdot F\,(ex \cdot un_n) \cdot dc \cdot L\,(leaf\ f)) \\ = \quad & \{\text{since } label \cdot ex \cdot L\,(leaf\ f) = ex \cdot L\,f \text{ and} \\ & \qquad label \cdot node\ g = g \cdot F\ label\} \\ & (sg \rightarrow ex \cdot L\,f, g \cdot F\,(label \cdot ex \cdot un_n) \cdot dc \cdot L\,(leaf\ f)) \\ = \quad & \{\text{naturality of } ex \text{ and } dc\} \\ & (sg \rightarrow f \cdot ex, g \cdot F\,(label \cdot ex \cdot un_n \cdot L\,(leaf\ f)) \cdot dc) \\ = \quad & \{\text{definition of } bu_n\} \\ & (sg \rightarrow f \cdot ex, g \cdot F\ bu_n \cdot dc) \\ = \quad & \{\text{induction and definition of } td_{n+1}\} \\ & td_{n+1} \end{aligned}$$

The identities

$$\begin{aligned} label \cdot ex \cdot L\,(leaf\ f) &= ex \cdot L\,f \\ label \cdot node\ g &= g \cdot F\ label \end{aligned}$$

are easy to prove from the definitions of *leaf* and *node*. The claim is that

$$ex \cdot un_n \cdot L\,(node\ g) \cdot cd = node\ g \cdot F\,(ex \cdot un_n) \cdot dc \tag{1}$$

The proof of (1) is again by induction. The base case follows from the assumption that $F\,0 = 0$. For the induction step we argue:

$$\begin{aligned} & ex \cdot un_{n+1} \cdot L(node\ g) \cdot cd \\ = \quad & \{\text{definition of } un_{n+1} \text{ and conditionals}\} \\ & (sg \rightarrow ex, ex \cdot un_n \cdot L\,(node\ g) \cdot cd) \cdot L\,(node\ g) \cdot cd \\ = \quad & \{\text{induction}\} \\ & (sg \rightarrow ex, node\ g \cdot F\,(ex \cdot un_n) \cdot dc) \cdot L\,(node\ g) \cdot cd \end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{conditionals}\} \\
& (sg \cdot L\,(node\ g) \cdot cd \rightarrow ex \cdot L\,(node\ g) \cdot cd, \\
& \qquad node\ g \cdot F\,(ex \cdot un_n) \cdot dc \cdot L\,(node\ g) \cdot cd) \\
= & \{\text{naturality of } sg,\ ex \text{ and } dc\} \\
& (sg \cdot cd \rightarrow node\ g \cdot ex \cdot cd, \\
& \qquad node\ g \cdot F\,(ex \cdot un_n \cdot L\,(node\ g)) \cdot dc \cdot cd) \\
= & \{\text{the three assumptions of Theorem 1}\} \\
& (sg \cdot zip(F, L) \cdot dc \rightarrow node\ g \cdot F\ ex \cdot dc, \\
& \qquad node\ g \cdot F\,(ex \cdot un_n \cdot L\,(node\ g) \cdot cd) \cdot dc) \\
= & \{\text{conditionals}\} \\
& node\ g \cdot F\ ex \cdot (sg \cdot zip(F, L) \rightarrow id, F\,(un_n \cdot L\,(node\ g) \cdot cd)) \cdot dc \\
= & \{\text{claim: see (2) below}\} \\
& node\ g \cdot F\ ex \cdot F\,(sg \rightarrow id, un_n \cdot L\,(node\ g) \cdot cd) \cdot dc \\
= & \{\text{definition of } un_{n+1}\} \\
& node\ g \cdot F\,(ex \cdot un_{n+1}) \cdot dc
\end{array}
$$

The final claim is that for any partial functions $h, k :: L\,A \rightarrow B$

$$(sg \cdot zip(F, L) \rightarrow F\,h, F\,k) = F\,(sg \rightarrow h, k) \cdot zip(F, L)\,\triangleright \tag{2}$$

Intuitively, (2) holds because sg is polymorphic and so depends only on the shape of its argument L-structure. Thus $sg \cdot zip(F, L)$ applied to an F-structure of L-structures all of the same shape returns *True* if sg returns *True* on all the elements of F, and *False* if sg returns *False* on all the elements. However, the formal proof of (2) seems surprisingly difficult and depends on a number of additional concepts related to zips that we don't have the space to go into. So we will omit the proof.

