Functional Pearl
Trouble Shared is Trouble Halved

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

Ralf Hinze
Institut für Informatik III
Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de

Abstract

A *nexus* is a tree that contains *shared* nodes, nodes that have more than one incoming arc. Shared nodes are created in almost every functional program—for instance, when updating a purely functional data structure—though programmers are seldom aware of this. In fact, there are only a few algorithms that exploit sharing of nodes consciously. One example is constructing a tree in sublinear time. In this pearl we discuss an intriguing application of nexuses; we show that they serve admirably as *memo structures* featuring *constant time* access to memoized function calls. Along the way we encounter Boolean lattices and binomial trees.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; E.1 [**Data**]: Data Structures—*trees*

General Terms

Algorithms, design, performance

Keywords

Memoization, purely functional data structures, sharing, Boolean lattices, binomial trees, Haskell

1 Introduction

A *nexus* is a tree that contains *shared* nodes, nodes that have more than one incoming arc. Shared nodes are created in almost every functional program, though programmers are seldom aware of this. As a simple example, consider adding an element to a binary search tree. Here is a suitable data type declaration for binary trees given in the functional programming language Haskell 98 [10]:

```
data Tree α  =  Empty
             |  Node{left :: Tree α, info :: α, right :: Tree α}
leaf         :: ∀α . α → Tree α
leaf x       =  Node Empty x Empty
```

Here is the definition of insertion:

```
insert                ::  ∀α . (Ord α) ⇒ α → Tree α → Tree α
insert x Empty        =   leaf x
insert x (Node l k r)
    | x ≤ k           =   Node (insert x l) k r       -- r is shared
    | otherwise       =   Node l k (insert x r)       -- l is shared
```

Observe that in each recursive call one subtree is copied unchanged to the output. Thus, after an insertion the updated tree *insert x t* and the original tree *t*—which happily coexist in the functional world—contain several shared nodes. As an aside, this technique is called *path copying* [11] in the data structure community.

Perhaps surprisingly, there are only a few functional programs that exploit sharing of nodes consciously [9]. For instance, sharing allows us to create a tree in sublinear time (with respect to the size of the tree). The call *full n x* creates a full or complete binary tree of depth *n* labelled with the same value *x*.

```
full            ::  ∀α . Integer → α → Tree α
full 0 x        =   leaf x
full (n+1) x    =   Node t x t       -- t is shared
   where t      =   full n x
```

The sharing is immediate: the result of the recursive call is used both for the left and for the right subtree. So is the sub-linearity: just count the nodes created!

Now, why are nexuses not more widely used? The main reason is that sharing is difficult to preserve and impossible to observe, except indirectly in the text of the program by counting the number of nodes that are created. In a purely functional setting *full* is equivalent to the following definition which exhibits linear running time:

```
full′           ::  ∀α . Integer → α → Tree α
full′ 0 x       =   leaf x
full′ (n+1) x   =   Node (full′ n x) x (full′ n x)
```

Indeed, an optimizing compiler might transform *full′* to *full* via common subexpression elimination. Since sharing is impossible to observe, it is also difficult to preserve. For instance, mapping a function across a tree, *fmap f t*, does away with all the sharing.

These observations suggest that nexuses are next to useless. This conclusion is, however, too rash. In this pearl, we show that nexuses serve admirably as *memo structures* featuring *constant time* access to memoized function calls. Since entries in a memo table are never changed—because they cache the results of a pure function—there is no need ever to update a memo table. Consequently and fortunately, maintaining sharing is a non-issue for memo tables.

REMARK 1. *Is a nexus the same as a DAG, a directed, acyclic graph? No, it is not. By definition, a nexus contains nodes with more than one incoming arc whereas a DAG may or may not have this property. By definition, a DAG may not be cyclic whereas a nexus may very well have this property (circularity being an extreme case of sharing). Finally, there is one fundamental difference between trees and graphs: a node in a tree has a* sequence *of successors, whereas a vertex in a graph has a* set *of successors.*

2 Tabulation

A *memo function* [7] is like an ordinary function except that it caches previously computed values. If it is applied a second time to a particular argument, it immediately returns the cached result, rather than recomputing it. For storing arguments and results, a memo function usually employs an indexed structure, the so-called *memo table*. The memo table can be implemented in a variety of ways using, for instance, hashing or comparison-based search tree schemes or digital search trees [3].

