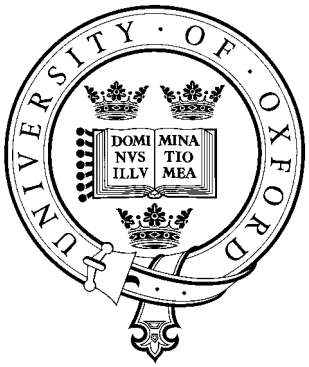
Loopless Functional Algorithms

Jamie Snape
Worcester College, University of Oxford



Thesis submitted for the degree of Master of Science
at the University of Oxford

Make me to see 't; or, at the least, so prove it,
That the probation bear no hinge nor loop
To hang a doubt on; or woe upon thy life!

William Shakespeare, *Othello: The Moor of Venice, Act III, Scene iii, 1604*

Contents

List of Figures

Part I

Background

Chapter 1

Combinatorics and Looplessness

> *In the labyrinth of a difficult text, we find unmarked forks in the path, detours, blind alleys, loops that deliver us back to our point of entry, and finally the monster who whispers an unintelligible truth in our ears.*
>
> Mason Cooley, *City Aphorisms, Fifth Selection, 1988*

Combinatorics is the study of finite sets of objects defined by certain properties. For example,

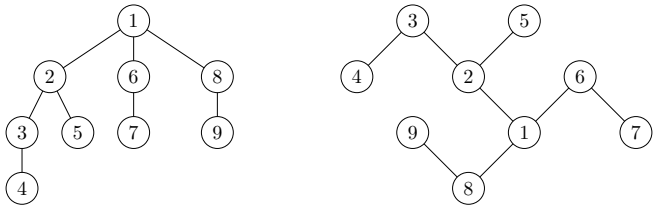
(i) Subsets of a finite set

$$\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}.$$

(ii) Permutations of n objects

$$123, 132, 213, 231, 312, 321.$$

(iii) Graphs and digraphs



(iv) Combinations of t from n objects

$$321, 421, 431, 432, 521, 531, 532, 541, 542, 543.$$

There are several related questions to consider for any particular set of combinatorial objects, such as existence and enumeration, but we will focus on generation, that is, generating all possibilities and visiting each object in turn.

Algorithms for the combinatorial generation of a set of objects vary according to many constraints. We are concerned with the efficiency of the algorithm in terms of time, and the order in which objects are visited. The timing of combinatorial algorithms will be studied in the context of loopless algorithms, while we will see several different methods of ordering for each problem, both loopless and non-loopless.

1.1. Loopless imperative algorithms

An imperative algorithm for generating combinatorial objects is 'loopless' if for every set of n elements: [1]

(i) The number of steps needed to generate the first object is less than $O(n)$.

(ii) The decision whether an object is the last is obtained within $O(1)$ steps.

(iii) Every transition between successive objects requires at most $O(1)$ steps.

(iv) The objects are represented in a simple form and can be read directly without requiring any additional steps.

To remove the loops from combinatorial algorithms we often need to use techniques such as focus pointers, doubly linked lists and coroutines. The overhead generated from this means that the total running time of a loopless algorithm on a sequential computer may not be less than that of a straightforward amortized $O(1)$ algorithm.

1.2. Folds and unfolds

To formulate a functional interpretation of a loopless algorithm, we clearly need to construct a list from a particular seed value. The obvious way to achieve this is by using the library function *unfoldr*:

$$\begin{array}{lcl} \mathit{unfoldr} & :: & (b \to \mathit{Maybe}\ (a, b)) \to b \to [a] \\ \mathit{unfoldr}\ \mathit{step}\ y & = & \textbf{case}\ \mathit{step}\ y\ \textbf{of} \\ & & \quad \mathit{Just}\ (x, y') \to x : \mathit{unfoldr}\ \mathit{step}\ y' \\ & & \quad \mathit{Nothing} \to [\,] \end{array}$$

where the standard type *Maybe* is defined by

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

Intuitively, *unfoldr* is given an initial state y and the *step* function is applied to it to determine whether an element of the output list is produced. If the value of *step* y is *Nothing*, then we have reached the end of the list. However, when *step* y is *Just* (x, y'), we cons a new element x to the start of the list and use y' in the generation of the remainder of this list.

The name *unfoldr* is appropriate since we can consider it to be the opposite, or dual, to the prelude function *foldr*:

$$\begin{array}{lcl} \mathit{foldr} & :: & (a \to b \to b) \to b \to [a] \to b \\ \mathit{foldr}\ f\ e\ [\,] & = & e \\ \mathit{foldr}\ f\ e\ (x : xs) & = & f\ x\ (\mathit{foldr}\ f\ e\ xs) \end{array}$$

While *foldr* constructs lists, *unfoldr* consumes them. In fact, we have the following result:

THEOREM 1.1. If $\mathit{step}\ (f\ x\ y) = \mathit{Just}\ (x, y)$ and $\mathit{step}\ z = \mathit{Nothing}$, then [2]

$$\mathit{unfoldr}\ \mathit{step} \cdot \mathit{foldr}\ f\ z = \mathit{id}$$

[1] Ehrlich (1973b) 500–501.

[2] Peyton Jones (2003) 183–184.

1.3. Loopless functional algorithms

Our version of a loopless functional algorithm is expressed in the form [3]

$$unfoldr\ step \cdot prolog$$

where *step* and *prolog* take $O(1)$ and $O(n)$ time, respectively, in the size of the input.

As a result of laziness, the work of *prolog* is distributed throughout the computation of *unfoldr step*. Ideally, for true looplessness, we would like *prolog* to be evaluated entirely before any evaluation of *unfoldr step*, but unfortunately we are not able to define a completely strict composition operator. Despite this, we do still catch the essential idea of a loopless algorithm.

It is important to note that since it may not be possible to generate a particular combinatorial object in $O(1)$ time, we are concerned with generating the transitions between objects instead of the objects themselves. These, for example, may be expressed as a list of integers representing the position in a binary n-tuple of a bit to be complemented, or a characterization of an interchange of two elements in a string. Identifying a suitable description for transitions may seem an added inconvenience, but in reality this is not a problem for most combinatorial algorithms.

However, we begin with a loopless version of the prelude function *concat* defined by

$$\begin{array}{lcl} concat & :: & [[a]] \to [a] \\ concat & = & foldr\ (+\!\!+)\ [\,] \end{array}$$

This can be defined in terms of *unfoldr*: [4]

$$\begin{array}{lcl} concat' & = & unfoldr\ step \cdot filter\ (\neg \cdot null) \\ & & \textbf{where}\ step\ [\,] = Nothing \\ & & \quad\ step\ ((x : xs) : xss) = Just\ (x, consList\ xs\ xss) \end{array}$$

The function *consList* is defined by

$$\begin{array}{lcl} consList\ xs\ xss & = & \textbf{if}\ null\ xs\ \textbf{then}\ xss \\ & & \textbf{else}\ xs : xss \end{array}$$

Empty lists are filtered from the input to ensure that *step* takes $O(1)$ time. This is because if we had a run of m empty lists between two non-empty lists, then after producing the last element of the first list, it would take m steps to the produce the first element of the last list. Our definition of *consList* ensures that we continue to cons only non-empty lists onto a list of lists.

---

[3] Bird (2005b). We will use the syntax and libraries of the lazy functional language Haskell 98 throughout.

[4] We could also define *concat′* by

$$\begin{array}{lcl} concat' & = & unfoldr\ step \cdot concat \\ & & \textbf{where}\ step\ [\,] = Nothing \\ & & \quad\ step\ (x : xs) = Just\ (x, xs) \end{array}$$

but this would not be particularly constructive since *unfoldr step* is the identity function on lists, with the real computation being carried out by the prelude function *concat*. Moreover, this idea would not work for functions whose output is exponentially longer than the input.

Chapter 2

Tree Traversals

> *What went forth to the ends of the world to traverse not itself, God, the sun, Shakespeare, a commercial traveller, having itself traversed in reality itself becomes that self.*
>
> JAMES JOYCE, *'Circe', Ulysses, 1922*

Frequently, to achieve the constraints on running time imposed by looplessness, we will need to use intermediate types, and, in particular, trees. For example, suppose we would like to manipulate the elements of a list. We may define an abstraction function *abst* in the *prolog* of a loopless algorithm with type signature

$$abst \quad :: \quad [a] \to Tree\ a$$

which builds a tree from the list. We can eventually recover this list in steps defined by a tree traversal *traverse* with type signature

$$traverse \quad :: \quad Tree\ a \to [a]$$

Common traversals are preorder, postorder and inorder. It will help us to make two of these traversals loopless.

2.1. Preorder traversal of rose trees

Preorder traversal of binary trees is defined recursively:

(i) Visit the root.

(ii) Traverse the left subtree.

(iii) Traverse the right subtree.

For forests of rose trees, we visit the root of the first tree, traverse each subtree of this tree in preorder and then traverse the other trees in the forest in turn. The preorder traversal of the forest in Figure 2.1 is therefore

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10.$$

In postorder traversals of a binary trees, the root is visited after each of its subtrees:

(i) Traverse the left subtree.

(ii) Traverse the right subtree.

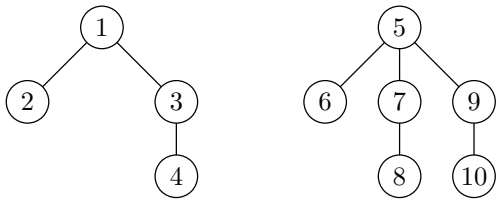


Figure 2.1. A forest of rose trees.

(iii) Visit the root.

This translates to forests of rose trees in an analogous way to preorder. Figure 2.1 reduces in this order to

$$2, 4, 3, 1, 6, 8, 7, 10, 9, 5.$$

We now consider a functional implementation of the preorder traversal of a rose tree. These [1] have the type

$$\textbf{data}\ \mathit{Rose}\ a \quad = \quad \mathit{Node}\ a\ [\mathit{Rose}\ a]$$

The obvious way to define the traversal *preorder* is recursively using the prelude functions *concat* and *map*:

$$\begin{array}{lcl} \mathit{preorder} & :: & \mathit{Rose}\ a \to [a] \\ \mathit{preorder}\ (\mathit{Node}\ x\ \mathit{xts}) & = & x : \mathit{concat}\ (\mathit{map}\ \mathit{preorder}\ \mathit{xts}) \end{array}$$

Unfortunately, this is not efficient, but we can introduce an intermediate function *ptf* such that

$$\mathit{ptf} \quad = \quad \mathit{concat} \cdot \mathit{map}\ \mathit{preorder}$$

We can define *ptf* directly by

$$\begin{array}{lcl} \mathit{ptf} & :: & [\mathit{Rose}\ a] \to [a] \\ \mathit{ptf}\ [\,] & = & [\,] \\ \mathit{ptf}\ (\mathit{Node}\ x\ \mathit{xts} : \mathit{yts}) & = & x : \mathit{ptf}\ (\mathit{xts} +\!\!+ \mathit{yts}) \end{array}$$

It follows that *ptf* computes the preorder traversal of a forest of rose trees. Therefore,

$$\begin{array}{lcl} \mathit{preorder} & = & \mathit{ptf} \cdot \mathit{wrap} \\ & & \textbf{where}\ \mathit{wrap}\ \mathit{xt} \quad = \quad [\mathit{xt}] \end{array}$$

This version of *preorder* takes $O(n)$ time in the size of the rose tree. Moreover, we can cast *ptf* into the loopless form

$$\mathit{ptf} \quad = \quad \mathit{unfoldr}\ \mathit{step}$$

where

$$\begin{array}{lcl} \mathit{step}\ [\,] & = & \mathit{Nothing} \\ \mathit{step}\ (\mathit{Node}\ x\ \mathit{xts} : \mathit{yts}) & = & \mathit{Just}\ (x, \mathit{xts} +\!\!+ \mathit{yts}) \end{array}$$

We need the function *step* to take $O(1)$ time, so we wrap each rose tree in a further list and process elements of the list of lists instead:

$$\begin{array}{lcl} \mathit{step}\ [\,] & = & \mathit{Nothing} \\ \mathit{step}\ ((\mathit{Node}\ x\ \mathit{xts} : \mathit{yts}) : \mathit{tss}) & = & \mathit{Just}\ (x, \mathit{consList}\ \mathit{xts}\ (\mathit{consList}\ \mathit{yts}\ \mathit{tss})) \end{array}$$

[1] Bird (2005b). See also Gibbons & Jones (1998).

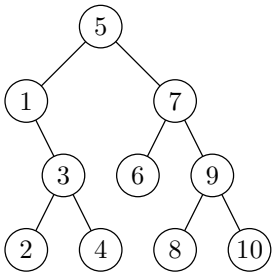


Figure 2.2. A binary tree.

Then our loopless functional algorithm for *preorder* is simply

$$\begin{array}{lcl} \mathit{preorder} & = & \mathit{unfoldr}\ \mathit{step} \cdot \mathit{doubleWrap} \\ & & \textbf{where}\ \mathit{doubleWrap}\ xt \quad = \quad [[xt]] \end{array}$$

2.2. Inorder traversal of binary trees

A more symmetric way to traverse binary trees is inorder. This is defined as follows:

(i) Traverse the left subtree.

(ii) Visit the root.

(iii) Traverse the right subtree.

There is no simple interpretation of inorder traversal for rose trees since there is no obvious place to insert the root amongst its descendants. Figure 2.2 is traversed inorder as

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10.$$

We represent binary trees functionally by

$$\textbf{data}\ \mathit{Tree}\ a \quad = \quad \mathit{Null} \mid \mathit{Fork}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Then *inorder* may be defined recursively using concatenation: [2]

$$\begin{array}{lcl} \mathit{inorder}\ \mathit{Null} & = & [\,] \\ \mathit{inorder}\ (\mathit{Fork}\ x\ lt\ rt) & = & \mathit{inorder}\ lt +\!\!+ x : \mathit{inorder}\ rt \end{array}$$

Now, suppose we convert the binary tree into a list of spines of the right subtrees along the path of the leftmost node to the root using a forest of rose trees. This requires a function *mkSpines* defined by

$$\begin{array}{lcl} \mathit{mkSpines} & :: & \mathit{Tree}\ a \to [\mathit{Rose}\ a] \\ \mathit{mkSpines}\ t & = & \mathit{addSpines}\ t\ [\,] \end{array}$$

where

$$\begin{array}{lcl} \mathit{addSpines}\ \mathit{Null}\ sps & = & sps \\ \mathit{addSpines}\ (\mathit{Fork}\ x\ lt\ rt)\ sps & = & \mathit{addSpines}\ lt\ (\mathit{Node}\ x\ (\mathit{mkSpines}\ rt) : sps) \end{array}$$

[2] Bird (2005b).

It follows that the spines of the binary tree in Figure 2.2 are described by the forest in Figure 2.1. We can also see that the preorder traversal of this forest is identical to the inorder traversal of the binary tree. Since this is the case for all binary trees, inorder traversal may be defined using *mkSpines* and the preorder traversal *ptf* from Section 2.1:

$$\begin{array}{lcl} \mathit{inorder} & = & \mathit{unfoldr}\ \mathit{step} \cdot \mathit{wrapList} \cdot \mathit{mkSpines} \\ & & \textbf{where}\ \ \mathit{wrapList}\ \mathit{xs} \ = \ \mathit{consList}\ \mathit{xs}\ [\,] \end{array}$$

This is a loopless functional algorithm for *inorder*.

Chapter 3

Fundamental Laws

> *The following are the universally fundamental laws of literary communication:*
>
> 1. *one must have something to communicate;*
> 2. *one must have someone to whom to communicate it;*
> 3. *one must really communicate it, not merely express it for oneself alone.*
>
> *Otherwise it would be more to the point to remain silent.*
>
> Friedrich Von Schlegel, *'Aphorism 98', Selected Aphorisms from the Lyceum, 1797*

We will often be able to formulate functional algorithms for combinatorial generation in terms of the function *foldr*. To make progress in transforming these into loopless functional algorithms, we will make use of two fundamental laws of functional programming.

3.1. The fusion law

The fusion law tells as that, assuming certain conditions hold, a strict function applied to a value assembled from a list can be produced directly from the elements of that list. [1]

THEOREM 3.1. If the function f is strict, and it satisfies $f\ a = b$ and $f\ (g\ x\ y) = h\ x\ (f\ y)$ for all x and y, then

$$f \cdot foldr\ g\ a \quad = \quad foldr\ h\ b$$

The condition relating f with the functions g and h ensures that it may be computed incrementally. The proof of this theorem is a simple induction using the definition of *foldr*.