7 Non-layered Decompositions

In practice, most dynamic programming problems do not have layered decompositions. For example, consider an optimal bracketing problem such as chain matrix multiplication (see Chapter 16 of [5]). The decomposition function dc for this problem has type $dc :: L\,a \rightarrow L\,(P\,(L\,a))$ where L is the data type of nonempty lists. For example,

$$dc\ \text{“abcd”} = [(\text{“a”}, \text{“bcd”}), (\text{“ab”}, \text{“cd”}), (\text{“abc”}, \text{“d”})]$$

We can define dc by

$$dc = zip(P, L) \cdot cross\,(inits, tails) \cdot fork\,(init, tail)$$

where *inits* and *tails* return the nonempty initial and tail segments of a list respectively, and *cross* is defined by

$$
\begin{array}{ll}
cross & :: (a \rightarrow c, b \rightarrow d) \rightarrow (a, b) \rightarrow (c, d) \\
cross\,(f, g)\,(x, y) & = (f\ x, g\ y)
\end{array}
$$

Although dc is neither layered nor symmetric, the function $fork\,(init, tail)$ has both these properties and we can base a tabulation scheme on this function instead.

More generally, suppose $dc :: L\,a \to F\,(L\,a)$ satisfies the conditions of Corollary 1, and $dcg :: L\,a \to G\,(L\,a)$ is defined by $dcg = extendL \cdot dc$, where $extendL :: F\,(L\,a) \to G\,(L\,a)$. For example, for optimal bracketing we have $G\,a = L\,(P\,a)$ and

$$extendL = zip(P, L) \cdot cross\,(inits, tails)$$

The generalised top-down algorithm gtd reads:

$$\begin{aligned} gtd\ \ &:: (a \to b) \to (G\,b \to b) \to L\,a \to b \\ gtd\,f\,g &= (sg \to f \cdot ex, g \cdot G\,(gtd\,f\,g) \cdot extendL \cdot dc) \end{aligned}$$

The generalised bottom-up tabulation algorithm gbu is defined by

$$\begin{aligned} gbu\ \ &:: (a \to b) \to (G\,b \to b) \to L\,a \to b \\ gbu\,f\,g &= label \cdot ex \cdot until\,sg\,(L\,(node\,g) \cdot cd) \cdot L\,(leaf\,f) \end{aligned}$$

where the definition of $node$ is changed to read

$$\begin{aligned} node\ \ &:: (G\,b \to b) \to F\,(N\,b) \to N\,b \\ node\,g &= uncurry\,Node \cdot fork\,(g \cdot G\,label \cdot extendN, id) \end{aligned}$$

The new function $extendN$ has type $extendN :: F\,(N\,b) \to G\,(N\,b)$. To describe the necessary relationship between $extendL$ and $extendN$ we need a function with type $L\,a \to N\,b$. The following function does the job:

$$\begin{aligned} nexus\ \ &:: (a \to b) \to (G\,b \to b) \to L\,a \to N\,b \\ nexus\,f\,g &= (sg \to leaf\,f \cdot ex, node\,g \cdot F\,(nexus\,f\,g) \cdot dc) \end{aligned}$$

Equivalently, $nexus\,f\,g = td\,(leaf\,f)\,(node\,g)$, where td is as defined in Section 2. The function $nexus\,f\,g$ builds a nexus except that subtrees are not shared.