Memoization trades space for time, assuming that a table look-up takes (considerably) less time than recomputing the corresponding function call. This is certainly true if the function argument is an atomic value such as an integer. However, for compound values such as lists or trees the look-up time is no longer negligible. Worse, if the argument is an element of an abstract data type, say a set, it may not even be possible to create a memo table because the abstract data type does not support ordering or hashing.

To sum up, the way memoization is traditionally set up is to concentrate on the argument structure. On the other hand, the structure of the function is totally ignored, which is, of course, a good thing: once a memo table has been implemented for values of type τ, one can memoize any function whose domain happens to be τ.

In this pearl, we pursue the other extreme: we concentrate solely on the structure of the function and largely ignore the structure of the argument. We say 'largely' because the argument type often dictates the recursion structure of a function as witnessed by the extensive literature on *foomorphisms* [6, 1].

The central idea is to capture the call graph of a function as a nexus, with shared nodes corresponding to repeated recursive calls with identical arguments. Of course, building the call graph puts a considerable burden on the programmer but as a reward we achieve tabulation for free: each recursive call is only a link away.

To illustrate the underlying idea let us tackle the standard example, the Fibonacci function:

$$\begin{array}{lcl} \mathit{fib} & :: & \mathit{Integer} \to \mathit{Integer} \\ \mathit{fib}\ 0 & = & 0 \\ \mathit{fib}\ 1 & = & 1 \\ \mathit{fib}\ (n+2) & = & \mathit{fib}\ (n) + \mathit{fib}\ (n+1) \end{array}$$

The naive implementation entails an exponential number of recursive calls; but clearly there are only a linear number of different calls. Thus, the call graph is essentially a linear list—the elements corresponding to $\mathit{fib}\ (n)$, $\mathit{fib}\ (n-1)$, ..., $\mathit{fib}\ (1)$, $\mathit{fib}\ (0)$—with additional links to the tail of the tail, that is, from $\mathit{fib}\ (n+2)$ to $\mathit{fib}\ (n)$. To implement a memoized version of *fib* we reuse the tree type of Sec. 1: the left subtree is the link to the tail and the right subtree serves as the additional link to the tail of the tail.

$$\begin{array}{lcl} \mathit{memo\text{-}fib} & :: & \mathit{Integer} \to \mathit{Tree\ Integer} \\ \mathit{memo\text{-}fib}\ 0 & = & \mathit{leaf}\ 0 \\ \mathit{memo\text{-}fib}\ 1 & = & \mathit{Node}\ (\mathit{leaf}\ 0)\ 1\ \mathit{Empty} \\ \mathit{memo\text{-}fib}\ (n+2) & = & \mathit{node}\ t\ (\mathit{left}\ t) \\ \quad \mathbf{where}\ t & = & \mathit{memo\text{-}fib}\ (n+1) \end{array}$$

The function *node* is a smart constructor that combines the results of the two recursive calls:

$$\begin{array}{lcl} \mathit{node} & :: & \mathit{Tree\ Integer} \to \mathit{Tree\ Integer} \to \mathit{Tree\ Integer} \\ \mathit{node}\ l\ r & = & \mathit{Node}\ l\ (\mathit{info}\ l + \mathit{info}\ r)\ r \end{array}$$

We will use smart constructors heavily in what follows as they allow us to separate the construction of the graph from the computation of the function values.

Now, the *fib* function can be redefined as follows:

$$\mathit{fib} = \mathit{info} \cdot \mathit{memo\text{-}fib}$$

Note, however, that in this setup only the recursive calls are memoized. If *fib* is called repeatedly, then the call graph is built repeatedly, as well. Indeed, this behaviour is typical of *dynamic-programming* algorithms [2], see below.

In the rest of the paper we investigate two families of functions operating on sequences that give rise to particularly interesting call graphs.

3 Segments

A *segment* is a non-empty, contiguous part of a sequence. For instance, the sequence *abcd* has 10 segments: *a*, *b*, *c*, *d*, *ab*, *bc*, *cd*, *abc*, *bcd*, and *abcd*. An *immediate segment* results from removing either the first or the last element of a sequence. In general, a sequence of length n has two immediate segments (for $n \geq 2$ and zero immediate segments for $0 \leq n \leq 1$) and $\frac{1}{2}n(n+1)$ segments in total.