We illustrate fusion by considering the prelude function *map* defined by

$$\begin{array}{lcl} map & :: & (a \to b) \to a \to b \\ map\ g\ [\,] & = & [\,] \\ map\ g\ (x : xs) & = & g\ x\ (map\ g\ xs) \end{array}$$

Suppose we wish to fuse *foldr* and *map* such that

$$foldr\ f\ a \cdot map\ g \quad = \quad foldr\ h\ b$$

Since *foldr* f a is strict, we can apply the fusion law by expressing *map* in terms of *foldr*:

$$map\ g \quad = \quad foldr\ ((:) \cdot g)\ [\,]$$

From this it is easy to derive a corollary of the fusion law known as fold-*map* fusion:

[1] Bird (1998) 131–132.

THEOREM 3.2.

$$foldr\ f\ e \cdot map\ g \quad = \quad foldr\ (f \cdot g)\ e$$

Therefore, a *map* followed by a fold can always be expressed as a single fold.

3.2. The banana-split law

The banana-split law when defined in terms of *foldr* is stated as follows: [2]

THEOREM 3.3.

$$fork\ (foldr\ f\ a, foldr\ g\ b) \quad = \quad foldr\ h\ (a, b)$$

where

$$\begin{array}{lcl} fork\ (f, g)\ x & = & (f\ x, g\ x) \\ h\ x\ (y, z) & = & (f\ x\ y, g\ x\ z) \end{array}$$

As an example of the application of this law, consider the function *average* which calculates the average value of a list of integers:

$$average \quad = \quad div \cdot fork\ (sum, length)$$

where

$$\begin{array}{lcl} div\ (0,0) & = & 0 \\ div\ (x, y) & = & x\ /\ y \\ \\ sum & = & foldr\ (+)\ 0 \\ \\ length & = & foldr\ succ\ 0 \\ & & \textbf{where}\ succ\ x\ n \quad = \quad n+1 \end{array}$$

We set $div\ (0,0) = 0$ to avoid problems when dealing with empty lists. Clearly, this definition of *average* requires the input list to be traversed twice. However, by applying the banana-split law, we can achieve a modest increase in efficiency by reducing this to a single traversal:

$$\begin{array}{lcl} average' & = & div \cdot foldr\ pluss\ (0,0) \\ & & \textbf{where}\ pluss\ x\ (y, n) \quad = \quad (x+y, n+1) \end{array}$$

[2] Bird & de Moor (1997) 55–58. The banana-split law is so named because, in its general form, it applies to pairs of catamorphisms. These are normally represented by banana-shaped brackets (⦇ ⦈), while the pairing operator is sometimes referred to as 'split'.

Chapter 4

Queues

An Englishman, even if he is alone, forms an orderly queue of one.

GEORGE MIKES, *How To Be An Alien, 1946*

When working under the constraints of an $O(n)$ time *prolog* and $O(1)$ time *step*, we need an efficient way to add and remove elements to and from the beginning and end of lists. In the case of queue-like structures, a function *remove* defined by

$$\begin{array}{lcl} \mathit{remove} & :: & [a] \to (a, [a]) \\ \mathit{remove}\ (x : xs) & = & (x, xs) \end{array}$$

clearly takes $O(1)$ time. However, the following function *insert* takes $O(n)$ time in the length of its input list:

$$\begin{array}{lcl} \mathit{insert} & :: & [a] \to a \to [a] \\ \mathit{insert}\ xs\ x & = & xs +\!\!+ [x] \end{array}$$

We could reverse the list, but then, while *insert* would be improved to $O(1)$ time, *remove* would fall back to $O(n)$ time. Fortunately, there are several ways to improve this situation and achieve both $O(1)$ time insertions and deletions.

4.1. Amortized queues using paired lists

Our first method uses a pair of lists to represent a queue:[1]

$$\mathbf{type}\ \mathit{Queue}\ a \quad = \quad ([a], [a])$$

Denoting these lists as (xs, ys), the front part of the queue is held in xs, while the rear part is reversed and stored in ys. Therefore, the first element of the queue is at the head of xs and the last element is at the head of ys. It follows that as long as neither xs nor ys are empty, we can access the elements we require in $O(1)$ time. Hence, *insert* is defined by

$$\mathit{insert}\ (xs, ys)\ x \quad = \quad (xs, x : ys)$$

Now, a complication occurs when the front of the queue is empty. Then the last element of the queue is the last of ys, so we must reverse ys and use this as the front of the queue, while setting the rear to be empty. This operation can be carried out when an element is removed. Therefore, we have

$$\begin{array}{lcl} \mathit{remove}\ ([\,], ys) & = & \mathit{remove}\ (\mathit{reverse}\ ys, [\,]) \\ \mathit{remove}\ (x : xs, ys) & = & (x, (xs, ys)) \end{array}$$

[1] Okasaki (1995) 584–585 and Okasaki (1998) 42–44, based upon Hood & Melville (1981).

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Queue}\ a & = & ([a], [a]) \\
\mathit{empty} & = & ([\,], [\,]) \\
\mathit{isEmpty}\ (xs, ys) & = & \mathit{null}\ xs \\
\mathit{insert}\ (xs, ys)\ x & = & (xs, x : ys) \\
\mathit{remove}\ ([\,], ys) & = & \mathit{remove}\ (\mathit{reverse}\ ys, [\,]) \\
\mathit{remove}\ (x : xs, ys) & = & (x, (xs, ys))
\end{array}
$$

Figure 4.1. Amortized queues using paired lists.

This takes $O(n)$ time in the length of the list because of the $O(n)$ time list reversal. However, as this only occurs when there have been n insertions since the last removal, we can amortize the cost over these insertions, giving us $O(1)$ amortized time operations.

Figure 4.1 shows the full implementation including a representation of an empty queue and a test for emptiness.

4.2. Amortized queues using laziness and incremental computation

We can improve from $O(n)$ to $O(\log n)$ worst case time by making use of laziness and performing the list reversal incrementally instead of all at once. Therefore, we must ensure that we begin the list reversal early enough so that a first element is always available when required. [2]

To prevent the occurrence of a long delay for some removals, we introduce a rotation *rot* which periodically replaces (xs, ys) with $(xs +\!\!+ ys, [\,])$. Because the concatenation operation is already incremental, we can make rotation incremental by carrying out one step of the reversal in every step of the concatenation. To define *rot* we use an accumulating parameter *zs*:

$$
\begin{array}{lcl}
\mathit{rot}\ [\,]\ (y : ys)\ zs & = & y : zs \\
\mathit{rot}\ (x : xs)\ (y : ys)\ zs & = & x : \mathit{rot}\ xs\ ys\ (y : zs)
\end{array}
$$

We maintain an invariant

$$
\mathit{length}\ ys \quad \leq \quad \mathit{length}\ xs
$$

and perform rotations only when this would be violated. This is encapsulated in the function *chkQueue* shown with the rest of the implementation in Figure 4.2.

4.3. Real-time queues using pre-evaluation

For our purposes, it is not sufficient to bound operations by $O(1)$ amortized time. We need to guarantee $O(1)$ time, even in the worst case. To achieve this, we must pre-evaluate the front of the queue to ensure that no tail of the list *xs* takes more than $O(1)$ time to compute. We introduce a third list *zs*, which will act as a pointer into *xs*, resulting in the new type [3]

$$
\textbf{type}\ \mathit{Queue}\ a \quad = \quad ([a], [a], [a])
$$

[2] Okasaki (1995) 586–587.

[3] Okasaki (1995) 587–588.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Queue}\ a & = & ([a],[a]) \\
\mathit{empty} & = & ([\,],[\,]) \\
\mathit{isEmpty}\ (xs,ys) & = & \mathit{null}\ xs \\
\mathit{insert}\ (xs,ys)\ x & = & \mathit{chkQueue}\ (xs, x:ys) \\
\mathit{remove}\ (x:xs,ys) & = & (x, \mathit{chkQueue}\ (xs,ys)) \\
\mathit{chkQueue}\ (xs,ys) & = & \textbf{if}\ \mathit{length}\ ys \leq \mathit{length}\ xs\ \textbf{then}\ (xs,ys) \\
 & & \textbf{else}\ (\mathit{rot}\ xs\ ys\ [\,],[\,]) \\
\mathit{rot}\ [\,]\ (y:ys)\ zs & = & y:zs \\
\mathit{rot}\ (x:xs)\ (y:ys)\ zs & = & x:\mathit{rot}\ xs\ ys\ (y:zs)
\end{array}
$$

Figure 4.2. Amortized queues using laziness and incremental computation.

When zs is the empty list, the entire list has been pre-evaluated. On each call to *insert* or *remove* that does not cause a rotation to take place, the function *chkQueue* advances zs by one position, pre-evaluating the next tail:

$$
\begin{array}{lcll}
\mathit{chkQueue}\ (xs,ys,[\,]) & = & (zs,[\,],zs) & \\
 & & \textbf{where}\ zs & = \quad \mathit{rot}\ xs\ ys\ [\,] \\
\mathit{chkQueue}\ (xs,ys,z:zs) & = & (xs,ys,zs) &
\end{array}
$$

Following a rotation, zs is set to xs. By maintaining the invariant

$$\mathit{length}\ zs \quad = \quad \mathit{length}\ xs - \mathit{length}\ ys$$

we can ensure that zs is empty by the next rotation. Therefore, we have the guaranteed $O(1)$ time implementation shown in Figure 4.3 on the page that follows.

```
type Queue a          =  ([a], [a], [a])

empty                 =  ([ ], [ ], [ ])

isEmpty (xs, ys, zs)  =  null xs

insert (xs, ys, zs) x =  chkQueue (xs, x : ys, zs)

remove (x : xs, ys, zs) = (x, chkQueue (xs, ys, zs))

chkQueue (xs, ys, [ ])  =  (zs, [ ], zs)
                           where zs = rot xs ys [ ]
chkQueue (xs, ys, z : zs) = (xs, ys, zs)

rot [ ] [y] zs        =  y : zs
rot (x : xs) (y : ys) zs = x : rot xs ys (y : zs)
```

Figure 4.3. Real-time queues using pre-evaluation.

Part II

Mixing

Chapter 5

Mixing with Rose Trees

> *What a lovely thing a rose is! ... Our highest assurance of the goodness of Providence seems to me to rest in the flowers. All other things, our powers, our desires, our food, are all really necessary for our existence in the first instance. But the rose is an extra. Its smell and its colour are an embellishment of life, not a condition of it. It is only goodness which gives extras.*
>
> Sir Arthur Conan Doyle, *'The Naval Treaty', The Memoirs of Sherlock Holmes, 1893*

Many of the combinatorial functional algorithms that we will formulate are linked by a construction that we will refer to as 'mixing'. We define this using simple functions *mix* and *mixall* that act on finite lists. If we can derive a loopless functional algorithm for *mixall*, then we may be able to use this to ensure our combinatorial functional algorithms are also loopless. There are several ways to make *mixall* loopless, the first of these uses rose trees as an intermediate type.

5.1. Mix and mixall

Our definition of the mixing function *mix* is as follows: [1]

$$\begin{array}{lcl} mix & :: & [a] \to [a] \to [a] \\ mix\ [\,]\ ys & = & ys \\ mix\ (x : xs)\ ys & = & ys +\!\!+ x : mix\ xs\ (reverse\ ys) \end{array}$$

This interleaves each element of the first list with, alternatively, the entire second list or the reverse of that list. Therefore,

$$mix\ [3,4,5,6]\ [0,1,2] \quad = \quad [0,1,2,3,2,1,0,4,0,1,2,5,2,1,0,6,0,1,2]$$

The function *mix* is associative with the empty list as its identity element. This is easily proved by induction using the intermediate results

$$\begin{array}{lcl} mix\ (xs +\!\!+ y : ys)\ zs & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ mix\ xs\ zs +\!\!+ y : mix\ ys\ (reverse\ zs) \\ & & \textbf{else}\ mix\ xs\ zs +\!\!+ y : mix\ ys\ zs \end{array}$$

$$\begin{array}{lcl} reverse\ (mix\ xs\ ys) & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ mix\ (reverse\ xs)\ (reverse\ ys) \\ & & \textbf{else}\ mix\ (reverse\ xs)\ ys \end{array}$$

We define a function *mixall* for mixing the lists in a list of lists by [2]

[1] Bird (2005b).

$$\begin{array}{lcl} mixall & :: & [[a]] \to [a] \\ mixall & = & foldr\ mix\ [\,] \end{array}$$

For example, by mixing the list of lists $[[6,7],[3,4,5],[1,2]]$, we obtain the result

$$[1,2,3,2,1,4,1,2,5,2,1,6,1,2,5,2,1,4,1,2,3,2,1,7,1,2,3,2,1,4,1,2,5,2,1]$$

When applied to a list of n lists of m elements, the length of *mixall* is exponential in mn, the total length of the input.

5.2. Fusion and banana-splitting

We begin by casting *mixall* into 'loopless form'. This will be a function $unfoldr\ step \cdot prolog$ where *step* and *prolog* do not necessarily meet the conditions for looplessness. Our first step is to use the fusion law to express $reverse \cdot mixall$ in terms of *foldr*. Clearly, the prelude function *reverse* is strict and reversal does not change an empty list. Therefore, we need only find a function *xim* such that

$$reverse\ (mix\ xs\ ys) \quad = \quad xim\ xs\ (reverse\ ys)$$

The definition of *xim* we will use in this section is recursive, obtained by reversing the terms on either side of the concatenation in our definition of *mix*: [3]

$$\begin{array}{lcl} xim\ [\,]\ ys & = & ys \\ xim\ (x : xs)\ ys & = & xim\ xs\ (reverse\ ys) +\!\!+ x : ys \end{array}$$

Abbreviating $reverse \cdot mixall$ to *ximall*, we have

$$ximall \quad = \quad foldr\ xim\ [\,]$$

By the banana-split law, we can combine the evaluation of *mixall* and *ximall* into one computation:

$$\begin{array}{lcl} fork\ (mixall, ximall) & = & foldr\ pmix\ ([\,],[\,]) \\ & & \textbf{where}\ pmix\ xs\ (ys, sy) \quad = \quad (mix\ xs\ ys, xim\ xs\ sy) \end{array}$$

If we assume that (ys, sy) satisfies $sy = reverse\ ys$, then by an inductive argument using the definitions of *mix* and *xim*, we can express *pmix* recursively, independent of *mixall* and *ximall*, by

$$\begin{array}{lcl} pmix\ [\,]\ (ys, sy) & = & (ys, sy) \\ pmix\ (x : xs)\ (ys, sy) & = & (ys +\!\!+ x : us, vs +\!\!+ x : sy) \\ & & \textbf{where}\ (us, vs) \quad = \quad pmix\ xs\ (sy, ys) \end{array}$$

Using an associative operator $\oslash$, we also have

$$\begin{array}{lcl} pmix\ (x : xs)\ (ys, sy) & = & (ys, sy) \oslash ([x],[x]) \oslash pmix\ xs\ (sy, ys) \\ & & \textbf{where}\ (xs, sx) \oslash (ys, sy) \quad = \quad (xs +\!\!+ ys, sy +\!\!+ xs) \end{array}$$

From this definition, together with the trivial identity $fst \cdot fork\ (f, g) = f$ it follows that

$$mixall \quad = \quad fst \cdot foldr\ pmix\ ([\,],[\,])$$

[2] As *mix* is associative we could also have defined *mixall* in terms of *foldl* without affecting the result.

[3] We will meet a non-recursive definition of *xim* in Section 6.1.

5.3. Casting mixall into loopless form using fission and rose trees

To obtain our loopless form, we need to split *foldr* into two separate folds. This could be described as using the fusion law in the opposite direction to before. Hence, we call this step 'fission'.

We represent the elements of the type [a] by the elements of some intermediate type $T\ a$ under an abstraction function *abst* with type signature

$$abst \quad :: \quad T\ a \to [a]$$

This is linked to *pmix* using a function *tmix* with type signature

$$tmix \quad :: \quad [a] \to (T\ a, T\ a) \to (T\ a, T\ a)$$

which satisfies the condition

$$pair\ abst \cdot tmix\ xs \quad = \quad pmix\ xs \cdot pair\ abst$$

where

$$pair\ f\ (x, y) \quad = \quad (f\ x, f\ y)$$

To apply fission, $T\ a$ must contain a constructor *Null* which will correspond to the empty list. It follows that

$$foldr\ pmix\ ([\,], [\,]) \quad = \quad pair\ abst \cdot foldr\ tmix\ (Null, Null)$$

As $fst \cdot pair\ f = f \cdot fst$, we have

$$mixall \quad = \quad abst \cdot fst \cdot foldr\ tmix\ (Null, Null)$$

If we are to obtain a loopless algorithm for *mixall*, then we need to choose $T\ a$ and *abst*, and derive *tmix* such that it does less work than *pmix* while *abst* does more than none at all.