Theorem 2. *Suppose cd and dc satisfy the conditions of Theorem 1. Then $gtd\,f\,g = gbu\,f\,g$ provided*

$$extendN \cdot F\,(nexus\,f\,g) = G\,(nexus\,f\,g) \cdot extendL$$

Proof. First observe that (1) of Section 6 involved no properties of $node\,g$ so it remains valid. Using (1) we can prove that

$$gbu\,f\,g = label \cdot nexus\,f\,g \tag{3}$$

The proof of (3) is by induction. The induction step reads:

$$\begin{aligned} & gbu_{n+1}\,f\,g \\ = \quad & \{\text{definition of } gbu\} \end{aligned}$$

$$
\begin{array}{ll}
 & label \cdot ex \cdot (sg \rightarrow id, un_n \cdot step\ g) \cdot L\,(leaf\ f) \\
= & \quad \{\text{conditionals and } sg \cdot L\,(leaf\ f) = sg\} \\
 & label \cdot (sg \rightarrow ex \cdot L\,(leaf\ f), ex \cdot un_n \cdot step\ g \cdot L\,(leaf\ f)) \\
= & \quad \{(1)\} \\
 & label \cdot (sg \rightarrow ex \cdot L\,(leaf\ f), node\ g \cdot F\,(ex \cdot un_n) \cdot dc \cdot L\,(leaf\ f)) \\
= & \quad \{\text{naturality of } ex \text{ and } dc\} \\
 & label \cdot (sg \rightarrow leaf\ f \cdot ex, node\ g \cdot F\,(ex \cdot un_n \cdot L\,(leaf\ f)) \cdot dc) \\
= & \quad \{\text{induction}\} \\
 & label \cdot (sg \rightarrow leaf\ f \cdot ex, node\ g \cdot F\,(nexus_n\ f\ g) \cdot dc) \\
= & \quad \{\text{definition of } nexus\} \\
 & label \cdot nexus_{n+1}\ f\ g
\end{array}
$$

Now to complete the proof that $gtd\ f\ g = gbu\ f\ g$ we have to show that

$$gtd\ f\ g = label \cdot nexus\ f\ g \tag{4}$$

The proof of (4) is again by induction. The induction step is

$$
\begin{array}{ll}
 & gtd_{n+1}\ f\ g \\
= & \quad \{\text{definition}\} \\
 & (sg \rightarrow f \cdot ex, g \cdot G\,(gtd_n\ f\ g) \cdot extendL \cdot dc) \\
= & \quad \{(3)\} \\
 & (sg \rightarrow f \cdot ex, g \cdot G\,(label \cdot nexus_n\ f\ g) \cdot extendL \cdot dc) \\
= & \quad \{\text{assumption}\} \\
 & (sg \rightarrow f \cdot ex, g \cdot G\ label \cdot extendN \cdot F\,(nexus_n\ f\ g) \cdot dc) \\
= & \quad \{\text{since } g \cdot G\ label \cdot extendN = label \cdot node\ g\} \\
 & (sg \rightarrow f \cdot ex, label \cdot node\ g \cdot F\,(nexus_n\ f\ g) \cdot dc) \\
= & \quad \{\text{since } f \cdot ex = label \cdot leaf\ f \cdot ex\} \\
 & (sg \rightarrow label \cdot leaf\ f \cdot ex, label \cdot node\ g \cdot F\,(nexus_n\ f\ g) \cdot dc) \\
= & \quad \{\text{conditionals and induction}\} \\
 & label \cdot nexus_{n+1}\ f\ g
\end{array}
$$

For example, consider the optimal bracketing problem again, in which $F = P$ and $G\ a = L\,(P\ a)$. Suppose we define $extendN$ by

$$
\begin{array}{l}
extendN :: P\,(N\ a) \rightarrow L\,(P\,(N\ a)) \\
extendN = zip(P, L) \cdot cross\,(lspine, rspine)
\end{array}
$$

where

$$
\begin{array}{ll}
lspine & :: N\ a \rightarrow L\,(N\ a) \\
lspine\,(Leaf\ x) & = [Leaf\ x] \\
lspine\,(Node\,(x, (\ell, r))) & = lspine\ \ell +\!\!+ [Node\,(x, (\ell, r))]
\end{array}
$$

and

$$\begin{array}{ll} rspine & :: N\ a \to L\,(N\ a) \\ rspine\ (Leaf\ x) & = [Leaf\ x] \\ rspine\ (Node\ (x, (\ell, r))) & = [Node\ (x, (\ell, r))] +\!\!+ rspine\ r \end{array}$$