A standard example of the use of segments is the problem of *optimal bracketing* in which one seeks to bracket an expression $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ in the best possible way. It is assumed that '$\oplus$' is an associative operation, so the way in which the brackets are inserted does not affect the value of the expression. However, bracketing may affect the costs of computing the value. One instance of this problem is *chain matrix multiplication*.

The following recursive formulation of the problem makes use of a binary tree to represent each possible bracketing:

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Expr}\ \alpha & = & \mathit{Const}\ \alpha \mid \mathit{Expr}\ \alpha :\!\oplus\!: \mathit{Expr}\ \alpha \\ \mathit{opt} & :: & [\sigma] \to \mathit{Expr}\ \sigma \\ \mathit{opt}\ [x] & = & \mathit{Const}\ x \\ \mathit{opt}\ \mathit{xs} & = & \mathit{best}\ [\mathit{opt}\ s_1 :\!\oplus\!: \mathit{opt}\ s_2 \mid (s_1, s_2) \leftarrow \mathit{uncat}\ \mathit{xs}] \end{array}$$

The function $\mathit{best} :: [\mathit{Expr}\ \sigma] \to \mathit{Expr}\ \sigma$ returns the best tree (its definition depends on the particular problem at hand), and *uncat* splits a sequence that contains at least two elements in all possible ways:

$$\begin{array}{lcl} \mathit{uncat} & :: & \forall \alpha . [\alpha] \to [([\alpha],[\alpha])] \\ \mathit{uncat}\ [x_1, x_2] & = & [([x_1],[x_2])] \\ \mathit{uncat}\ (x : \mathit{xs}) & = & ([x], \mathit{xs}) : \mathit{map}\ (\lambda(l,r) \to (x : l, r))\ (\mathit{uncat}\ \mathit{xs}) \end{array}$$

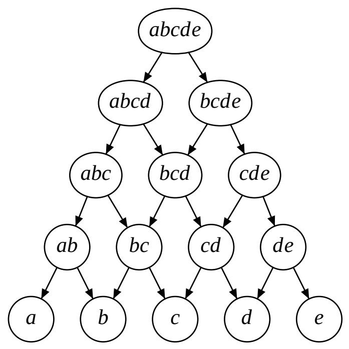

Figure 1. Call graph of a function that recurs on the immediate segments.

The recursive formulation leads to an exponential time algorithm, and the standard dynamic programming solution is to make use of a memo table to avoid computing *opt* more than once on the same argument. One purely functional scheme, a rather clumsy one, is developed on pages 233 to 236 of [1]. However, using nexuses, there is a much simpler solution.

Before we tackle optimal bracketing, let us first look at a related but simpler problem, in which each recursive call depends only on the immediate segments.

3.1 Immediate segments

Consider the function f defined by the following scheme:

$$\begin{array}{lcl} f & :: & [\sigma] \to \tau \\ f\,[x] & = & \varphi\,x \\ f\,xs \mid \mathit{length}\,xs \geq 2 & = & f\,(\mathit{init}\,xs) \diamond f\,(\mathit{tail}\,xs) \end{array}$$

where $\varphi :: \sigma \to \tau$ and $(\diamond) :: \tau \to \tau \to \tau$. Note that *init xs* and *tail xs* are the immediate segments of *xs*. Furthermore, note that f is defined only for non-empty sequences. The recursion tree or call graph of f for the initial argument *abcde* is depicted in Fig 1. The call graph has the form of a triangle; the inner nodes of the triangle are shared since $\mathit{init} \cdot \mathit{tail} = \mathit{tail} \cdot \mathit{init}$.