In Section 2.1, we derived a loopless functional algorithm for the preorder traversal of a rose tree. It would seem reasonable to choose $T\ a$ as a rose tree and *abst* as this tree traversal. However, there is no *Null* constructor for empty trees. A simple solution is to take $T\ a$ as *Maybe* (*Rose a*), with *Nothing* representing the empty list. It follows that

$$preorder \quad = \quad unfoldr\ step \cdot wrapList \cdot wrapTree$$

where *wrapTree* is defined by

$$\begin{array}{lcl} wrapTree\ Nothing & = & [\,] \\ wrapTree\ (Just\ xt) & = & [xt] \end{array}$$

and the definition of *step* is unchanged:

$$\begin{array}{lcl} step\ [\,] & = & Nothing \\ step\ ((Node\ x\ xts : yts) : tss) & = & Just\ (x, consList\ xts\ (consList\ yts\ tss)) \end{array}$$

Now, we need a suitable definition of *tmix*. We can take this to be the same as *pmix* except that we replace $\oslash$ with $\otimes$:

$$\begin{array}{lcl} tmix\ [\,]\ (myt, mty) & = & (myt, mty) \\ tmix\ (x : xs)\ (myt, mty) & = & (myt, mty) \otimes (mxt, mxt) \otimes tmix\ xs\ (mty, myt) \\ & & \textbf{where}\ mxt \quad = \quad Just\ (Node\ x\ [\,]) \end{array}$$

It follows that the condition imposed on *tmix* holds provided

$$\begin{aligned} &\mathit{pair}\ \mathit{preorder}\ ((\mathit{mxt}, \mathit{mtx}) \otimes (\mathit{myt}, \mathit{mty})) \\ &\qquad = \quad \mathit{pair}\ \mathit{preorder}\ (\mathit{mxt}, \mathit{mtx}) \oslash \mathit{pair}\ \mathit{preorder}\ (\mathit{myt}, \mathit{mty}) \end{aligned}$$

Although not associative, we can choose to define $\otimes$ by

$$(\mathit{mxt}, \mathit{mtx}) \otimes (\mathit{myt}, \mathit{mty}) \quad = \quad (\mathit{mxt} \odot \mathit{myt}, \mathit{mty} \odot \mathit{mtx})$$

where

$$\begin{array}{lcl} \mathit{Nothing} \odot \mathit{myt} & = & \mathit{myt} \\ \mathit{mxt} \odot \mathit{Nothing} & = & \mathit{mxt} \\ \mathit{Just}\ (\mathit{Node}\ x\ \mathit{xts}) \odot \mathit{Just}\ \mathit{yt} & = & \mathit{Just}\ (\mathit{Node}\ x\ (\mathit{xts} +\!\!+ [\mathit{yt}])) \end{array}$$

By application of the fusion law, we have [4]

$$\mathit{mixall} \quad = \quad \mathit{preorder} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Nothing}, \mathit{Nothing})$$

Therefore, our loopless form for *mixall* is

$$\mathit{mixall} \quad = \quad \mathit{unfoldr}\ \mathit{step} \cdot \mathit{prolog}$$

where

$$\mathit{prolog} \quad = \quad \mathit{wrapList} \cdot \mathit{wrapTree} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Nothing}, \mathit{Nothing})$$

A graphical illustration of *prolog* applied to the list of lists $[[6,7],[3,4,5],[1,2]]$ is shown in Figure 5.1. Clearly, the preorder traversal of this rose tree is equal to the result of evaluating *mixall* with the same input. If we substituted *snd* for *fst* in *prolog*, we would have a loopless form for *reverse* · *mixall* instead.

5.4. Replacing lists with real-time queues

We have *mixall* in loopless form, but we need to ensure that *step* takes $O(1)$ time and *prolog* takes $O(n)$ time in the total length of the input list. While our definition of *step* already satisfies its part of this joint constraint, the running time of *prolog* is $O(n)$ only if *tmix xs* is $O(n)$ in the length of the list *xs*. Moreover, this is only the case if $\otimes$ takes $O(1)$ time. However, the operator $\otimes$ is defined in terms of $\odot$ and

$$\mathit{Just}\ (\mathit{Node}\ x\ \mathit{xts}) \odot \mathit{Just}\ \mathit{yt} \quad = \quad \mathit{Just}\ (\mathit{Node}\ x\ (\mathit{xts} +\!\!+ [\mathit{yt}]))$$

so it follows that $\mathit{mxt} \odot \mathit{myt}$ takes time proportional to the number of children of *mxt*.

[4] If we instead chose to use binary trees and inorder traversal, then *tmix* would be defined by

$$\begin{array}{lcl} \mathit{tmix}\ [\,]\ (\mathit{yt}, \mathit{ty}) & = & (\mathit{yt}, \mathit{ty}) \\ \mathit{tmix}\ (x : \mathit{xs})\ (\mathit{yt}, \mathit{ty}) & = & (\mathit{Fork}\ \mathit{yt}\ x\ \mathit{zt}, \mathit{Fork}\ \mathit{tz}\ x\ \mathit{ty}) \\ & & \mathbf{where}\ (\mathit{zt}, \mathit{tz}) \quad = \quad \mathit{tmix}\ \mathit{xs}\ (\mathit{ty}, \mathit{yt}) \end{array}$$

and *mixall* would become

$$\mathit{mixall} \quad = \quad \mathit{inorder} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Null}, \mathit{Null})$$

Since we have to convert binary trees to rose trees to carry out the loopless inorder traversal, *prolog* would then be defined by

$$\mathit{prolog} \quad = \quad \mathit{unfoldr}\ \mathit{step} \cdot \mathit{wrapList} \cdot \mathit{mkSpines} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Null}, \mathit{Null})$$

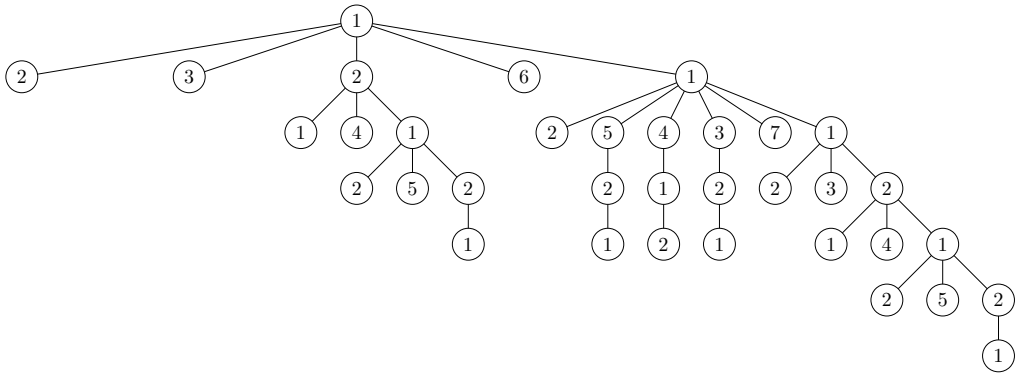


Figure 5.1. A rose tree built by prolog.

Recalling Chapter 4, we can see that the list of children of a rose tree in this version of *mixall* is effectively a queue. The function $\odot$ inserts a tree at the end of a list, while *step* removes lists of lists from the front. Therefore, we can apply real-time queues to reduce the running time of *prolog* to $O(n)$.

To achieve the conversion from lists to queues, we first replace the list of children in the definition of rose trees with queues:

$$\textbf{data}\ \mathit{Rose}\ a \quad = \quad \mathit{Node}\ a\ (\mathit{Queue}\ (\mathit{Rose}\ a))$$

Then we can define, for example, functions *wrapQueue* and *consQueue* analogous to *wrapList* and *consList* for lists:

$$\begin{array}{lcl} \mathit{wrapQueue}\ \mathit{xtq} & = & \mathit{consQueue}\ \mathit{xtq}\ [\,] \\ \\ \mathit{consQueue}\ \mathit{xtq}\ \mathit{xtqs} & = & \textbf{if}\ \mathit{isEmpty}\ \mathit{xtq}\ \textbf{then}\ \mathit{xtqs} \\ & & \textbf{else}\ \mathit{xtq} : \mathit{xtqs} \end{array}$$

It follows that *prolog* becomes

$$\mathit{prolog} \quad = \quad \mathit{wrapQueue} \cdot \mathit{wrapTree} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Nothing}, \mathit{Nothing})$$

while *step* simplifies to

$$\begin{array}{lcl} \mathit{step}\ [\,] & = & \mathit{Nothing} \\ \mathit{step}\ (\mathit{xtq} : \mathit{xtqs}) & = & \mathit{Just}\ (x, \mathit{consQueue}\ \mathit{ytq}\ (\mathit{consQueue}\ \mathit{ztq}\ \mathit{xtqs})) \\ & & \textbf{where}\ (\mathit{Node}\ x\ \mathit{ytq}, \mathit{ztq}) \quad = \quad \mathit{remove}\ \mathit{xtq} \end{array}$$

The full translation of our loopless form using lists to a loopless functional algorithm using real-time queues appears in Figure 5.2 overleaf.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Rose}\ a & = & \mathit{Node}\ a\ (\mathit{Queue}\ (\mathit{Rose}\ a)) \\
 & & \\
\mathit{mixall} & = & \mathit{unfoldr}\ \mathit{step} \cdot \mathit{prolog} \\
 & & \\
\mathit{prolog} & = & \mathit{wrapQueue} \cdot \mathit{wrapTree} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{tmix}\ (\mathit{Nothing}, \mathit{Nothing}) \\
 & & \\
\mathit{tmix}\ [\,]\ (\mathit{myt}, \mathit{mty}) & = & (\mathit{myt}, \mathit{mty}) \\
\mathit{tmix}\ (x : \mathit{xs})\ (\mathit{myt}, \mathit{mty}) & = & (\mathit{myt}, \mathit{mty}) \otimes (\mathit{mxt}, \mathit{mxt}) \otimes \mathit{tmix}\ \mathit{xs}\ (\mathit{mty}, \mathit{myt}) \\
 & & \textbf{where}\ \mathit{mxt} \quad = \quad \mathit{Just}\ (\mathit{Node}\ x\ \mathit{empty}) \\
 & & \\
(\mathit{mxt}, \mathit{mtx}) \otimes (\mathit{myt}, \mathit{mty}) & = & (\mathit{mxt} \odot \mathit{myt}, \mathit{mty} \odot \mathit{mtx}) \\
 & & \\
\mathit{Nothing} \odot \mathit{myt} & = & \mathit{myt} \\
\mathit{mxt} \odot \mathit{Nothing} & = & \mathit{mxt} \\
\mathit{Just}\ (\mathit{Node}\ x\ \mathit{xtq}) \odot \mathit{Just}\ \mathit{yt} & & \\
 & = & \mathit{Just}\ (\mathit{Node}\ x\ (\mathit{insert}\ \mathit{xtq}\ \mathit{yt})) \\
 & & \\
\mathit{wrapTree}\ \mathit{Nothing} & = & \mathit{empty} \\
\mathit{wrapTree}\ (\mathit{Just}\ \mathit{xt}) & = & \mathit{insert}\ \mathit{empty}\ \mathit{xt} \\
 & & \\
\mathit{wrapQueue}\ \mathit{xtq} & = & \mathit{consQueue}\ \mathit{xtq}\ [\,] \\
 & & \\
\mathit{consQueue}\ \mathit{xtq}\ \mathit{xtqs} & = & \textbf{if}\ \mathit{isEmpty}\ \mathit{xtq}\ \textbf{then}\ \mathit{xtqs} \\
 & & \textbf{else}\ \mathit{xtq} : \mathit{xtqs} \\
 & & \\
\mathit{step}\ [\,] & = & \mathit{Nothing} \\
\mathit{step}\ (\mathit{xtq} : \mathit{xtqs}) & = & \mathit{Just}\ (\mathrm{x}, \mathit{consQueue}\ \mathit{ytq}\ (\mathit{consQueue}\ \mathit{ztq}\ \mathit{xtqs})) \\
 & & \textbf{where}\ (\mathit{Node}\ x\ \mathit{ytq}, \mathit{ztq}) \quad = \quad \mathit{remove}\ \mathit{xtq}
\end{array}
$$

Figure 5.2. A loopless functional algorithm for mixall using rose trees.

Chapter 6

Mixing with Forests

> *'Mankind is getting smarter every day'. Actually, it only seems so. 'At least we are making progress'. We're progressing, to be sure, ever more deeply into the forest.*
>
> FRANZ GRILLPARZER, *'Natural Sciences', Poems, 1853*

While a loopless functional algorithm using rose trees is sufficient for most applications we will study, it will be constructive to derive a different version along similar lines, but using forests of rose trees as an intermediate instead.

6.1. Another definition of xim

In Section 5.2, we recursively defined the function *xim* satisfying

$$reverse\ (mix\ xs\ ys) \quad = \quad xim\ xs\ (reverse\ ys)$$

Clearly, we can also express *xim* directly in terms of *mix*:

$$\begin{array}{lcl} xim\ xs\ ys & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ mix\ (reverse\ xs)\ (reverse\ ys) \\ & & \textbf{else}\ mix\ (reverse\ xs)\ ys \end{array}$$

The function *ximall* is again defined by

$$ximall \quad = \quad foldr\ xim\ [\,]$$

Now, consider the following alternative definition of *mix* using the prelude function *zipWith* with an infinite list of infinite lists:

$$mix\ xs\ ys \quad = \quad gmix\ (ys, reverse\ ys)$$

where

$$\begin{array}{lcl} gmix\ xs\ (ys, sy) & = & ys +\!\!+ concat\ (zipWith\ (:)\ xs\ sys) \\ & & \textbf{where}\ sys \quad = \quad sy : ys : sys \end{array}$$

We can then use the banana-split law to obtain

$$fork\ (mixall, ximall) \quad = \quad foldr\ pmix\ ([\,], [\,])$$

with *pmix* defined by

$$\begin{array}{lcl} pmix\ xs\ (ys, sy) & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ (gmix\ xs\ (ys, sy), gmix\ sx\ (sy, ys)) \\ & & \textbf{else}\ (gmix\ xs\ (ys, sy), gmix\ sx\ (ys, sy)) \\ & & \textbf{where}\ sx \quad = \quad reverse\ xs \end{array}$$

6.2. Casting mixall into loopless form using fission and forests

Continuing in a similar way to our derivation in Chapter 5, we introduce an intermediate type and suitable abstraction function. In this case, we will use a forest of rose trees, built by preorder traversal, to represent a list of lists. It follows that we require a new function *tmix* satisfying

$$pair\ ptf \cdot tmix\ xs \quad = \quad pmix\ xs \cdot pair\ ptf$$

We have

$$foldr\ pmix\ ([\,],[\,]) \quad = \quad pair\ ptf \cdot foldr\ tmix\ ([\,],[\,])$$

Therefore,

$$mixall \quad = \quad ptf \cdot fst \cdot foldr\ tmix\ ([\,],[\,])$$

Now, suppose we have a function *fmix* that satisfies

$$ptf\ (fmix\ xs\ (yts, sty)) \quad = \quad gmix\ xs\ (ptf\ yts, ptf\ sty)$$

Then we may define *tmix* by

$$\begin{array}{lcl} tmix\ xs\ (yts, sty) & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ (fmix\ xs\ (yts, sty), fmix\ sx\ (sty, yts)) \\ & & \textbf{else}\ (fmix\ xs\ (yts, sty), fmix\ sx\ (yts, sty)) \\ & & \textbf{where}\ sx \quad = \quad reverse\ xs \end{array}$$

Fortunately, an appropriate *fmix* is given by the simple definition

$$\begin{array}{lcl} fmix\ xs\ (yts, sty) & = & yts +\!\!+ (zipWith\ Node\ xs\ sys) \\ & & \textbf{where}\ sys \quad = \quad sty : yts : sys \end{array}$$

This results in another loopless form for *mixall*:

$$mixall \quad = \quad unfoldr \cdot step \cdot prolog$$

where

$$prolog \quad = \quad wrapList \cdot fst \cdot foldr\ tmix\ ([\,],[\,])$$

Figure 6.1 shows the three forests of rose trees generated by *prolog* $[[6,7],[3,4,5],[1,2]]$. Again we obtain a loopless form for $reverse \cdot mixall$ by exchanging *fst* with *snd*.