It is easy to show that

$$\begin{aligned} lspine \cdot nexus\, f\ g &= L\,(nexus\, f\ g) \cdot inits \\ rspine \cdot nexus\, f\ g &= L\,(nexus\, f\ g) \cdot tails \end{aligned}$$

Now we can reason

$$\begin{array}{ll} & extendN \cdot P\,(nexus\, f\ g) \\ = & \quad \{\text{definition of } extendN\} \\ & zip(P, L) \cdot cross\,(lspine, rspine) \cdot P\,(nexus\, f\ g) \\ = & \quad \{\text{property of } cross\} \\ & zip(P, L) \cdot cross\,(lspine \cdot nexus\, f\ g, rspine \cdot nexus\, f\ g) \\ = & \quad \{\text{above}\} \\ & zip(P, L) \cdot cross\,(L\,(nexus\, f\ g) \cdot inits, L\,(nexus\, f\ g) \cdot tails) \\ = & \quad \{\text{property of } cross\} \\ & zip(P, L) \cdot PL\,(nexus\, f\ g) \cdot cross\,(inits, tails) \\ = & \quad \{\text{naturality of } zip(P, L)\} \\ & LP\,(nexus\, f\ g) \cdot zip(L, P) \cdot cross\,(inits, tails) \\ = & \quad \{\text{definition of } extendL\} \\ & LP\,(nexus\, f\ g) \cdot extendL \end{array}$$

Hence Theorem 2 is applicable.

8 Non-symmetric Decompositions

For some problems the decomposition function is layered but not symmetric. For instance, consider the function *dc* that returns the immediate subsequences of a list. For example,

$$dc\ \text{“abcde”} = [\text{“abcd”}, \text{“abce”}, \text{“abde”}, \text{“acde”}, \text{“bcde”}]$$

Applied to a list of length n the function *dc* returns n lists each of length $n-1$ obtained by dropping a single element. The nexus associated with *dc* is essentially a Boolean lattice, see Figure 2. The nexus is layered but the connectivity varies from level to level. One way of constructing a Boolean lattice was given in [3], but the construction was not shown formally to meet the requirements of the associated tabulation scheme. Instead, it is possible to justify an alternative construction by appeal to Theorem 1 and we will just sketch the details.

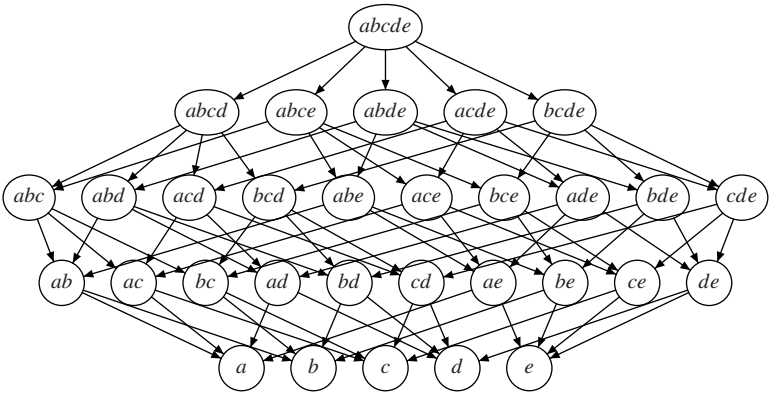

Fig. 2. A Boolean lattice

Firstly, we have $dc :: L\,a \to F\,(L\,a)$, where $F = L$ and L is the data type of nonempty lists. One definition of dc is

$$\begin{aligned} dc\,[x, y] &= [[x], [y]] \\ dc\,(x : xs) &= [[x] +\!\!+ ys \mid ys \leftarrow dc\,xs] +\!\!+ [xs] \end{aligned}$$