Now, let us build the recursion tree explicitly. We reuse the *Tree* data type of Sec. 1 and redefine the smart constructors *leaf* and *node*, which now take care of calling φ and '$\diamond$'.

$$\begin{array}{lcl} \mathit{leaf} & :: & \sigma \to \mathit{Tree}\,\tau \\ \mathit{leaf}\,x & = & \mathit{Node}\,\mathit{Empty}\,(\varphi\,x)\,\mathit{Empty} \\ \mathit{node} & :: & \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \\ \mathit{node}\,l\,r & = & \mathit{Node}\,l\,(\mathit{info}\,l \diamond \mathit{info}\,r)\,r \end{array}$$

The most immediate approach is to build the call tree in a bottom-up, iterative manner: starting with a list of singleton trees we repeatedly join adjacent nodes until one tree remains.

$$\begin{array}{lcl} \mathit{bottom\text{-}up} & :: & [\sigma] \to \mathit{Tree}\,\tau \\ \mathit{bottom\text{-}up} & = & \mathit{build} \cdot \mathit{map}\,\mathit{leaf} \\ \mathit{build} & :: & [\mathit{Tree}\,\tau] \to \mathit{Tree}\,\tau \\ \mathit{build}\,[t] & = & t \\ \mathit{build}\,ts & = & \mathit{build}\,(\mathit{step}\,ts) \\ \mathit{step} & :: & [\mathit{Tree}\,\tau] \to [\mathit{Tree}\,\tau] \\ \mathit{step}\,[t] & = & [\,] \\ \mathit{step}\,(t_1 : t_2 : ts) & = & \mathit{node}\,t_1\,t_2 : \mathit{step}\,(t_2 : ts) \end{array}$$

The last equation introduces sharing: t_2 is used two times on the right-hand side.

Alternatively, the tree can be constructed in a top-down, recursive fashion: the triangle is created by adding a diagonal slice (corresponding to a left spine) for each element of the sequence.

$$\begin{array}{lcl} \mathit{top\text{-}down} & :: & [\sigma] \to \mathit{Tree}\,\tau \\ \mathit{top\text{-}down} & = & \mathit{foldr1}\,(\lhd) \cdot \mathit{map}\,\mathit{leaf} \end{array}$$

The helper function '$\lhd$' adds one slice to a nexus: its first argument is the singleton tree to be placed at the bottom, its second argument is the nexus itself. For instance, when called with *leaf a* and the tree rooted at *bcde* (see Fig. 1), '$\lhd$' creates the nodes labelled with *abcde*, *abcd*, *abc*, *ab* and finally places *leaf a* at the bottom.

$$\begin{array}{lcl} (\lhd) & :: & \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \\ t \lhd u@(\mathit{Empty}) & = & t \\ t \lhd u@(\mathit{Node}\,l\,x\,r) & = & \mathit{node}\,(t \lhd l)\,u \end{array}$$

Of course, since the smart constructor *node* accesses only the roots of the immediate subtrees, it is not necessary to construct the tree at all. We could simply define $\mathit{leaf} = \varphi$ and $\mathit{node} = (\diamond)$. (In fact, this is only true of the bottom-up version. The top-down version must keep the entire left spine of the tree.) The tree structure comes in handy if we want to access arbitrary subtrees, as we need to do for solving the optimal bracketing problem. This is what we turn our attention to now.

3.2 All segments

The function *opt* is an instance of the following recursion scheme:

$$\begin{array}{lcl} f & :: & [\sigma] \to \tau \\ f\,[x] & = & \varphi\,x \\ f\,xs \mid \mathit{length}\,xs \geq 2 & = & \varsigma\,[(f\,s_1, f\,s_2) \mid (s_1, s_2) \leftarrow \mathit{uncat}\,xs] \end{array}$$

where $\varphi :: \sigma \to \tau$ and $\varsigma :: [(\tau, \tau)] \to \tau$. The function ς combines the solutions for the 'uncats' of *xs* to a solution for *xs*.

Since the call tree constructed in the previous section contains all the necessary information we only have to adapt the smart constructor *node*:

$$\begin{array}{lcl} \mathit{node} & :: & \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \to \mathit{Tree}\,\tau \\ \mathit{node}\,l\,r & = & \mathit{Node}\,l\,(\varsigma\,(\mathit{zip}\,(\mathit{lspine}\,l)\,(\mathit{rspine}\,r)))\,r \end{array}$$

The 'uncats' of the sequence are located on the left and on the right spine of the corresponding node.