6.3. Introducing real-time queues

We know that *step* takes $O(1)$ time. Unfortunately, our new definition of *prolog* is not quite sufficient for our needs. Therefore, we introduce real-time queues to achieve $O(n)$ time in the total length of the input list. As before, we redefine rose trees, but we also replace list concatenations by a function *append* with type signature

$$append \quad :: \quad [a] \to Queue\ a \to Queue\ a$$

This is defined by

$$append \quad = \quad foldl\ insert$$

It follows that *fmix* for queues is given by

$$\begin{array}{lcl} fmix\ xs\ (ytq, qty) & = & append\ ytq\ (zipWith\ Node\ xs\ qyqs) \\ & & \textbf{where}\ qyqs \quad = \quad qty : ytq : qyqs \end{array}$$

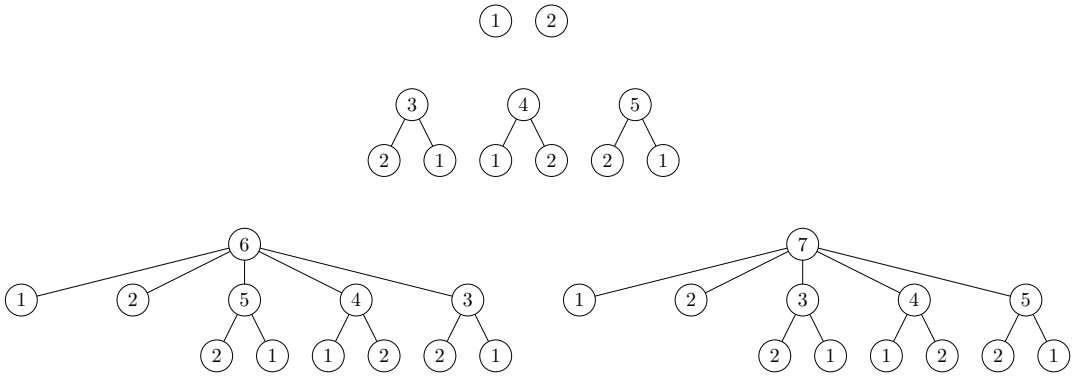


Figure 6.1. Forests of rose trees built by prolog.

Then *prolog* is simply modified to

$$prolog \quad = \quad wrapQueue \cdot fst \cdot foldr\ tmix\ (empty, empty)$$

The full listing of our second loopless functional algorithm for *mixall* appears in Figure 6.2 at the end of this chapter.

6.4. Mixing lists generated by a loopless functional algorithm

Using this loopless algorithm, we can consider whether it is possible to also derive a loopless algorithm if the input lists to *mixall* are generated by another loopless algorithm instead of being given explicitly. More precisely, we would like to express

$$mixall \cdot map\ (unfoldr\ step' \cdot prolog')$$

as a loopless functional algorithm provided that we have additional functions $step''$ and $prolog''$ such that

$$unfoldr\ step'' \cdot prolog'' \quad = \quad reverse \cdot unfoldr\ step' \cdot prolog'$$

and a function $length'$ that satisfies

$$length' \quad = \quad length \cdot unfoldr\ step' \cdot prolog'$$

It will be sufficient for our purposes to partially limit this generalization by adding the further conditions

$$prolog' \quad = \quad fst \cdot pg \qquad prolog'' \quad = \quad snd \cdot pg \qquad step' \quad = \quad sp \quad = \quad step''$$

for suitable functions sp and pg. Then we have

$$reverse \cdot unfoldr\ sp \cdot fst \cdot pg \quad = \quad unfoldr\ sp \cdot snd \cdot pg$$

To derive our generalized loopless functional algorithm, we begin with our latest loopless version of *mixall*. It follows that

$$prolog \quad = \quad wrapQueue \cdot fst \cdot foldr\ tmix\ (empty, empty) \cdot map\ (unfoldr\ sp \cdot pg)$$

Clearly, we need to modify our definition of *tmix* to eliminate *map* (*unfoldr sp*·*pg*). Therefore, we need to replace each finite list *xs* in its definition with a list generated by the loopless algorithm. We can achieve this by defining a function that calculates values a, *b* and *n* that represent, respectively, the application of $fst \cdot pg$, $snd \cdot pg$ and $length'$ to *xs* and then applying *unfoldr sp* to its output within *tmix* and *fmix*. It follows that our revised definition of *prolog* is

$$prolog\ sp\ lg\ pg \quad = \quad wrapQueue \cdot fst \cdot foldr\ (tmix\ sp)\ (empty, empty) \cdot list\ lg\ pg$$

where

$$\begin{array}{lcl}
list\ lg\ pg\ xs & = & [(lg\ x, a, b) \mid x \leftarrow xs,\ (a, b) \leftarrow [pg\ x]] \\
\\
tmix\ sp\ (n, a, b)\ (ytq, qty) & = & \textbf{if}\ even\ n \\
 & & \textbf{then}\ (fmix\ sp\ a\ (ytq, qty), fmix\ sp\ b\ (qty, ytq)) \\
 & & \textbf{else}\ (fmix\ sp\ a\ (ytq, qty), fmix\ sp\ b\ (ytq, qty)) \\
\\
fmix\ sp\ a\ (ytq, qty) & = & append\ ytq\ (zipWith\ Node\ (unfoldr\ sp\ a)\ qyqs \\
 & & \textbf{where}\ qyqs = qty : ytq : qyqs
\end{array}$$

Unfortunately, this change means that *fmix* now takes exponential time in the worst case. Therefore, we need to delay the evaluation of *unfoldr sp a* and then compute it within *step*. We represent delayed evaluation by the type

$$\textbf{data}\ Delay\ a\ b \quad = \quad Hold\ a\ b\ (Queue\ (Delay\ a\ b), Queue\ (Delay\ a\ b))$$

Then we redefine *fmix* to obtain

$$\begin{array}{lcll}
fmix\ sp\ a\ (ytq, qty) & = & \textbf{case}\ sp\ a\ \textbf{of} & \\
 & & Nothing & \rightarrow\ ytq \\
 & & Just\ (x, b) & \rightarrow\ insert\ ytq\ (Hold\ x\ b\ (ytq, qty))
\end{array}$$

We also need a new definition of *step* as follows:

$$\begin{array}{lcl}
step\ sp\ [\,] & = & Nothing \\
step\ sp\ (xtq : xtqs) & = & Just\ (x, consQueue\ (fmix\ sp\ a\ (qty, ytq))\ (consQueue\ ztq\ xtqs)) \\
 & & \textbf{where}\ (Hold\ x\ a\ (ytq, qty), ztq) \quad = \quad remove\ xtq
\end{array}$$

These new functions take $O(1)$ time because *sp* does by the assumption that it is part of a loopless functional algorithm. Hence,

$$mixall \cdot map\ (unfoldr\ sp \cdot fst \cdot pg) \quad = \quad unfoldr\ (step\ sp) \cdot prolog\ sp\ lg\ pg$$

```
data Rose a      =  Node a (Queue (Rose a))

mixall           =  unfoldr step · prolog

prolog           =  wrapQueue · fst · foldr tmix (empty, empty)

tmix xs (ytq, qty)  =  if even (length xs)
                       then (fmix xs (ytq, qty), fmix sx (qty, ytq))
                       else (fmix xs (ytq, qty), fmix sx (ytq, qty))
                       where sx = reverse xs

fmix xs (ytq, qty)  =  append ytq (zipWith Node xs qyqs)
                       where qyqs = qty : ytq : qyqs

append           =  foldl insert

wrapQueue xtq    =  consQueue xtq [ ]

consQueue xtq xtqs  =  if isEmpty xtq then xtqs
                       else xtq : xtqs

step [ ]         =  Nothing
step (xtq : xtqs)  =  Just (x, consQueue ytq (consQueue ztq xtqs))
                      where (Node x ytq, ztq) = remove xtq
```

Figure 6.2. A loopless functional algorithm for mixall using forests.

Chapter 7

Mixing with Lists

> *I made a list of things I have to remember and a list of things I want to forget, but I see they are the same list.*
>
> LINDA PASTAN, *'Lists', 1982*

It turns out that we can replace the forests in the loopless functional algorithm calculated in Chapter 6 with lists and eliminate the need for real-time queues.

7.1. Building forests from two lists

Consider the following function *mkRows* which builds a pair of lists (tr, br) from a list of lists:

$$\begin{array}{lcl} mkRows & :: & [[a]] \to ([(a,(Int,Int))],[(a,(Int,Int))]) \\ mkRows & = & pair\ reverse \cdot snd \cdot foldr\ op\ ((0,0),([\,],[\,])) \end{array}$$

where *op* is defined using the prelude function *zip* applied to an infinite list:

$$\begin{array}{lcl} op\ xs\ ((p,q),(tr,br)) & = & \textbf{if}\ even\ n\ \textbf{then}\ ((p,q+n),(reverse\ cs +\!\!+ tr, cs +\!\!+ br)) \\ & & \textbf{else}\ ((p+q,n),(reverse\ cs +\!\!+ tr, cs +\!\!+ br)) \\ & & \textbf{where}\ n = length\ xs \\ & & \quad\ cs = zip\ xs\ pqs \\ & & \quad\ pqs = (p,q) : (p+q,0) : pqs \end{array}$$

It follows that

$$mkRows\ [[6,7],[3,4,5],[1,2]] \quad = \quad (tr, br)$$

where

$$\begin{array}{lcl} tr & = & [(1,(0,0)),(2,(0,0)),(3,(0,2)),(4,(2,0)),(5,(0,2)),(6,(2,3)),(7,(5,0))] \\ br & = & [(2,(0,0)),(1,(0,0)),(5,(0,2)),(4,(2,0)),(3,(0,2)),(7,(5,0)),(6,(2,3))] \end{array}$$

We now wish use these rows to construct the forests of rose trees pictured in Figure 6.1. Then we will be able to use the function *step* defined in Section 2.1 to traverse these forests in preorder and obtain a loopless form for *mixall*.

To achieve the conversion from rows to forests, we first define a function *mkTree* with type signature

$$mkRose \quad :: \quad ([(a,(Int,Int))],[(a,(Int,Int))]) \to (a,(Int,Int)) \to Rose\ a$$

which constructs a rose tree with the element x at its root:

$$\begin{array}{lcl} mkRose\ rs\ (x,(p,q)) & = & Node\ x\ (take\ p\ (map\ (mkRose\ rs)\ (fst\ rs)) \\ & & \quad +\!\!+ take\ q\ (drop\ p\ (map\ (mkRose\ rs)\ (snd\ rs)))) \end{array}$$

The integers p and q can now be seen to represent adding children whose roots are the elements at positions 0 to $p-1$ in tr and p to $p+q-1$ in br. It follows that we can use this to define a function *mkForests* which builds forests:

$$mkForests\ rs \quad = \quad wrapList\ (map\ (mkRose\ rs))\ (fst\ rs))$$

Hence, we have the loopless form

$$mixall \quad = \quad unfoldr\ step \cdot prolog$$

where

$$prolog \quad = \quad mkForests \cdot mkRows$$

If replace *fst* with *snd* in the definition of *mkForests*, then we have a loopless form for *reverse* · *mixall* instead.

7.2. Removing the conversion to forests

To avoid using rose trees, we begin by replacing *mkForest* with a function *extract* defined by

$$\begin{array}{lcl} extract\ ((p,q),rs) & = & (sr, wrapRow\ ((p+q,0),sr)) \\ & & \textbf{where}\ sr \quad = \quad pair\ reverse\ rs \end{array}$$

where

$$\begin{array}{lcl} consRow\ ((p,q),rs)\ r & = & \textbf{if}\ p == 0 \wedge q == 0\ \textbf{then}\ r \\ & & \textbf{else}\ ((p,q),rs) : r \end{array}$$

$$wrapRow\ r \quad = \quad consRow\ r\ [\,]$$

If we modify our definition of *mkRows* to

$$mkRows' \quad = \quad foldr\ op\ ((0,0),([\,],[\,]))$$

then we have

$$prolog \quad = \quad extract \cdot mkRows'$$

It follows that *step* becomes

$$\begin{array}{lcl} step\ (rs,[\,]) & = & Nothing \\ step\ (rs,((p,q),(t:tr,b:br)):r) & = & \textbf{if}\ p == 0 \\ & & \textbf{then}\ next\ rs\ (b, consRow\ ((p,q-1),(tr,br))\ r) \\ & & \textbf{else}\ next\ rs\ (t, consRow\ ((p-1,q),(tr,br))\ r) \end{array}$$

where

$$next\ rs\ ((x,(p,q)),r) \quad = \quad Just\ (x,(rs, consRow\ ((p,q),rs)\ r))$$

The full definition of this loopless algorithm for *mixall* is given in Figure 7.1 on the next page.

```
data Rose a                          =  Node a [Rose a]

mixall                               =  unfoldr step · prolog

prolog                               =  extract · mkRows′

mkRows′                              =  foldr op ((0, 0), ([ ], [ ]))

op xs ((p, q), (tr, br))             =  if even n
                                        then ((p, q + n), (reverse cs ++ tr, cs ++ br))
                                        else ((p + q, n), (reverse cs ++ tr, cs ++ br))
                                        where n   =  length xs
                                              cs  =  zip xs pqs
                                              pqs =  (p, q) : (p + q, 0) : pqs

extract ((p, q), rs)                 =  (sr, wrapRow ((p + q, 0), sr))
                                        where sr  =  pair reverse rs

pair f (x, y)                        =  (f x, f y)

wrapRow r                            =  consRow r [ ]

consRow ((p, q), rs) r               =  if p == 0 ∧ q == 0 then r
                                        else ((p, q), rs) : r

next rs ((x, (p, q)), r)             =  Just (x, (rs, consRow ((p, q), rs) r))

step (rs, [ ])                       =  Nothing
step (rs, ((p, q), (t : tr, b : br)) : r)  =  if p == 0
                                        then next rs (b, consRow ((p, q − 1), (tr, br)) r)
                                        else next rs (t, consRow ((p − 1, q), (tr, br)) r)
```

Figure 7.1. A loopless functional algorithm for mixall using lists.

Chapter 8

Mixing with Binary Trees

A tree's a tree. How many more do you need to look at?

RONALD REAGAN, *Speech opposing the expansion of Redwood National Park, 1965*

So far our loopless functional algorithms for *mixall* have been derived using rose trees, in some form, and real-time queues. However, it is possible to formulate an algorithm using binary trees instead, and we are able to retain a list structure to avoid making essential use of laziness.

8.1. Mixorder traversal of binary trees

Consider the definition of binary trees:

$$\textbf{data}\ \mathit{Tree}\ a \quad = \quad \mathit{Null} \mid \mathit{Fork}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

In Section 2.2, we considered the inorder traversal of this type of tree. However, suppose that the labels of the tree are lists. Then we can define a traversal which we will call 'mixorder':

(i) Traverse the left subtree.

(ii) Visit first element of the root.

(iii) Traverse the right subtree.

(iv) Visit the second element of the root

(v) Traverse the left subtree.

...

This continues with visits to each element the root alternating with traversals of the left and right subtrees until all elements of the tree have been produced. Mixorder may be implemented functionally by

$$\begin{array}{lcl} \mathit{mixorder} & :: & \mathit{Tree}\ [a] \to [a] \\ \mathit{mixorder}\ \mathit{Null} & = & [\,] \\ \mathit{mixorder}\ (\mathit{Fork}\ [\,]\ lt\ rt) & = & \mathit{mixorder}\ lt \\ \mathit{mixorder}\ (\mathit{Fork}\ (x : xs)\ lt\ rt) & = & \mathit{mixorder}\ lt +\!\!+ x : \mathit{mixorder}\ (\mathit{Fork}\ xs\ rt\ lt) \end{array}$$

It follows that we can relate *mixall* to *mixorder*:

$$\mathit{mixall} \quad = \quad \mathit{mixorder} \cdot \mathit{mkTree}$$

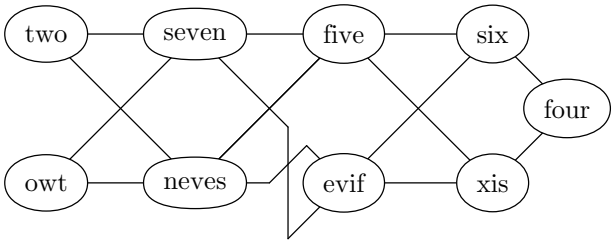


Figure 8.1. A perfect binary tree built by mkTree.

where

$$\begin{array}{lcl} \mathit{mkTree} & :: & [[a]] \to \mathit{Tree}\ [a] \\ \mathit{mkTree} & = & \mathit{foldr}\ \mathit{op}\ \mathit{Null} \\ \\ \mathit{op}\ [\,]\ t & = & t \\ \mathit{op}\ \mathit{xs}\ t & = & \mathit{Node}\ \mathit{xs}\ t\ (\mathit{switch}\ t) \\ \\ \mathit{switch}\ \mathit{Null} & = & \mathit{Null} \\ \mathit{switch}\ (\mathit{Fork}\ \mathit{xs}\ \mathit{lt}\ \mathit{rt}) & = & \textbf{if}\ \mathit{even}\ (\mathit{length}\ \mathit{xs})\ \textbf{then}\ \mathit{Fork}\ (\mathit{reverse}\ \mathit{xs})\ \mathit{rt}\ \mathit{lt} \\ & & \textbf{else}\ \mathit{Fork}\ (\mathit{reverse}\ \mathit{xs})\ \mathit{lt}\ \mathit{rt} \end{array}$$

This may be proved by the fusion law using the intermediate result

$$\mathit{mixorder}\ (\mathit{op}\ \mathit{xs}\ t) \quad = \quad \mathit{mix}\ (\mathit{mixorder}\ t)\ \mathit{xs}$$

The function *mkTree* builds a perfect tree with left subtrees labelled with the non-empty lists of the input and right subtrees with the reverse of each of these lists. The order of the children in the right subtrees is dependent on the parity of the length of the label. Since the left and right subtrees are shared, we construct the tree in $O(n)$ time for an input of total length n. The result of

$$\mathit{mkTree}\ [\text{“}\mathit{two}\text{”}, \text{“}\mathit{seven}\text{”}, \text{“}\mathit{five}\text{”}, \text{“}\mathit{six}\text{”}, \text{“}\mathit{four}\text{”}]$$

is shown in Figure 8.1, where for clarity we chosen our list of lists to be a list of strings.