An equivalent definition, though not legal Haskell, is

$$\begin{aligned} dc\,[x, y] &= [[x], [y]] \\ dc\,(xs +\!\!+ [x]) &= [xs] +\!\!+ [ys +\!\!+ [x] \mid ys \leftarrow dc\,xs] \end{aligned}$$

To satisfy Theorem 1 we have to invent an appropriate definition of cd meeting the three conditions. The essential trick is to group the elements at each level in a particular way in order to prepare for the next level. For example,

$$\begin{array}{l} a \quad b \quad c \quad d \quad e \\ ab \quad (ac\ bc) \quad (ad\ bd\ cd) \quad (ae\ be\ ce\ de) \\ abc \quad (abd\,(acd\ bcd)) \quad ((abe\,(ace\ bce))\,(ade\ bde\ cde)) \\ abcd \quad (abce\,(abde\,(acde\ bcde))) \\ abcde \end{array}$$

The first level is a list of singletons, the second level consists of groups containing, in order, 1, 2, 3 and 4 members, the third level has groups of 1, 3 and 6 elements, and the fourth level has groups of 1 and 4 elements. These numbers are the diagonals of Pascal's triangle. An element ab represents the pair $[a, b]$ and abc represents the triple $[ab, ac, bc]$, and so on. The group structure can be captured by representing lists as binary trees satisfying a shape constraint, essentially that of binomial trees. For example, the middle two lines above are pictured as binomial trees in Figure 3. Reading upwards, the left spine of the first tree has subtrees of sizes 1, 2, 3 and 4, and the left spine of the second tree has subtrees

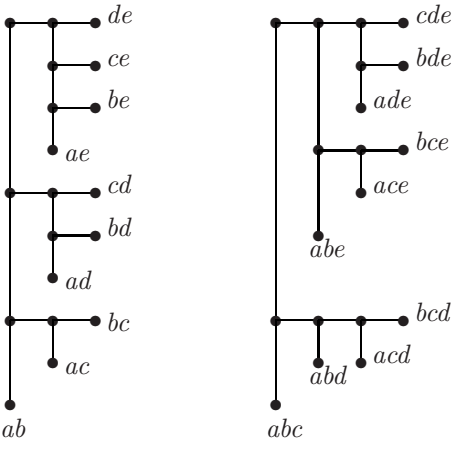

Fig. 3. Two binomial trees

of sizes 1, 3 and 6, in which the second subtree has subtrees of sizes 1, 2 and 3, and the third has subtree of sizes 1 and 2.

Given the data type declaration

$$\mathbf{data}\ B\ a = Tip\ a \mid Bin\ (B\ a)\ (B\ a)$$

the conversion function $cvtLB :: L\,a \to B\,a$ for converting a list into a binary tree is defined by

$$cvtLB = foldl1\ Bin \cdot map\ Tip$$

This function builds a binary tree all of whose right subtrees are tips, representing the first level of the nexus. With this binary tree representation of lists the function *sg* translates into a test for whether its argument is a tip, and *ex* extracts the tip value. The formal definition of *cd* is now given by

$$\begin{array}{ll}
cd & :: B\,a \to B\,(L\,a) \\
cd\,(Bin\,(Tip\,a)\,(Tip\,b)) & = Tip\,[a,b] \\
cd\,(Bin\,u\,(Tip\,b)) & = Bin\,(cd\,u)\,(B\,(:[b])\,u) \\
cd\,(Bin\,(Tip\,a)\,v) & = Tip\,(a:as) \quad \mathbf{where}\ Tip\,as = cd\,v \\
cd\,(Bin\,u\,v) & = Bin\,(cd\,u)\,(zipBWith\,(:)\,u\,(cd\,v))
\end{array}$$

The function $zipBWith :: (a \to b \to c) \to B\,a \to B\,b \to B\,c$ is analogous to the function *zipWith* on lists. For example, applying *cd* to the first tree of figure 3 yields the second tree.