$$\begin{array}{lcl} \mathit{lspine}, \mathit{rspine} & :: & \forall \alpha\,.\,\mathit{Tree}\,\alpha \to [\alpha] \\ \mathit{lspine}\,(\mathit{Empty}) & = & [\,] \\ \mathit{lspine}\,(\mathit{Node}\,l\,x\,r) & = & \mathit{lspine}\,l \mathbin{+\!\!+} [x] \\ \mathit{rspine}\,(\mathit{Empty}) & = & [\,] \\ \mathit{rspine}\,(\mathit{Node}\,l\,x\,r) & = & [x] \mathbin{+\!\!+} \mathit{rspine}\,r \end{array}$$

The functions *lspine* and *rspine* can be seen as 'partial' inorder traversals: *lspine* ignores the right subtrees while *rspine* ignores the left subtrees. (The function *lspine* exhibits quadratic running time, but this can be remedied using standard techniques.) For instance, the left spine of the tree rooted at *abcd* is *a*, *ab*, *abc*, and *abcd*. Likewise, the right spine of the tree rooted at *bcde* is *bcde*, *cde*, *de*, and *e*. To obtain the uncats of *abcde*, we merely have to zip the two sequences.

Now, to solve the optimal bracketing problem we only have to define $\varphi = \mathit{Const}$ and $\varsigma = \mathit{best} \cdot \mathit{map}\,(\mathit{uncurry}\,(:\oplus:))$.

4 Subsequences

A *subsequence* is a possibly empty, possibly non-contiguous part of a sequence. For instance, the sequence *abcd* has 16 subsequences: ε, *a*, *b*, *c*, *d*, *ab*, *ac*, *ad*, *bc*, *bd*, *cd*, *abc*, *abd*, *acd*, *bcd*, and *abcd*. An *immediate subsequence* results when just one element is removed from the sequence. A sequence of length n has n immediate subsequences and 2^n subsequences in total.

As an illustration of the use of subsequences we have Hutton's Countdown problem [4]. Briefly, one is given a bag of source numbers and a target number, and the aim is to generate an arithmetic expression from some of the source numbers whose value is as close to the target as possible. The problem can be solved in a variety of ways, see [8]. One straightforward approach is to set it up as an instance of *generate and test* (we are only interested in the first phase here). We represent bags as ordered sequences employing the fact that a subsequence of an ordered sequence is again ordered. The generation phase itself can be separated into two steps: first generate all subsequences, then for each subsequence generate all arithmetic expressions that contain the elements *exactly* once.

```
data Expr     =  Const Integer
              |  Add Expr Expr | Sub Expr Expr
              |  Mul Expr Expr | Div Expr Expr
exprs         :: [Integer] → [Expr]
exprs         =  concatMap generate · subsequences
generate      :: [Integer] → [Expr]
generate [x]  =  [Const x]
generate xs   =  [e | (s₁,s₂) ← unmerge xs,
                      e₁ ← generate s₁,
                      e₂ ← generate s₂,
                      e ← combine e₁ e₂]
```

The function *combine*, whose code is omitted, yields a list of all possible ways to form an arithmetic expression out of two subexpressions. The function *unmerge* splits a sequence that contains at least two elements into two subsequences (whose merge yields the original sequence) in all possible ways.

```
unmerge          :: ∀α . [α] → [([α],[α])]
unmerge [x₁,x₂]  =  [([x₁],[x₂])]
unmerge (x:xs)   =  ([x],xs) : map (λ(l,r) → (l,x:r)) s
                              ++ map (λ(l,r) → (x:l,r)) s
    where s      =  unmerge xs
```

For instance, *unmerge abcd* yields the following list of pairs: $[(a,bcd),(b,acd),(c,abd),(bc,ad),(ab,cd),(ac,bd),(abc,d)]$.

Now, how can we weave a nexus that captures the call graph of *generate*? As before, we first consider a simpler problem, in which each recursive call depends only on the immediate subsequences.

4.1 Immediate subsequences

Consider the function f defined by the following scheme:

```
f     :: [σ] → τ
f []  =  ω
f [x] =  φ x
f xs  =  ς [f s | s ← delete xs]
```

where $\omega :: \tau$, $\varphi :: \sigma \to \tau$, and $\varsigma :: [\tau] \to \tau$. The function ς combines the solutions for the immediate subsequences of *xs* to a solution for

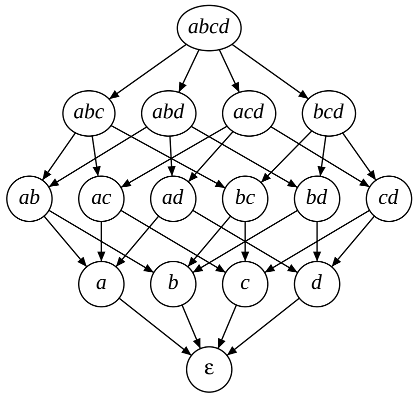

Figure 2. Call graph of a function that recurs on the immediate subsequences.