8.2. Casting mixorder into loopless form using spines

We cast *mixorder* into loopless form using a similar method to that of inorder traversal of a binary tree. However, as we need to swap between the spines of both left and right subtrees, we decorate the rose trees that we will construct as follows:

$$\textbf{data}\ \mathit{Rose}\ a \quad = \quad \mathit{Node}\ a\ ([\mathit{Rose}\ a], [\mathit{Rose}\ a])$$

Then the function *mkSpines*, which converts a binary tree to a list of spines, may be easily defined by

$$\begin{array}{lcl} \mathit{mkSpines} & :: & \mathit{Tree}\ a \to [\mathit{Rose}\ a] \\ \mathit{mkSpines}\ t & = & \mathit{addSpines}\ t\ [\,] \end{array}$$

where

$$\begin{array}{lcl} \mathit{addSpines}\ \mathit{Null}\ \mathit{sps} & = & \mathit{sps} \\ \mathit{addSpines}\ (\mathit{Fork}\ x\ \mathit{lt}\ \mathit{rt})\ \mathit{sps} & = & \mathit{addSpines}\ \mathit{lt}\ (\mathit{Node}\ x\ (\mathit{lsp}, \mathit{rsp}) : \mathit{sps}) \\ & & \textbf{where}\ (\mathit{lsp}, \mathit{rsp}) \quad = \quad (\mathit{mkSpines}\ \mathit{lt}, \mathit{mkSpines}\ \mathit{rt}) \end{array}$$

We can improve on this definition and eliminate the repeated evaluation of *addSpines lt* by noticing that

$$addSpines\ t\ sps \quad = \quad mkSpines\ t +\!\!+ sps$$

Therefore, we have a new version of *mkSpines* that takes $O(n)$ instead of exponential time in the size of the tree, and in which *addSpines* is defined as follows:

$$\begin{array}{lcl} addSpines\ Null\ sps & = & sps \\ addSpines\ (Fork\ x\ lt\ rt)\ sps & = & sps +\!\!+ Node\ x\ (lsp, rsp) : sps \\ & & \textbf{where}\ (lsp, rsp) \quad = \quad (mkSpines\ lt, mkSpines\ rt) \end{array}$$

This also allows us to convert a perfect binary size of size n to a list of spines in $O(n)$ time.

Using a variation our existing definition of *step* for inorder traversal given by

$$\begin{array}{lcl} step\ [\,] & = & Nothing \\ step\ ((Node\ (x : xs)\ (lsp, rsp) : sp) : sps) & & \\ & = & \textbf{if}\ null\ xs\ \textbf{then}\ Just\ (x, consList\ rsp\ (consList\ sp\ sps)) \\ & & \textbf{else}\ Just\ (x, consList\ rsp\ ((Node\ xs\ (rsp, lsp) : sp) : sps)) \end{array}$$

we have a loopless form for *mixorder*:

$$mixorder \quad = \quad unfoldr\ step \cdot wrapList \cdot mkSpines$$

Our *step* function takes $O(1)$ time and swaps between subspines *lsp* and *rsp*.

Since we can define *mixall* in terms of *mixorder* and *mkTree* it follows that

$$mixall \quad = \quad unfoldr\ step \cdot wrapList \cdot mkSpines \cdot mkTree$$

8.3. Restoring the sharing of spines

While *prolog* takes $O(n)$ time in the size of the tree built by *mkTree*, it takes exponential time in the total length of the input list. The sharing of spines evident in Figure 8.1 has been lost. Therefore, we need to fuse *mkSpines* and *mkTree* to restore this.

We begin by defining functions *mkTreePairs* and *mkSpinePairs* that output pairs of trees and spines, respectively:

$$\begin{array}{lcl} mkTreePairs\ xss & = & (mkTree\ xss, switch\ (mkTree\ xss)) \\ \\ mkSpinePairs\ (s, t) & = & (mkSpines\ s, mkSpines\ t) \end{array}$$

Now, we have

$$switch\ (op\ xs\ t) \quad = \quad op'\ xs\ (switch\ t)$$

where

$$\begin{array}{lcl} op'\ [\,]\ s & = & s \\ op'\ xs\ s & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ Fork\ (reverse\ xs)\ s\ (switch\ s) \\ & & \textbf{else}\ Fork\ (reverse\ xs)\ (switch\ s)\ s \end{array}$$

and it follows that

$$switch \cdot mkTree \quad = \quad foldr\ op'\ Null$$

Moreover, using the banana-split law, we obtain

$$mkTreePairs \quad = \quad foldr\ opp\ (Null, Null)$$

where

$$\begin{array}{lcl} opp\ [\,]\ (s,t) & = & (s,t) \\ opp\ xs\ (s,t) & = & \textbf{if}\ even\ (length\ xs)\ \textbf{then}\ (Fork\ xs\ s\ t, Fork\ (reverse\ xs)\ t\ s) \\ & & \textbf{else}\ (Fork\ xs\ s\ t, Fork\ (reverse\ xs)\ s\ t) \end{array}$$

It is also easy to show that

$$mkSpinePairs\ (opp\ xs\ (s,t)) \quad = \quad addPair\ xs\ (mkSpinePairs\ (s,t))$$

where

$$\begin{array}{lcl} addPair\ [\,]\ (lsp,rsp) & = & (lsp,rsp) \\ addPair\ xs\ (lsp,rsp) & = & \textbf{if}\ even\ (length\ xs) \\ & & \textbf{then}\ (lsp +\!\!+ [Node\ xs\ (lsp,rsp)], rsp +\!\!+ [Node\ sx\ (rsp,lsp)]) \\ & & \textbf{else}\ (lsp +\!\!+ [Node\ xs\ (lsp,rsp)], lsp +\!\!+ [Node\ sx\ (lsp,rsp)]) \\ & & \textbf{where}\ sx\ =\ reverse\ xs \end{array}$$

Using the fusion law, we can therefore define a function *shareSpines* which restores the sharing of the spines in our tree:

$$shareSpines \quad = \quad foldr\ addPair\ ([\,],[\,])$$

This allows us to modify our loopless form for *mixall* to

$$mixall \quad = \quad unfoldr\ step \cdot wrapList \cdot fst \cdot shareSpines$$

We similarly have

$$reverse \cdot mixall \quad = \quad unfoldr\ step \cdot wrapList \cdot snd \cdot shareSpines$$

8.4. Splicing to achieve looplessness

Our new *prolog* is an improvement, but it still takes $O(n^2)$ time, falling short of the $O(n)$ time that we require. Since *addPair* adds new elements to the rear of each spine *lsp* and *rsp*, and *step* deletes elements from the front of these spines, we could use the approach previously employed and introduce real-time queues. Instead, however, we will continue to derive an implementation that does not require the use of laziness in this way.

As we wish to avoid adding elements to the end of the lists of spines, we could instead cons them to the front and reverse the lists afterwards. For example, we could replace *shareSpines* with a new function *shareSpines′* defined by

$$shareSpines' \quad = \quad pair\ reverse \cdot foldr\ addPair'\ ([\,],[\,])$$

where

$$\begin{array}{lcl} addPair'\ [\,]\ (psl,psr) & = & (psl,psr) \\ addPair'\ xs\ (psl,psr) & = & \textbf{if}\ even\ (length\ xs) \\ & & \textbf{then}\ (Node\ xs\ (lsp,rsp) : psl, Node\ sx\ (rsp,lsp) : psr) \\ & & \textbf{else}\ (Node\ xs\ (lsp,rsp) : psl, Node\ sx\ (lsp,rsp) : psl) \\ & & \textbf{where}\ (lsp,rsp) \ = \ pair\ reverse\ (psl,psr) \\ & & \qquad\quad sx \ = \ reverse\ xs \end{array}$$

Unfortunately, while we have removed the expensive concatenations, we now need to compute *pair reverse* at each step, so *addPair′* still takes $O(n)$ time.

To make progress, we need to exploit a pattern in the form of the spines *lsp* and *rsp*. If *lsp* is the left spine of a node at a distance n in the top row tr and *rsp* is the left spine of the corresponding node in the bottom row br, then *lsp* is the sequence formed by taking the first n elements of tr and *rsp* is the sequence formed by taking p elements from tr and q elements from br, where $p + q = n$. This can be encapsulated in a function *splice* defined by

$$splice\ (p, q)\ (tr, br) \quad = \quad take\ p\ tr +\!\!+ take\ q\ (drop\ p\ br)$$

and it follows that

$$\begin{array}{lcl} lsp & = & splice\ (p+q, 0)\ (tr, br) \\ rsp & = & splice\ (p, q)\ (tr, br) \end{array}$$

Therefore, if we can calculate p and q in each step, then we can extract *lsp* and *rsp* from tr and br.

To implement this idea, we need to add extra information to the labels of our spines. This is the pair of integers (p, q) and a boolean value *swap* which tells us whether the spines *lsp* and *rsp* need to be swapped. Then we have the following definition of *shareSpines′*:

$$\begin{array}{l} shareSpines'\ xss \\ \quad = \quad pair\ reverse\ (foldr\ addPair'\ ([\,],[\,])\ xss) \\ \qquad \textbf{where}\ addPair'\ xs\ (psl, psr) \quad = \quad \textbf{if}\ null\ xs\ \textbf{then}\ (psl, psr) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (Node\ (xs, False, p, q)\ rs : psl, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Node\ (sx, even\ (length\ xs), p, q)\ rs : psr) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where}\ sx \quad = \quad reverse\ xs \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ (p, q) \quad = \quad shape\ psr \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ rs \quad = \quad shareSpines'\ xss \end{array}$$

We also define *shape*, based upon our observations about the particular form of spines:

$$\begin{array}{lcl} shape\ [\,] & = & (0, 0) \\ shape\ (Node\ (x : xs, swap, p, q)\ rs : psr) & = & \textbf{if}\ swap\ \textbf{then}\ (p, q+1) \\ & & \textbf{else}\ (p+q, 1) \end{array}$$

Therefore, the new version of *step* incorporating our idea of splicing is given by

$$\begin{array}{l} step\ [\,] \quad = \quad Nothing \\ step\ (Node\ (x : xs, swap, p, q)\ rs : ps) \\ \qquad\qquad = \quad \textbf{if}\ null\ xs\ \textbf{then}\ Just\ (x, sp +\!\!+ ps) \\ \qquad\qquad\quad \textbf{else}\ Just\ (x, sp +\!\!+ Node\ (xs, \neg swap, p, q)\ rs : ps) \\ \qquad\qquad\quad \textbf{where}\ \ sp \quad = \quad \textbf{if}\ swap\ \textbf{then}\ splice\ (p+q, 0)\ rs \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ splice\ (p, q)\ rs \end{array}$$

8.5. Delaying evaluation of splice

We have a loopless functional algorithm for *mixall* under lazy evaluation, but under strict evaluation our definition of *splice* (p, q) *rs* takes $O(p + q)$ steps. Therefore, we need to redefine *step* using a further modified definition of rose trees:

$$\textbf{data}\ Rose\ a \quad = \quad Node\ a\ ([Rose\ a], [Rose\ a]) \mid Splice\ (Int, Int)\ ([Rose\ a], [Rose\ a])$$

The extra constructor allows us to delay the evaluation of *splice* to ensure that *step* has the same characteristics under both strict and lazy evaluation. It follows that *step* is defined by

$$
\begin{array}{lcl}
\mathit{step}\ [\,] & = & \mathit{Nothing} \\
\mathit{step}\ (\mathit{Node}\ (x : \mathit{xs}, \mathit{swap}, p, q)\ \mathit{rs} : \mathit{ps}) & & \\
\qquad & = & \textbf{if}\ \mathit{null}\ \mathit{xs}\ \textbf{then}\ \mathit{Just}\ (x, \mathit{consSplice}\ \mathit{sp}\ \mathit{ps}) \\
\qquad & & \textbf{else}\ \mathit{Just}\ (x, \mathit{consSplice}\ \mathit{sp}\ (\mathit{Node}\ (\mathit{xs}, \neg \mathit{swap}, p, q)\ \mathit{rs} : \mathit{ps})) \\
\qquad & & \textbf{where}\ \ \mathit{sp} \ = \ \textbf{if}\ \mathit{swap}\ \textbf{then}\ \mathit{Splice}\ (p + q, 0)\ \mathit{rs} \\
\qquad & & \qquad\qquad \textbf{else}\ \mathit{Splice}\ (p, q)\ \mathit{rs} \\
\mathit{step}\ (\mathit{Splice}\ (p, q)\ (t : \mathit{tr}, b : \mathit{br}) : \mathit{ps}) & & \\
\qquad & = & \textbf{if}\ p == 0\ \textbf{then}\ \mathit{step}\ (b : \mathit{consSplice}\ (\mathit{Splice}\ (p, q - 1)\ (\mathit{tr}, \mathit{br}))\ \mathit{ps}) \\
\qquad & & \textbf{else}\ \mathit{step}\ (t : \mathit{consSplice}\ (\mathit{Splice}\ (p - 1, q)\ (\mathit{tr}, \mathit{br}))\ \mathit{ps})
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathit{consSplice}\ (\mathit{Splice}\ (p, q)\ \mathit{rs})\ \mathit{ps} & = & \textbf{if}\ p == 0 \wedge q == 0\ \textbf{then}\ \mathit{ps} \\
\qquad & & \textbf{else}\ \mathit{Splice}\ (p, q)\ \mathit{rs} : \mathit{ps}
\end{array}
$$

Because *tr* and *br* are lists of labels of nodes and not *Splice* values, this definition of *step* takes $O(1)$ time and we have a loopless functional algorithm for *mixall* derived using binary trees:

$$\mathit{mixall} \ = \ \mathit{unfoldr}\ \mathit{step} \cdot \mathit{prolog}$$

where

$$\mathit{prolog} \ = \ \mathit{fst} \cdot \mathit{shareSpines}'$$

The full listing of this algorithm is shown in Figure 8.2 overleaf.

```
data Rose a      =  Node a ([Rose a], [Rose a]) | Splice (Int, Int) ([Rose a], [Rose a])

mixall           =  unfoldr step · prolog

prolog           =  fst · shareSpines′

shareSpines′ xss
    =  pair reverse (foldr addPair′ ([ ], [ ]) xss)
       where addPair′ xs (psl, psr)  =  if null xs then (psl, psr)
                                        else (Node (xs, False, p, q) rs : psl,
                                              Node (sx, even (length xs), p, q) rs : psr)
                                        where sx      =  reverse xs
                                              (p, q)  =  shape psr
                                              rs      =  shareSpines′ xss

shape [ ]        =  (0, 0)
shape (Node (x : xs, swap, p, q) rs : psr)
                 =  if swap then (p, q + 1)
                    else (p + q, 1)

pair f (x, y)    =  (f x, f y)

consSplice (Splice (p, q) rs) ps
                 =  if p == 0 ∧ q == 0 then ps
                    else Splice (p, q) rs : ps

step [ ]         =  Nothing
step (Node (x : xs, swap, p, q) rs : ps)
                 =  if null xs then Just (x, consSplice sp ps)
                    else Just (x, consSplice sp (Node (xs, ¬swap, p, q) rs : ps))
                    where sp  =  if swap then Splice (p + q, 0) rs
                                 else Splice (p, q) rs
step (Splice (p, q) (t : tr, b : br) : ps)
                 =  if p == 0 then step (b : consSplice (Splice (p, q − 1) (tr, br)) ps)
                 =  else step (t : consSplice (Splice (p − 1, q) (tr, br)) ps)
```

Figure 8.2. A loopless functional algorithm for mixall using binary trees.

Part III

Applications

Chapter 9

Gray Codes

> *It is impossible, of course, to remove all ambiguity in the lowest-order digit except by a scheme like one the Irish railways are said to have used of removing the last car of every train because it is too susceptible to collision damage.*
>
> George R. Stibitz & Jules A. Larrivee, *Mathematics and Computers, 1957*

Suppose we wish to generate all subsets of a finite set $\{x_{n-1}, \ldots, x_1, x_0\}$. This can be achieved by visiting every n-tuple $(a_{n-1}, \ldots, a_1, a_0)$ where each a_j is either 0 or 1. Then we can say that x_j is a member of the subset if and only if $a_j = 1$. There are many orders in which we can run through all n-tuples, but we will focus on the binary Gray code which visits all 2^n strings of n bits such that exactly one bit changes in each step in a simple and regular way.