We now claim that $td\,f\,g = bu\,f\,g \cdot cvtLB$. In order to justify the claim we have to invent a definition of $dc' :: B\,a \to L\,(B\,a)$ with two properties: firstly, $td\,f\,g = td'\,f\,g \cdot cvtLB$ where td' is the same as *td* except that *dc* is replaced by

dc'; and, secondly, dc' and cd satisfy the conditions of Theorem 1. The necessary definition of dc' turns out to be

$$\begin{array}{ll} dc' & :: B\ a \to L\,(B\ a) \\ dc'\ (Bin\ (Tip\ a)\ (Tip\ b)) & = [Tip\ a, Tip\ b] \\ dc'\ (Bin\ u\ (Tip\ b)) & = [u] +\!\!+ [Bin\ v\ (Tip\ b) \mid v \leftarrow dc'\ u] \\ dc'\ (Bin\ (Tip\ a)\ v) & = Tip\ a : dc'\ v \\ dc'\ (Bin\ u\ v) & = [u] +\!\!+ zipWith\ Bin\ (dc'\ u)\ (dc'\ v) \end{array}$$

The first two clauses are obvious translations of the second list-based definition of dc given above. More precisely, we have

$$dc' \cdot cvtLB = F\ cvtLB \cdot dc$$

This equation is sufficient to prove $td\ f\ g = td'\ f\ g \cdot cvtLB$. The second two clauses deal with binary trees of higher rank and are needed for the three conditions of Theorem 1. But the proofs are long and involved, so they are omitted.

9 Summary

Let us recap. Theorem 1 captures the essential relationship between top-down and bottom-up computations for layered decomposition functions. In the restricted case of a symmetric decomposition function we can appeal to Corollary 1. When the decomposition function is not layered, but can be viewed as an extension of one that is, we can appeal to Theorem 2. Finally, in order to apply Theorem 1 some invention is required, both to find an alternative data type for representing L and the appropriate decomposition and composition functions. It remains future work to see whether the invention can be placed on a more systematic footing.

Acknowledgements

A special debt of gratitude is owed to Roland Backhouse for joint collaboration on the (omitted) proof of (2), which will be recorded elsewhere. Thanks are also due to Ralf Hinze and Shin-Chen Mu for many discussions on the subject of tabulations way back in 2003 when the ideas were first being formulated. Ralf Hinze also very kindly gave of his time to draw Figures 1 and 2 using Functional MetaPost. Finally, acknowledgement is owed to all the referees for their constructive remarks, and to one in particular for suggesting that it would be interesting to see whether the combinatorial function *comb* could be treated within the framework, and my pleasure in discovering that it could.

References

1. Backhouse, R.C., Doornbos, H., Hoogendijk, P.: A Class of Commuting Relators. In: STOP workshop, Ameland, The Netherlands (September 1992), `http://www.cs.nott.ac.uk/~rcb/MPC/papers/zips.ps.gz`

2. Backhouse, R.C., Hoogendijk, P.: Generic Properties of Datatypes. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 97–132. Springer, Heidelberg (2003)
3. Bird, R.S., Hinze, R.: Trouble shared is trouble halved. In: ACM SIGPLAN Haskell Workshop, Uppsala, Sweden, pp. 1–6 (August 2003)
4. Bird, R.S., de Moor, O.: The Algebra of Programming. Prentice Hall International Series in Computer Science (1997)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Algorithms. MIT Press, Cambridge Mass (1997)
6. Hoogendijk, P.: A Generic Theory of Data Types Ph.D Thesis, Eindhoven Technical University (1997)
7. Hoogendijk, P., Backhouse, R.C.: When do datatypes commute? In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 242–260. Springer, Heidelberg (1997)
8. Steffen, P., Giegerich, R.: Table design in dynamic programming. Information and Computation 204(9) (September 2006)