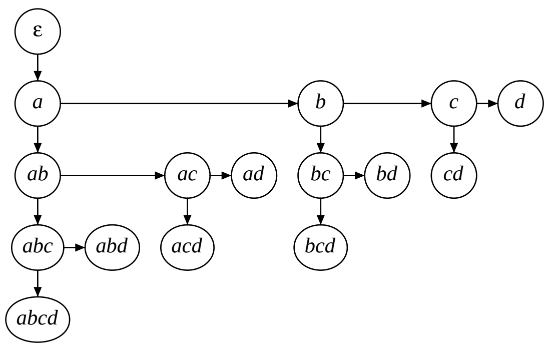

Figure 3. The binomial tree corresponding to the inverse of the lattice of Fig. 2.

xs. The function *delete* yields the immediate subsequences.

```
delete         :: ∀α . [α] → [[α]]
delete []      =  []
delete (x:xs)  =  map (x:) (delete xs) ++ [xs]
```

The call graph of f for the initial argument *abcd* is depicted in Fig. 2. Clearly, it has the structure of a *Boolean lattice*. Though a Boolean lattice has a very regular structure it is not immediately clear how to create a corresponding nexus. So let us start with the more modest aim of constructing its *spanning tree*. Interestingly, *a* spanning tree of a Boolean lattice is a *binomial tree*. Recall that a binomial tree is a multiway tree defined inductively as follows: a binomial tree of rank n has n children of rank $n-1, n-2, \ldots, 0$. To represent a multiway tree we use the left child, right sibling representation.[1] Since it will slightly simplify the presentation, we will build the binomial tree upside down, so the subtrees are labelled with 'supersequences' rather than subsequences.

```
top-down, tree  :: ∀α . [α] → Tree [α]
top-down xs     =  Node (tree xs) [] Empty
tree []         =  Empty
tree (x:xs)     =  Node l [x] r
   where l      =  fmap (x:) (tree xs)     -- child
         r      =  tree xs     -- sibling
```

The binomial tree for the sequence *abcd* is pictured in Fig. 3. Note

[1] Actually, a binary tree represents a *forest* of multiway trees, see [5]. A single multiway tree is represented by a binary tree with an empty right subtree.

that the left child, right sibling representation of a binomial tree is a perfectly balanced binary tree (if we chop off the root node). Now, it is immediate that each node in the left subtree l has an immediate subsequence in the right subtree r at the corresponding position. This observation is the key for extending each node by additional *up-links* to the immediate subsequences. In order to do so we have to extend the data type *Tree* first.

```
data Tree α  =  Empty
             |  Node{ up :: [Tree α],        -- up links
                      left :: Tree α,        -- child
                      info :: α,
                      right :: Tree α }      -- sibling
```

As usual, we introduce a smart constructor that takes care of calling φ and ς.

```
node            ::  [Tree τ] → Tree τ → σ → Tree τ → Tree τ
node [u] l x r  =   Node [u] l (φ x) r
node us l x r   =   Node us l (ς (map info us)) r
```

Here is the revised version of *top-down* that creates the augmented binomial tree.

```
top-down     ::  [σ] → Tree τ
top-down xs  =   v
  where v    =   Node [ ] (tree xs v [ ]) ω Empty
```

The helper function *tree* takes a sequence, a pointer to the father and a list of pointers to predecessors, that is, to the immediate subsequences excluding the father. To understand the code, recall our observation above: each node in the left subtree l has an immediate subsequence in the right subtree r at the corresponding position.

```
tree              ::  [σ] → Tree τ → [Tree τ] → Tree τ
tree [ ] p ps     =   Empty
tree (x : xs) p ps =  v
  where v         =   node (p : ps) l x r
        l         =   tree xs v (r : map left ps)      -- child
        r         =   tree xs p (map right ps)         -- sibling
```

The parent of the child l is the newly created node v; since we go down, l's predecessors are the right subtree r and the left subtrees of v's predecessors. The parent of the sibling r is u, as well; its predecessors are the right subtrees of v's predecessors. The subtree l has one predecessor more than r, because the sequences in l are one element longer than the sequences in r (at corresponding positions).