9.1. Binary reflected Gray codes

The best known example of a binary Gray code is the binary reflected Gray code. If G_n denotes the sequence of all strings of n bits in this order, then we have the following recursive definition:

$$G_n = \begin{cases} \epsilon & \text{if } n = 0 \\ 0G_{n-1},\ 1G_{n-1}^R & \text{otherwise} \end{cases}$$

where ϵ is the empty bit string, $0G_{n-1}$ is the sequence G_{n-1} with a 0 bit prefixed to each string and $1G_{n-1}^R$ is the reverse of the sequence G_{n-1} with a 1 bit prefixed to each string. It is clear that exactly one bit changes in every step because the last string of G_{n-1} is equal to the first string of G_{n-1}^R. The first and last strings differ by one bit, so the sequence is a cycle as displayed in Figure 9.1. For example, the binary reflected Gray code for $n = 4$ is shown in Figure 9.2 with the bit complemented in each step highlighted.

A straightforward imperative algorithm for generating binary reflected Gray codes is as follows:

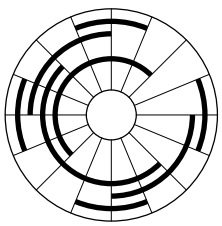


Figure 9.1. A graphical representation of a binary reflected Gray code.

$$\begin{array}{cccc} 000\bar{0} & 011\bar{0} & 110\bar{0} & 101\bar{0} \\ 00\bar{0}1 & 01\bar{1}1 & 11\bar{0}1 & 10\bar{1}1 \\ 001\bar{1} & 010\bar{1} & 111\bar{1} & 100\bar{1} \\ 0\bar{0}10 & \bar{0}100 & 1\bar{1}10 & \bar{1}000 \end{array}$$

Figure 9.2. A binary reflected Gray code.

ALGORITHM A (Binary reflected Gray generation). This algorithm generates all binary n-tuples $(a_{n-1}, \ldots, a_1, a_0)$, starting with $(0, \ldots, 0, 0)$ and changing exactly one bit in each step. We maintain a parity bit a_∞ such that [1]

$$a_\infty = a_{n-1} \oplus \cdots \oplus a_1 \oplus a_0.$$

A1. (Initialize) Set $a_j \leftarrow 0$ for $0 \le j < n$. Also set $a_\infty \leftarrow 0$.

A2. (Visit) Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

A3. (Change parity) Set $a_\infty \leftarrow 1 - a_\infty$.

A4. (Choose j) If $a_\infty = 1$, set $j \leftarrow 0$. Otherwise let $j \ge 1$ be the minimum such that $a_{j-1} = 1$.

A5. (Complement coordinate j) Terminate if $j = n$. Otherwise set $a_j \leftarrow 1 - a_j$ and return to A2.

This algorithm is based upon a method for solving the Chinese ring puzzle. The goal of this puzzle is to remove interlocking rings from a bar. By the nature of the rings, only two basic types of move are possible:

(i) The rightmost ring can be removed or replaced at any time.

(ii) Any other ring can be removed and replaced if and only if the ring to its right is on the bar and all rings to the right of that one are off.

If we indicate that a ring is on the bar by a 1 bit and off the bar by a 0 bit, then we wish to reach the n-tuple $(0, \ldots, 0, 0)$. A binary reflected Gray code results from reversing the steps taken to achieve this.

While we only complement one bit a_j per visit to $(a_{n-1}, \ldots, a_1, a_0)$, we need the loop in A4 to choose the appropriate value of j. However, in this case, we can make the algorithm faster by replacing the loop with focus pointers which implicitly represent a value that allows us to calculate j directly. Therefore, we have a loopless imperative algorithm for generating binary reflected Gray codes:

ALGORITHM B (Loopless Binary reflected Gray generation). This loopless algorithm generates all binary n-tuples $(a_{n-1}, \ldots, a_1, a_0)$, starting with $(0, \ldots, 0, 0)$ and changing exactly one bit in each step. Instead of maintaining a parity bit, we use an array of focus pointers $(f_n, \ldots, f_1, f_0)$. [2]

B1. (Initialize) Set $a_j \leftarrow 0$ and $f_j \leftarrow j$ for $0 \le j < n$. Also set $f_n \leftarrow n$.

B2. (Visit) Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

[1] Knuth (2005a) 6.

[2] Knuth (2005a) 9–11, based upon Bitner *et al.* (1976).

B3. (Choose j) Set $j \leftarrow f_0$ and $f_0 \leftarrow 0$. Terminate if $j = n$, otherwise set $f_j \leftarrow f_{j+1}$ and $f_{j+1} \leftarrow j + 1$.

B4. (Complement coordinate j) Set $a_j \leftarrow 1 - a_j$. Return to B2.

We perform only five assignment operations and one test for termination between each visit to a generated n-tuple.

9.2. Mixing and Gray codes

Functionally, the transitions in the Gray code algorithm can be easily expressed using *mixall*: [3]

$$\begin{array}{lcl} gray & = & mixall \cdot singletons \\ & & \textbf{where}\ singletons\ n \ = \ [[i] \mid i \leftarrow [n-1, n-2\,..\,0]] \end{array}$$

Our transitions are chosen simply to be the indices j of the coordinates $(a_{n-1}, \ldots, a_1, a_0)$ that we should complement. For example, the transitions necessary to generate the binary reflected Gray code for $n = 4$ in Figure 9.2 are

$$0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0.$$

Clearly, most transitions occur at the least significant bit, so our definition of *mix*, which we could describe as mixing from the right, is ideal.

We can make this loopless by composing our loopless functional algorithm for *mixall* using rose trees from Chapter 5 with *singletons*. However, because the two trees generated by *foldr tmix* $(Nothing, Nothing)$ have the same preorder traversals in the case of Gray codes, we need only one tree and can simplify *prolog* as follows:

$$prolog\ n \ = \ (wrapQueue \cdot wrapTree \cdot foldr\ tmix\ Nothing)\ [n-1, n-2\,..\,0]$$

where

$$tmix\ n\ mxt \ = \ mxt \odot Just\ (Node\ n\ empty) \odot mxt$$

The *step* function is unchanged, as shown in Figure 9.5 at the end of this chapter.

9.3. Gray codes using cyclic lists

We can condense the loopless definition of *gray* further by adopting cyclic lists instead of queues. We follow the idea of our loopless functional algorithm for *mixall* using binary trees from Chapter 8 by maintaining pairs of values in the labels of spines. These are represented by undecorated rose trees:

$$\textbf{data}\ Rose\ a \ = \ Node\ a\ [Rose\ a]$$

As our original algorithm for Gray codes involved only singleton lists, we know that they are of odd length and unaffected by reversal. Therefore, the bottom row of spines *br* will be the same as the top row *tr* and we maintain just one of these in a cyclic list. Hence, *prolog* takes the shorter form

$$\begin{array}{lcl} prolog\ n & = & consPair\ (length\ tr, tr)\ [\,] \\ & & \textbf{where}\ tr \ = \ [Node\ (m, x)\ tr \mid (m, x) \leftarrow ns] \\ & & \quad\ ns \ = \ zip\ [0\,..\,]\ [0\,..\,n-1] \end{array}$$

[3] Bird (2005b).

(a) Balanced

$000\bar{0}$	$011\bar{0}$	$1\bar{0}01$	$11\bar{0}0$
$00\bar{0}1$	$\bar{0}111$	$\bar{1}101$	$1\bar{1}10$
$001\bar{1}$	$1\bar{1}11$	$010\bar{1}$	$10\bar{1}0$
$0\bar{0}10$	$10\bar{1}1$	$\bar{0}100$	$\bar{1}000$

(b) Complementary

$000\bar{0}$	$01\bar{1}0$	$111\bar{1}$	$10\bar{0}1$
$00\bar{0}1$	$010\bar{0}$	$11\bar{1}0$	$101\bar{1}$
$001\bar{1}$	$01\bar{0}1$	$110\bar{0}$	$10\bar{1}0$
$0\bar{0}10$	$\bar{0}111$	$1\bar{1}01$	$\bar{1}000$

(c) Maximum gap

$\bar{0}0000$	$1111\bar{0}$	$\bar{1}0001$	$01\bar{0}10$	$\bar{0}0101$	$1101\bar{1}$	$\bar{1}0100$	$01\bar{1}11$
$1\bar{0}000$	$1\bar{1}111$	$0\bar{0}001$	$0\bar{1}110$	$1\bar{0}101$	$1\bar{1}010$	$0\bar{0}100$	$0\bar{1}011$
$11\bar{0}00$	$10\bar{1}11$	$0100\bar{1}$	$0011\bar{0}$	$11\bar{1}01$	$10\bar{0}10$	$0110\bar{0}$	$0001\bar{1}$
$111\bar{0}0$	$100\bar{1}1$	$010\bar{0}0$	$001\bar{1}1$	$110\bar{0}1$	$101\bar{1}0$	$011\bar{0}1$	$000\bar{1}0$

Figure 9.3. Constrained binary Gray codes.

while *step* is modified to

$$\begin{aligned} &\mathit{step}\ [\,] &&= \mathit{Nothing} \\ &\mathit{step}\ ((m, \mathit{Node}\ (n, x)\ tr : ts) : nts) &&= \mathit{Just}\ (x, \mathit{consPair}\ (n, tr)\ (\mathit{consPair}\ (m-1, ts)\ nts)) \end{aligned}$$

where

$$\mathit{consPair}\ (n, ts)\ nts \quad = \quad \begin{array}{l} \textbf{if}\ n == 0\ \textbf{then}\ nts \\ \textbf{else}\ (n, ts) : nts \end{array}$$

9.4. Constrained binary Gray codes

We may wish to generate binary Gray codes that have additional properties above their simple definition. For example, in some applications it is desirable for the number of transitions be more evenly distributed throughout the bit positions than in the binary reflected Gray code. In this arrangement, the lowest order bit changes 2^{n-1} times, while the highest order bit changes only twice including the return to the first element. As listed in Figure 9.3(a) and illustrated in Figure 9.4(a), it is possible to generate a balanced binary Gray code in which an equal number of transitions occur at each bit position for all values of n equal to 2^k where $k \geq 1$.[4]

Alternatively, for even values of n, it is possible to produce a complementary binary Gray code in which when represented in a circular way, as shown in Figure 9.4(b), every string $a_{n-1} \ldots a_1 a_0$ is diametrically opposite its complement $\bar{a}_{n-1} \ldots \bar{a}_1 \bar{a}_0$. The case $n = 4$ is shown in Figure 9.3(b).

We can also aim to maximize the 'gap' in a Gray code. This is the shortest maximal consecutive sequence of 0 or 1 bits amongst all bit positions. Exact values for the length of the maximal gap are unknown for $n \geq 8$, but the table below shows values found by exhaustive computer searches:

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	2	2	2	4	4	5	5	6	8	8	8	9	9	11	11

The case $n = 5$ appears in Figure 9.3(c).

[4] Savage (1997) 608–609 and Goddyn & Gvozdjak (2003).

(a) Balanced (b) Complementary

Figure 9.4. Graphical representations of constrained binary Gray codes.

9.5. Non-binary Gray codes

It is also possible to construct Gray code sequences for more general n-tuples $(a_{n-1}, \ldots a_1, a_0)$ where $0 \le a_j < m_j$. The radices m_j may be different for each coordinate, but when $0 \le a_j < 10$ for all j, we have decimal digits. A suitable sequence would then be the reflected decimal Gray code in which each coordinate moves alternatively from 0 up to 9 and then back down from 9 to 0. For example, we have the following sequence for $n = 3$:

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, \ldots, 091, 090, 190, 191, \ldots, 900.$$

In general our Gray code is defined such that in each step only one coordinate changes, and only by ± 1. We can then generalize Algorithms A and B as follows:

ALGORITHM C (Loopless reflected mixed-radix Gray generation). This loopless algorithm generates all n-tuples $(a_{n-1}, \ldots, a_1, a_0)$ such that $0 \le a_j < m_j$ for $0 \le j < n$, starting with $(0, \ldots, 0, 0)$ and changing exactly one coordinate by ± 1 in each step. We use an array of focus pointers $(f_n, \ldots, f_1, f_0)$ and an array of directions $(o_{n-1}, \ldots, o_1, o_0)$. Also each radix $m_j \ge 2$. [5]

C1. (Initialize) Set $a_j \leftarrow 0$, $f_j \leftarrow j$, and $o_j \leftarrow 1$ for $0 \le j < n$. Also set $f_n \leftarrow n$.

C2. (Visit) Visit the n-tuple $(a_{n-1}, \ldots, a_1, a_0)$.

C3. (Choose j) Set $j \leftarrow f_0$ and $f_0 \leftarrow 0$.

C4. (Change coordinate j) Terminate if $j = n$. Otherwise set $a_j \leftarrow a_j + o_j$.

C5. (Reflect?) If $a_j = 0$ or $a_j = m_j - 1$, set $o_j \leftarrow -o_j$, $f_j \leftarrow f_{j+1}$ and $f_{j+1} \leftarrow j + 1$. Return to C2.

An alternative to the reflected decimal Gray code is the modular decimal Gray code in which digits increase by 1 mod 10, wrapping around from 9 to 0 as follows in the case $n = 3$:

$$000, 001, \ldots, 009, 019, 010, \ldots, 017, 018, 028, 029, \ldots, 099, 090, 190, 191, \ldots, 900.$$

This may be generated by a method similar to Algorithm C.

[5] Knuth (2005a) 20.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Rose}\ a & = & \mathit{Node}\ a\ (\mathit{Queue}\ (\mathit{Rose}\ a)) \\
\\
\mathit{gray} & = & \mathit{unfoldr}\ \mathit{step} \cdot \mathit{prolog} \\
\\
\mathit{prolog}\ n & = & (\mathit{wrapQueue} \cdot \mathit{wrapTree} \cdot \mathit{foldr}\ \mathit{tmix}\ \mathit{Nothing})\ \mathit{ns} \\
 & & \textbf{where}\ \mathit{ns} \quad = \quad [n-1, n-2\,..\,0] \\
\\
\mathit{tmix}\ n\ \mathit{mxt} & = & \mathit{mxt} \odot \mathit{Just}\ (\mathit{Node}\ n\ \mathit{empty}) \odot \mathit{mxt} \\
\\
\mathit{Nothing} \odot \mathit{myt} & = & \mathit{myt} \\
\mathit{mxt} \odot \mathit{Nothing} & = & \mathit{mxt} \\
\mathit{Just}\ (\mathit{Node}\ x\ \mathit{xtq}) \odot \mathit{Just}\ \mathit{yt} & = & \mathit{Just}\ (\mathit{Node}\ x\ (\mathit{insert}\ \mathit{xtq}\ \mathit{yt})) \\
\\
\mathit{wrapTree}\ \mathit{Nothing} & = & \mathit{empty} \\
\mathit{wrapTree}\ (\mathit{Just}\ \mathit{xt}) & = & \mathit{insert}\ \mathit{empty}\ \mathit{xt} \\
\\
\mathit{wrapQueue}\ \mathit{xtq} & = & \mathit{consQueue}\ \mathit{xtq}\ [\,] \\
\\
\mathit{consQueue}\ \mathit{xtq}\ \mathit{xtqs} & = & \textbf{if}\ \mathit{isEmpty}\ \mathit{xtq}\ \textbf{then}\ \mathit{xtqs} \\
 & & \textbf{else}\ \mathit{xtq} : \mathit{xtqs} \\
\\
\mathit{step}\ [\,] & = & \mathit{Nothing} \\
\mathit{step}\ (\mathit{xtq} : \mathit{xtqs}) & = & \mathit{Just}\ (x, \mathit{consQueue}\ \mathit{ytq}\ (\mathit{consQueue}\ \mathit{ztq}\ \mathit{xtqs})) \\
 & & \textbf{where}\ (\mathit{Node}\ x\ \mathit{ytq}, \mathit{ztq}) \quad = \quad \mathit{remove}\ \mathit{xtq}
\end{array}
$$

Figure 9.5. A loopless functional algorithm for gray using rose trees.

Chapter 10

Generating Ideals of a Forest Poset

> *Both the man of science and the man of art live always at the edge of mystery, surrounded by it. Both, as a measure of their creation, have always had to do with the harmonization of what is new with what is familiar, with the balance between novelty and synthesis, with the struggle to make partial order in total chaos... This cannot be an easy life.*
>
> J. Robert Oppenheimer, *Prospects in the Arts and Sciences, 1954*

A 'forest poset' is a set of trees together with a partial order. We can generalize the imperative algorithm for non-binary Gray codes in such a way that we can generate the ideals of a poset in an order analogous to the reflected Gray code.

10.1. The Koda-Ruskey algorithm

The order we choose corresponds to a representation of ideals gained by recolouring nodes. More precisely,

(i) Each node is coloured either white or black.

(ii) Two consecutive colourings must differ in the colour assigned to exactly one node.

(iii) If a node j is coloured white and k is a child of j, then k cannot be coloured black.

The ideals will then correspond to the nodes coloured black. It follows that our ordering will be such that exactly one node is recoloured in each step. We can see this in Figure 10.1.

Representing white nodes by a 0 bit and black nodes by a 1 bit, we also have a representation in terms of binary n-tuples $(a_1, a_2, \ldots, a_n)$ where the parity of a_j corresponds to the colouring of the node labelled j. If $a_j = 1$, then j is a node in the current ideal. This leads us to formulate the imperative Koda-Ruskey algorithm:

ALGORITHM D (Loopless reflected subforest generation). This loopless algorithm generates all [1]
binary n-tuples $(a_1, a_2, \ldots, a_n)$ such that $a_p \geq a_q$ whenever p is a parent of q in a forest whose nodes are $(1, \ldots, n)$ when arranged in postorder. It starts with $(0, \ldots, 0, 0)$ and changes exactly one bit in each step. We use an array of focus pointers $(f_0, f_1 \ldots, f_n)$ and two arrays of pointers $(l_0, l_1, \ldots, l_n)$ and $(r_0, r_1, \ldots, r_n)$ to represent a doubly linked list. This contains all nodes of a subforest and their children, with r_0 pointing to its leftmost node and l_0 its rightmost.

The forest is described by an array $(c_0, c_1, \ldots, c_n)$. If p has no children, then $c_p = 0$. Otherwise c_p is the leftmost child of p. Also c_0 is the leftmost root of this forest. When the algorithm is

[1] Knuth (2005a) 20–21, based upon Koda & Ruskey (1993) 337–339.

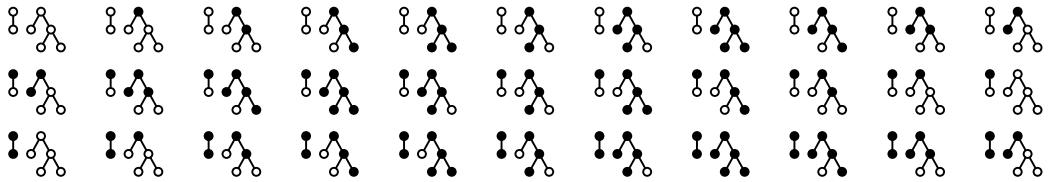


Figure 10.1. Ideals of a forest poset.

initialized, the doubly linked list has values $r_p = q$ and $l_q = p$ whenever p and q are consecutive children of the same family.

To illustrate this construction, consider the forest in Figure 10.2, with nodes arranged in postorder. It follows that $(c_0, \ldots, c_7) = (2, 0, 1, 0, 0, 0, 4, 3)$, and $r_2 = 7$, $l_7 = 2$, $r_3 = 6$, $l_6 = 3$, $r_4 = 5$ and $l_5 = 2$.

D1. (Initialize) Set $a_j \leftarrow 0$ and $f_j \leftarrow j$ for $1 \le j < n$. Also set $f_0 \leftarrow 0$, $l_0 \leftarrow n$, $r_n \leftarrow 0$, $r_0 \leftarrow c_0$ and $l_{c_0} \leftarrow 0$.

D2. (Visit) Visit the subforest defined by $(a_1, a_2, \ldots, a_n)$.

D3. (Choose p) Set $q \leftarrow l_0$ and $p \leftarrow f_q$. Also set $f_q \leftarrow q$.

D4. (Check a_p) Terminate if $p = 0$. Otherwise, if $a_p = 1$, go to D6.

D5. (Insert p's children) Set $a_p \leftarrow 1$. Then, if $c_p \neq 0$, set $q \leftarrow r_p$, $l_q \leftarrow p-1$, $r_{p-1} \leftarrow q$, $r_p \leftarrow c_p$ and $l_{c_p} \leftarrow p$. Go to D7.

D6. (Remove p's children) Set $a_p \leftarrow 0$. Then, if $c_p \neq 0$, set $q \leftarrow r_{p-1}$, $r_p \leftarrow q$ and $l_q \leftarrow p$.

D7. (Make p passive) Set $f_p \leftarrow f_{l_p}$ and $f_{l_p} \leftarrow l_p$. Return to D2.

The basic idea of this algorithm is to interleave the sequences of colourings of each tree in the forest. We can see in the first line of Figure 10.2 the first colouring of the left tree combined in turn with each of the colourings of the right tree. Similarly, the second line shows the second colouring of the left tree, but this time with the colourings of the of the right tree in reverse order. Finally, the third line consists of the third colouring of the left tree with all the colourings of the right tree in their original order.

We can illustrate the connection with loopless reflected mixed-radix Gray generation by considering the forest of degenerate non-branching trees in Figure 10.3. This has $3 \times 2 \times 4 \times 2$ ideals which correspond to the 4-tuples (x_1, x_2, x_3, x_4) where x_j is the number of nodes coloured black in the jth tree. When our algorithm is applied to this forest, it visits the ideals in the same order as the reflected mixed-radix Gray code on radices $(3, 2, 4, 2)$.

10.2. Mixing and ideals of a forest poset

We define the colouring of a forest of rose trees with distinct integer labels using a function *koda* based upon *mixall* and the prelude function *map*: [2]

$$
\begin{array}{lcl}
\mathit{koda} & :: & [\mathit{Rose}\ a] \to [a] \\
\mathit{koda} & = & \mathit{mixall} \cdot \mathit{map}\ \mathit{ruskey} \\
 & & \textbf{where}\ \mathit{ruskey}\ (\mathit{Node}\ x\ ts) \quad = \quad x : \mathit{koda}\ ts
\end{array}
$$

[2] Bird (2005b). A functional algorithm using continuations which generates the trees themselves appears in Filliâtre & Pottier (2003) 947–949.

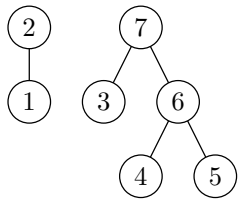


Figure 10.2. A forest poset.

We begin with each node coloured white and our transitions are to recolour the node with the specified label. We then interleave the colours of the subtrees using *mixall*. The root of each tree is coloured black and then the colourings of the subtrees of this tree are also interleaved. It follows that for the forest in Figure 10.2 the transitions between colourings, and therefore ideals of the forest poset, are

$$7, 6, 5, 4, 5, 3, 5, 4, 5, 6, 2, 6, 5, 4, 5, 3, 5, 4, 5, 6, 7, 1, 7, 6, 5, 4, 5, 3, 5, 4, 5, 6.$$

To obtain a loopless functional algorithm for *koda*, we modify our loopless version of *mixall* using rose trees from Chapter 5. During that derivation we defined *mixall* by

$$mixall \quad = \quad fst \cdot foldr\ pmix\ ([\,], [\,])$$

It follows that we can use fold-map fusion to fuse *mixall* in this form to *map ruskey*. Hence,

$$koda \quad = \quad fst \cdot foldr\ (pmix \cdot ruskey)\ ([\,], [\,])$$

Moreover, by the definition of *ruskey*, we have

$$pmix\ (ruskey\ (Node\ x\ ts)) \quad = \quad pmix\ (x : koda\ ts)\ ([\,], [\,])$$

Now, by the definition of *pmix*, we obtain the following:

$$pmix\ (ruskey\ (Node\ x\ ts))\ (ys, sy) \quad = \quad (ys, sy) \oslash ([x], [x]) \oslash (pmix\ (koda\ ts)\ (sy, ys))$$

We would like to express $pmix\ (koda\ ts)$ in terms of *foldr*. Using the definitions of *koda* and *mixall*, we have

$$pmix\ (koda\ (t : ts)) \quad = \quad pmix\ (mix\ (ruskey\ t)\ (koda\ ts))$$

By the associativity of *mix* and *xim*, we can use the identity

$$pmix\ (mix\ xs\ ys) \quad = \quad pmix\ xs \cdot pmix\ ys$$

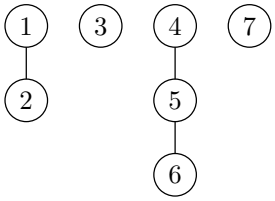


Figure 10.3. A forest poset of degenerate non-branching trees.

to obtain a recursive definition of $pmix \cdot koda$:

$$\begin{array}{lcl} pmix\ (koda\ [\,]) & = & id \\ pmix\ (koda\ (t : ts)) & = & pmix\ (ruskey\ t) \cdot pmix\ (koda\ ts) \end{array}$$

It follows that

$$pmix\ (koda\ ts)\ (sy, ys) \quad = \quad foldr\ (pmix \cdot ruskey)\ (sy, ty)\ ts$$

by the definition of *foldr*. If we define a function *qmix* by

$$qmix \quad = \quad pmix \cdot ruskey$$

then

$$koda \quad = \quad fst \cdot foldr\ qmix\ ([\,], [\,])$$

Hence,

$$qmix\ (Node\ x\ ts)\ (ys, sy) \quad = \quad (ys, sy) \oslash ([x], [x]) \oslash (foldr\ qmix\ (sy, ys)\ ts)$$

In common with our derivation of *mixall*, we introduce another function, *kmix*, satisfying

$$pair\ preorder \cdot kmix\ t \quad = \quad qmix\ t \cdot pair\ preorder$$

This may be defined by

$$\begin{array}{lcl} kmix\ (Node\ x\ ts)\ (myt, mty) & = & (myt, mty) \otimes (mxt, mxt) \otimes (foldr\ kmix\ (mty, mty)\ ts) \\ & & \textbf{where}\ mxt \quad = \quad Just\ (Node\ x\ [\,]) \end{array}$$

We are again using roses trees along with the type *Maybe*. Therefore, we have the loopless form

$$koda \quad = \quad unfoldr\ step \cdot prolog$$

where

$$prolog \quad = \quad wrapList \cdot wrapTree \cdot fst \cdot foldr\ kmix\ (Nothing, Nothing)$$

The loopless functional algorithm for *koda* in Figure 10.4, on the next page, follows by the conversion of lists to real-time queues. We also need to retain our original definition of rose trees using lists since the input rose tree will be represented in this way. [3]

[3] We could avoid having two types of rose tree by formulating our input using queues. However, we would then need to replace *foldr* by a fold for queues *foldQueue* defined by

$$\begin{array}{lcl} foldQueue & :: & (\mathrm{a} \to b \to b) \to b \to Queue\ \mathrm{a} \to b \\ foldQueue\ f\ e\ xq & = & \textbf{if}\ isEmpty\ xq\ \textbf{then}\ e \\ & & \textbf{else}\ f\ y\ (foldQueue\ yq) \\ & & \textbf{where}\ (y, yq) \quad = \quad remove\ xq \end{array}$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Rose}\ a & = & \mathit{Node}\ a\ [\mathit{Rose}\ a] \\
\textbf{data}\ \mathit{Rose}'\ a & = & \mathit{Node}'\ a\ (\mathit{Queue}\ (\mathit{Rose}'\ a)) \\
\\
\mathit{koda} & = & \mathit{unfoldr}\ \mathit{step} \cdot \mathit{prolog} \\
\\
\mathit{prolog} & = & \mathit{wrapQueue} \cdot \mathit{wrapTree} \cdot \mathit{fst} \cdot \mathit{foldr}\ \mathit{kmix}\ (\mathit{Nothing}, \mathit{Nothing}) \\
\\
\mathit{kmix}\ (\mathit{Node}\ x\ ts)\ (myt, mty) & & \\
 & = & (myt, mty) \otimes (mxt, mxt) \otimes (\mathit{foldr}\ \mathit{kmix}\ (mty, mty)\ ts) \\
 & & \textbf{where}\ mxt\ =\ \mathit{Just}\ (\mathit{Node}'\ x\ \mathit{empty}) \\
\\
(mxt, mtx) \otimes (myt, mty) & = & (mxt \odot myt, mty \odot mtx) \\
\\
\mathit{Nothing} \odot myt & = & myt \\
mxt \odot \mathit{Nothing} & = & mxt \\
\mathit{Just}\ (\mathit{Node}'\ x\ xtq) \odot \mathit{Just}\ yt & & \\
 & = & \mathit{Just}\ (\mathit{Node}'\ x\ (\mathit{insert}\ xtq\ yt)) \\
\\
\mathit{wrapTree}\ \mathit{Nothing} & = & \mathit{empty} \\
\mathit{wrapTree}\ (\mathit{Just}\ xt) & = & \mathit{insert}\ \mathit{empty}\ xt \\
\\
\mathit{wrapQueue}\ xtq & = & \mathit{consQueue}\ xtq\ [\,] \\
\\
\mathit{consQueue}\ xtq\ xtqs & = & \textbf{if}\ \mathit{isEmpty}\ xtq\ \textbf{then}\ xtqs \\
 & & \textbf{else}\ xtq : xtqs \\
\\
\mathit{step}\ [\,] & = & \mathit{Nothing} \\
\mathit{step}\ (xtq : xtqs) & = & \mathit{Just}\ (\mathrm{x}, \mathit{consQueue}\ ytq\ (\mathit{consQueue}\ ztq\ xtqs)) \\
 & & \textbf{where}\ (\mathit{Node}'\ x\ ytq, ztq)\ =\ \mathit{remove}\ xtq
\end{array}
$$

Figure 10.4. A loopless functional algorithm for koda using rose trees.

Chapter 11

Generating Ideals of an Acyclic Poset

> *Ideals are like stars; you will not succeed in touching them with your hands. But like the seafaring man on the desert of waters, you choose them as your guides, and following them you will reach your destiny.*
>
> Carl Schurz, *Speech, 1859*

We now consider a variation on the generation of ideals of a forest poset in which the edges in each tree are directed. Therefore, given a digraph that is totally acyclic, even if the direction of the edges is ignored, we construct an algorithm that generates its ideals.

11.1. The Li-Ruskey algorithm

In common with the Koda-Ruskey algorithm, we recolour nodes to generate a sequence of distinct colourings represented by bit strings. Then the ideals of our acyclic poset will correspond to the nodes coloured black. This will be subject to the following constraints:

(i) Each node is coloured either white or black.

(ii) Two consecutive colourings must differ in the colour assigned to exactly one node.

(iii) If a node i is coloured white and $i \to j$ is an edge directed downwards, then j cannot be coloured black.

Clearly, we will be able to describe the transitions between one colouring and the next by naming a single node. Moreover, all children of a white node must themselves be white. This is shown by the ideals of Figure 11.1 in which all directions point downwards. If a white node corresponds to a 0 bit and a black node to 1 bit, we have a formulation of the problem in terms of binary n-tuples $(a_1, a_2, \ldots, a_n)$:

(i) Whenever $i \leftarrow j$ is an edge in a total acyclic digraph, then $a_i \leq a_j$.

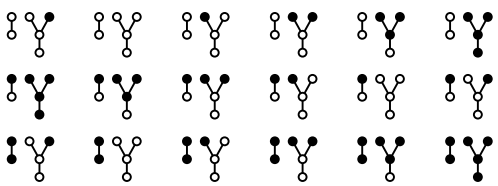


Figure 11.1. Ideals of an acyclic poset.

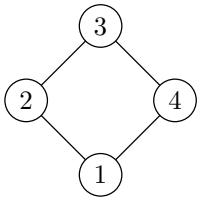


Figure 11.2. A cyclic digraph.

(ii) Exactly one bit is changed in each step.

Therefore, our n-tuples are in Gray code order. The second restriction explains why we only consider totally acyclic posets. The digraph in Figure 11.2 has ideals represented by

$$0000 \quad 0001 \quad 0011 \quad 0101 \quad 0111 \quad 1111$$

However, to order these such that exactly one bit changes at a time we need to be able to alternate between n-tuples of odd and even parity, and in this case we have two odd and four even n-tuples.

We can also see that we will not necessarily be able to begin with every node coloured white. For example the ideals in Figure 11.1 have binary representation

$$\begin{array}{lll} 0000\bar{1}0 & 10111\bar{1} & 1100\bar{1}0 \\ 000\bar{0}00 & 10\bar{1}110 & 110\bar{0}00 \\ 0001\bar{0}0 & 1001\bar{1}0 & 1101\bar{0}0 \\ 00\bar{0}110 & 100\bar{1}00 & 11\bar{0}110 \\ 00111\bar{0} & 1000\bar{0}0 & 11111\bar{0} \\ \bar{0}01111 & 1\bar{0}0010 & 111111 \end{array}$$

where the nodes are labelled in preorder as in Figure 11.3.

An exception occurs when we have a digraph consisting of a forest of directed trees in which all edges are directed downwards. It follows that we have a forest of rose trees and we can apply the Koda-Ruskey algorithm in which the starting configuration is always $(0, 0, \ldots, 0)$. Our general algorithm is the Li-Ruskey algorithm, formulated imperatively using coroutines. [1]

11.2. Mixing and ideals of an acyclic poset

Functionally, we can represent totally acyclic digraphs by a forest of directed trees of type

$$\textbf{data}\ \mathit{Dtree} \quad = \quad \mathit{Node}\ \mathit{Int}\ [(\mathit{Direction}, \mathit{Dtree})]$$

where

$$\textbf{data}\ \mathit{Direction} \quad = \quad \mathit{Down} \mid \mathit{Up}$$

For example, Figure 11.3 would be represented by the forest

$$[\mathit{Node}\ 1\ [(\mathit{Down}, \mathit{Node}\ 2\ [\,])], \mathit{Node}\ 3\ [\mathit{Up}, \mathit{Node}\ 4\ [\,]), (\mathit{Up}, \mathit{Node}\ 5\ [\,]), (\mathit{Down}, \mathit{Node}\ 6\ [\,])]$$

[1] Knuth & Ruskey (2004).