Do you see where all a node's immediate subsequences are? Pick a node in Fig. 3, say *abd*. Its parent (in the multiway tree view) is an immediate subsequence, in our example *ab*. Furthermore, the node at the corresponding position in the right subtree of the parent is an immediate subsequence, namely *ad*. The next immediate subsequence, *bd*, is located in the right subtree of the grandparent and so forth.

To sum up, *top-down xs* creates a *circular nexus*, with links going up and going down. The down links constitute the binomial tree structure (using a binary tree representation) and the up links constitute the Boolean lattice structure (using a multiway tree representation). Since the nexus has a circular structure, *tree* depends on *lazy evaluation* (whereas the previous programs happily work in a strict setting).

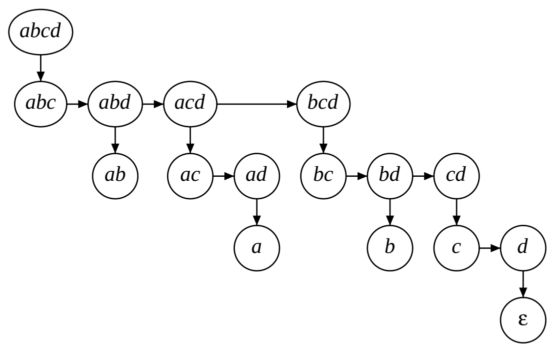

Figure 4. The binomial tree corresponding to the lattice of Fig. 2.

4.2 All subsequences

The function *generate* is an instance of the following scheme:

```
f                     ::  [σ] → τ
f [ ]                 =   ω
f [x]                 =   φ x
f xs | length xs ≥ 2  =   ς [(f s₁, f s₂) | (s₁, s₂) ← unmerge xs]
```

where $\omega :: \tau$, $\varphi :: \sigma \to \tau$, and $\varsigma :: [(\tau,\tau)] \to \tau$. The function ς combines the solutions for the unmerges of *xs* to a solution for *xs*.

Each node in the nexus created by *top-down* spans a sublattice of sequences. Since each recursive call of *generate* depends on all subsequences of the argument, we have to access every element of this sublattice. In principle, this can be done by a *breadth-first traversal* of the graph structure. However, for a graph traversal we have to keep track of visited nodes. Alas, this is not possible with the current setup since we cannot check two nodes for equality. Fortunately, there is an attractive alternative at hand: we first calculate a spanning tree of the sublattice and then do a *level-order traversal* of the spanning tree.

As we already know, one spanning tree of a Boolean lattice is the binomial tree. Since we follow the up links, we use again the multiway tree representation.

```
data Rose a  =  Branch{ label :: a, subtrees :: [Rose a] }
binom        ::  ∀α . Int → Tree α → Rose α
binom r (Node us _ x _)
             =   Branch x [binom i u | (i, u) ← zip [0 . . r − 1] us]
```

Given a rank r, the call *binom r t* yields the binomial tree of the nexus t. Note that the ranks of the children are increasing from left to right (whereas normally they are arranged in decreasing order of rank). This is because we are working on the upside-down lattice with the largest sequence on top, see Fig. 4.

```
level-order     ::  ∀α . [Rose α] → [α]
level-order ts
  | null ts     =   [ ]
  | otherwise   =   map label ts
                    ++ level-order (concatMap subtrees ts)
```

As an example, the level-order traversal of the binomial tree shown in Fig. 4 is *abcd*, *abc*, *abd*, *acd*, *bcd*, *ab*, *ac*, *ad*, *bc*, *bd*, *cd*, *a*, *b*, *c*, *d*, and ε. Clearly, to obtain all possible unmerges we just have to zip this list with its reverse.

The level-order traversal has to be done for each node of the nexus.

As before, it suffices to adapt the smart constructor *node* accordingly:

$$\begin{array}{lcl}
\mathit{node} & :: & [\mathit{Tree}\ \tau] \to \mathit{Tree}\ \tau \to \sigma \to \mathit{Tree}\ \tau \to \mathit{Tree}\ \tau \\
\mathit{node}\ [u]\ l\ x\ r & = & \mathit{Node}\ [u]\ l\ (\varphi\ x)\ r \\
\mathit{node}\ \mathit{us}\ l\ x\ r & = & t \\
\quad \mathbf{where} & & \\
\quad t & = & \mathit{Node}\ \mathit{us}\ l\ (\varsigma\ (\mathit{tail}\ (\mathit{zip}\ (\mathit{reverse}\ \mathit{xs}_1)\ \mathit{xs}_2)))\ r \\
\quad (\mathit{xs}_1, \mathit{xs}_2) & = & \mathit{halve}\ (\mathit{level\text{-}order}\ [\mathit{binom}\ (\mathit{length}\ \mathit{us})\ t])
\end{array}$$

Note that we have to remove the trivial unmerge $(\varepsilon, \mathit{abcd})$ from the list of zips in order to avoid a black hole (using *tail*). The function *halve* splits a list into two segments of equal length.

Now, to solve the Countdown problem we first have to define suitable versions of ω, φ, and ς—we leave the details to the reader. Since the nexus woven by *top-down* already contains the solutions for *all* subsequences, collecting the arithmetic expression trees is simply a matter of flattening a binary tree. Here is the re-implementation of *exprs*:

$$\begin{array}{lcl}
\mathit{exprs}\ \mathit{xs} & = & \mathit{concat}\ (\mathit{flatten}\ [\mathit{top\text{-}down}\ \mathit{xs}]) \\
\mathit{flatten} & :: & \forall\alpha\,.\,[\mathit{Tree}\ \alpha] \to [\alpha] \\
\mathit{flatten}\ [\,] & = & [\,] \\
\mathit{flatten}\ (\mathit{Empty} : \mathit{ts}) & = & \mathit{flatten}\ \mathit{ts} \\
\mathit{flatten}\ (\mathit{Node}\ \_\ l\ x\ r : \mathit{ts}) & = & x : \mathit{flatten}\ ([r] + \!\!+\ \mathit{ts} + \!\!+\ [l])
\end{array}$$

All in all, an intriguing solution for a challenging problem.

5 Conclusion

The use of nexus programming to create the call graph of a recursive program seems to be an attractive alternative to using an indexed memo table. As we said above, the result of a recursive subcomputation is then only a link away. But the programming is more difficult and we have worked out the details only for two examples: segments and subsequences. The method is clearly in need of generalisation, both to other substructures of sequences, but also beyond sequences to an arbitrary data type. What the examples have in common is the idea of recursing on a function *ipreds* that returns the immediate predecessors in the lattice of substructures of interest. For segments the value of *ipreds xs* is the pair $(\mathit{init}\ \mathit{xs}, \mathit{tail}\ \mathit{xs})$. For subsequences, *ipreds* is the function *delete* that returns immediate subsequences. It is the function *ipreds* that determines the shape of the call graph. To avoid having the programmer embark on a voyage of discovery for each new problem, we need a general theorem that shows how to build the call tree with sharing given only knowledge about *ipreds*. Whether or not the sharing version of the call tree should be built bottom-up by iterating some process *step*, or recursively by a fold, remains open. We have some preliminary ideas about what such a general theorem should look like (the conditions of the theorem relate *ipreds* and *step*), but they are in need of polishing. What is clear, however, is that the exploitation of sharing is yet another technique available to functional programmers interested in optimising their programs.

Acknowledgements

We would like to thank four anonymous referees for their constructive and helpful comments.

6 References

[1] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall Europe, London, 1997.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, second edition, 2001.

[3] Ralf Hinze. Memo functions, polytypically! In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 17–32, July 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[4] Graham Hutton. Functional Pearl: the countdown problem. *Journal of Functional Programming*, 12(6):609–616, November 2002.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Publishing Company, 3rd edition, 1997.

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

[7] Donald Michie. "Memo" functions and machine learning. *Nature*, (218):19–22, April 1968.

[8] Shin-Cheng Mu. *A calculational approach to program inversion*. PhD thesis, Oxford University Computing Laboratory, 2003.

[9] Chris Okasaki. Functional Pearl: Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6), November 1997.

[10] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[11] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.