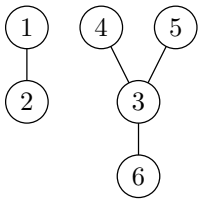


Figure 11.3. An acyclic digraph with two connected components.

In common with the function *koda* in Section 10.2, which generated the transitions between ideals of a forest poset, we can define a function *trans* in terms of *mixall* and the prelude function *map*:

$$\begin{array}{lcl} trans & :: & [Dtree] \to [Int] \\ trans & = & mixall \cdot map\ treeTrans \end{array}$$

The transitions again represent the recolouring of the node with the specified label, though in this case, as explained above, we do not necessarily begin with all nodes coloured white. The transitions between the colours of Figure 11.3 are

$$5, 4, 5, 3, 6, 1, 6, 3, 5, 4, 5, 2, 5, 4, 5, 3, 6.$$

Initially, all nodes are white except node 5 which must be coloured black.

We now need to derive a function *treeTrans* with type signature

$$treeTrans \quad :: \quad Dtree \to [Dtree]$$

This will generate the transitions for a single directed tree in the forest. To help us define *treeTrans*, we split the transition sequence into three parts:

(i) A white sequence *ws* in which the root node is coloured white.

(ii) A black sequence *bs* in which the root node is coloured black.

(iii) A Gray sequence *gs* which combines *ws* and *bs* by running through the white sequence, recolouring the root to black and then running through the black sequence.

Clearly using this formulation the root of each directed tree in the forest will start with the root coloured white. It is also easy to see that

$$gs \quad = \quad ws +\!\!+ [n] +\!\!+ bs$$

The function *mkTrans n* will construct the Gray transitions for a directed tree with root labelled *n*:

$$mkGtrans\ n\ (ws, bs) \quad = \quad (ws, bs, ws +\!\!+ [n] +\!\!+ bs)$$

Now, suppose we have a directed tree

$$u = Node\ n\ ((d, t) : dts)$$

and denote the white and black sequences associated with *dts* by (wy, bs), and the white, black and Gray sequences associated with *t* by (wx, bx, gx). If *d* is *Down*, then because the bit representing the root of *t* has to be 0, the white sequence of *u* must be an interleaving of *wx*

with wy, or possibly with one or both lists reversed. By similar reasoning, the black sequences of u must be some interleaving of gx with by because the bit representing the root of t could be either 0 or 1. In the case that d is Up, the white sequence of u will be some interleaving of gx with wy and the black sequence some interleaving of bx and by. If we can calculate these interleavings, then our definition of *treeTrans* will follow.

We introduce the notation

$$gx \mid 0 : ix \quad \rightsquigarrow \quad 1 : ex$$

to describe a Gray sequence beginning with a sequence $0 : ix$ and ending in a sequence $1 : ex$. This means that the bit representing root of t will be 0 and it will be coloured white.

We know that a Gray sequence is a white sequence followed by a recolouring of the root and then followed by a black sequence, so suppose we have intermediate configurations mx and my. It follows that

$$wx \mid 0 : ix \quad \rightsquigarrow \quad 0 : mx \qquad \text{and} \qquad bx \mid 1 : mx \quad \rightsquigarrow \quad 1 : ex$$

and the configurations of the subtrees dts are

$$wy \mid iy \quad \rightsquigarrow \quad my \qquad \text{and} \qquad by \mid my \quad \rightsquigarrow \quad ey$$

We need a function to interleave our sequences, so it would seem sensible to try to use mix. However, when d is $Down$, we cannot choose $mix\ wx\ wy$ for the white sequence of u because

$$mix\ wx\ wy \mid \begin{array}{llll} 0 : ix +\!\!+ iy & \rightsquigarrow & 0 : mx +\!\!+ my & \textbf{if } even\ (length\ wx) \\ 0 : ix +\!\!+ iy & \rightsquigarrow & 0 : mx +\!\!+ iy & \textbf{if } odd\ (length\ wx) \end{array}$$

and we would be unable to construct an appropriate black sequence to follow since neither the beginning nor end of gx involves mx. Similarly, because iy does not appear at the beginning or end of by, we cannot define the sequence by

$$mix\ (reverse\ wx)\ wy \mid \begin{array}{llll} 0 : mx +\!\!+ iy & \rightsquigarrow & 0 : ix +\!\!+ my & \textbf{if } even\ (length\ wx) \\ 0 : mx +\!\!+ iy & \rightsquigarrow & 0 : ix +\!\!+ iy & \textbf{if } odd\ (length\ wx) \end{array}$$

For identical reasons, it is also not possible to use

$$mix\ (reverse\ wx)\ (reverse\ wy) \mid \begin{array}{llll} 0 : mx +\!\!+ my & \rightsquigarrow & 0 : ix +\!\!+ iy & \textbf{if } even\ (length\ wx) \\ 0 : mx +\!\!+ my & \rightsquigarrow & 0 : ix +\!\!+ my & \textbf{if } odd\ (length\ wx) \end{array}$$

Therefore, we define mex, a variation of mix by

$$mex\ wx\ wy \quad = \quad reverse\ (mix\ wx\ (reverse\ wy))$$

It follows that

$$mex\ wx\ wy \mid \begin{array}{llll} 0 : mx +\!\!+ iy & \rightsquigarrow & 0 : ix +\!\!+ my & \textbf{if } even\ (length\ wx) \\ 0 : mx +\!\!+ my & \rightsquigarrow & 0 : ix +\!\!+ my & \textbf{if } odd\ (length\ wx) \end{array}$$

Clearly, since the beginning of gx involves ix and the beginning of by involves my, this is an appropriate interleaving. It is then easy to define the black sequence for u:

$$mix\ gx\ by \mid \begin{array}{llll} 0 : ix +\!\!+ my & \rightsquigarrow & 1 : ex +\!\!+ ey & \textbf{if } even\ (length\ gx) \\ 0 : ix +\!\!+ my & \rightsquigarrow & 1 : ex +\!\!+ my & \textbf{if } odd\ (length\ gx) \end{array}$$

We can also apply *mex* to construct the white sequence of u when d is Up. Then we have

$$mex\ (reverse\ gx)\ wy \left| \begin{array}{lcll} 0 : ix +\!\!+ iy & \rightsquigarrow & 1 : ex +\!\!+ my & \textbf{if}\ even\ (length\ gx) \\ 0 : ix +\!\!+ my & \rightsquigarrow & 1 : ex +\!\!+ my & \textbf{if}\ odd\ (length\ gx) \end{array} \right.$$

The matching black sequence for u is given by

$$mix\ (reverse\ bx)\ by \left| \begin{array}{lcll} 1 : ex +\!\!+ my & \rightsquigarrow & 1 : mx +\!\!+ ey & \textbf{if}\ even\ (length\ bx) \\ 1 : ex +\!\!+ my & \rightsquigarrow & 1 : mx +\!\!+ my & \textbf{if}\ odd\ (length\ bx) \end{array} \right.$$

It follows that the white and black sequences of u can be generated by a function *mkWBtrans*:

$$\begin{array}{lcl} mkWBtrans\ (Down, (wx, bx, gx))\ (wy, by) & = & (mex\ wx\ wy, mix\ gx\ by) \\ mkWBtrans\ (Up, (wx, bx, gx))\ (wy, by) & = & (mex\ (reverse\ gx)\ wy, mix\ (reverse\ bx)\ by) \end{array}$$

After combining these two sequences into a Gray sequence using *mkGtrans*, we can calculate the transitions of a directed tree using an appropriate fold function for directed trees defined by

$$\begin{array}{lcl} foldTree & :: & (Int \to [(Direction, \mathrm{a})] \to \mathrm{a}) \to Dtree \to a \\ foldTree\ f\ (Node\ n\ dts) & = & f\ n\ [(d, foldTree\ f\ t) \mid (d, t) \leftarrow dts] \end{array}$$

Hence, our function *treeTrans* is given by

$$treeTrans \quad = \quad gray \cdot foldTree\ mkWBGtrans$$

where

$$mkWBGtrans\ n \quad = \quad mkGtrans\ n \cdot foldr\ mkWBtrans\ ([\,], [\,])$$

$$gray\ (ws, bs, gs) \quad = \quad gs$$

11.3. Choosing the starting configuration

To find the starting configuration for a Gray sequence, we need a function *start* with type signature

$$start \quad :: \quad [Dtree] \to [Bit]$$

where it is sufficient to define the type *Bit* by

$$\textbf{type}\ Bit \quad = \quad Int$$

In a similar way to the generation of transitions, we can formulate *start* in terms of the starting configurations of each directed tree in the forest. Hence, we use the prelude functions *concat* and *map* to define

$$start \quad = \quad concat \cdot map\ treeStart$$

Now, to compute the necessary configurations, we need the parity of the white, black and Gray sequences. We introduce boolean values *elwx* and *elbx* to indicate whether *wx* and *bx*, respectively, are of even length. It clearly follows that the resulting Gray sequence *gx* will be of even length only if one constituent sequence is of odd length. Therefore,

$$elgx = (elwx \neq elbx)$$

These observations allow us to construct a function *mkWBconfigs* defined by

$$
\begin{array}{l}
mkWBconfigs\ (Down, (elwx, elbx, ix, mx, ex))\ (elwy, elby, iy, my, ey) \\
\quad \mid elwx \wedge elbx \quad = \quad (elwy, False, 0 : mx +\!\!+ iy, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my) \\
\quad \mid elwx \wedge \neg elbx \quad = \quad (elwy, elby, 0 : mx +\!\!+ iy, 0 : ix +\!\!+ my, 1 : ex +\!\!+ ey) \\
\quad \mid \neg elwx \wedge elbx \quad = \quad (False, elby, 0 : mx +\!\!+ my, 0 : ix +\!\!+ my, 1 : ex +\!\!+ ey) \\
\quad \mid \neg elwx \wedge \neg elbx \quad = \quad (False, False, 0 : mx +\!\!+ my, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my)
\end{array}
$$

$$
\begin{array}{l}
mkWBconfigs\ (Up, (elwx, elbx, ix, mx, ex))\ (elwy, elby, iy, my, ey) \\
\quad \mid elwx \wedge elbx \quad = \quad (False, elby, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ ey) \\
\quad \mid elwx \wedge \neg elbx \quad = \quad (elwy, False, 0 : ix +\!\!+ iy, 1 : ex +\!\!+ my, 1 : mx +\!\!+ my) \\
\quad \mid \neg elwx \wedge elbx \quad = \quad (elwy, elby, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ ey) \\
\quad \mid \neg elwx \wedge \neg elbx \quad = \quad (False, False, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ my)
\end{array}
$$

We then use *foldTree* once more to obtain

$$treeStart \quad = \quad third \cdot foldTree\ treeConfigs$$

where

$$
\begin{array}{lcl}
treeConfigs\ n & = & foldr\ mkWBconfigs\ (True, True, [\,], [\,], [\,]) \\
\\
third\ (elwy, elby, iy, my, ey) & = & 0 : iy
\end{array}
$$

The full listings of our functional algorithms for *trans* and *start* appear in Figure 11.4 on the page that follows.

$$
\begin{array}{lcl}
\textbf{data}\ Dtree & = & Node\ Int\ [(Direction, Dtree)] \\
\textbf{data}\ Direction & = & Down \mid Up \\
\textbf{type}\ Bit & = & Int \\
\\
trans & = & mixall \cdot map\ treeTrans \\
\\
treeTrans & = & gray \cdot foldTree\ mkWBGtrans \\
\\
mkWBGtrans\ n & = & mkGtrans\ n \cdot foldr\ mkWBtrans\ ([\,], [\,]) \\
\\
mkWBtrans\ (Down, (wx, bx, gx))\ (wy, by) & & \\
\qquad = & & (mex\ wx\ wy, mix\ gx\ by) \\
mkWBtrans\ (Up, (wx, bx, gx))\ (wy, by) & & \\
\qquad = & & (mex\ (reverse\ gx)\ wy, mix\ (reverse\ bx)\ by) \\
\\
mex\ wx\ wy & = & reverse\ (mix\ wx\ (reverse\ wy)) \\
\\
mix\ [\,]\ ys & = & ys \\
mix\ (x : xs)\ ys & = & ys +\!\!+ x : mix\ xs\ (reverse\ ys) \\
\\
mixall & = & foldr\ mix\ [\,] \\
\\
mkGtrans\ n\ (ws, bs) & = & (ws, bs, ws +\!\!+ [n] +\!\!+ bs) \\
\\
foldTree\ f\ (Node\ n\ dts) & = & f\ n\ [(d, foldTree\ f\ t) \mid (d, t) \leftarrow dts] \\
\\
gray\ (ws, bs, gs) & = & gs \\
\\
start & = & concat \cdot map\ treeStart \\
\\
treeStart & = & third \cdot foldTree\ treeConfigs \\
\\
treeConfigs\ n & = & foldr\ mkWBconfigs\ (True, True, [\,], [\,], [\,]) \\
\\
third\ (elwy, elby, iy, my, ey) & = & 0 : iy \\
\\
mkWBconfigs\ (Down, (elwx, elbx, ix, mx, ex))\ (elwy, elby, iy, my, ey) & & \\
\quad \mid\ elwx \wedge elbx & = & (elwy, False, 0 : mx +\!\!+ iy, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my) \\
\quad \mid\ elwx \wedge \neg elbx & = & (elwy, elby, 0 : mx +\!\!+ iy, 0 : ix +\!\!+ my, 1 : ex +\!\!+ ey) \\
\quad \mid\ \neg elwx \wedge elbx & = & (False, elby, 0 : mx +\!\!+ my, 0 : ix +\!\!+ my, 1 : ex +\!\!+ ey) \\
\quad \mid\ \neg elwx \wedge \neg elbx & = & (False, False, 0 : mx +\!\!+ my, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my) \\
mkWBconfigs\ (Up, (elwx, elbx, ix, mx, ex))\ (elwy, elby, iy, my, ey) & & \\
\quad \mid\ elwx \wedge elbx & = & (False, elby, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ ey) \\
\quad \mid\ elwx \wedge \neg elbx & = & (elwy, False, 0 : ix +\!\!+ iy, 1 : ex +\!\!+ my, 1 : mx +\!\!+ my) \\
\quad \mid\ \neg elwx \wedge elbx & = & (elwy, elby, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ ey) \\
\quad \mid\ \neg elwx \wedge \neg elbx & = & (False, False, 0 : ix +\!\!+ my, 1 : ex +\!\!+ my, 1 : mx +\!\!+ my)
\end{array}
$$

Figure 11.4. Functional algorithms for trans and start.

Chapter 12

Generating Permutations

> *Promises are the uniquely human way of ordering the future, making it predictable and reliable to the extent that this is humanly possible.*
>
> HANNAH ARENDT, *'Civil Disobedience', Crises of the Republic, 1972*

A permutation of a set is an arrangement of the elements in that set in some order. The inverse permutation of $a_1a_2\ldots a_n$ is a permutation $a'_1a'_2\ldots a'_n$ such that $a'_k = j$ if and only if $a_j = k$.

We consider a Gray code for permutations to be a sequence in which successive permutations differ only by the interchange of two adjacent elements, and if we interchange two elements in the last permutation, we return to the starting point. An indication that this is possible follows if we consider the bubble sort algorithm. Unsorted data can clearly be represented as a permutation and the bubble sort algorithm sorts this into some order, in effect another permutation, using adjacent interchanges.

12.1. Gray codes for permutations and the Johnson-Trotter algorithm

We can generate a Gray code for all permutations by adjacent interchanges using the recursive scheme illustrated in Figure 12.1. The element n is inserted into each position in the permutation of $n-1$, moving alternatively from right to left and then left to right. From this, we obtain a formulation known as 'plain changes', graphically represented in Figure 12.2.

An imperative algorithm for the generation of all permutations by plain changes is the Johnson-Trotter algorithm, as follows:

ALGORITHM E (Plain changes). This algorithm generates all permutations of a given sequence $a_1a_2\ldots a_n$ of n distinct elements, exchanging adjacent pairs of elements in each step. We use an array $c_1c_2\ldots c_n$ to represent pairs of not necessarily adjacent elements that are out of order, with $0 \le c_j < j$ for $1 \le j \le n$. An array of directions $o_1o_2\ldots o_n$ indicates how these c_j change. [1]

E1. (Initialize) Set $c_j \leftarrow 0$ and $o_j \leftarrow 1$ for $1 \le j \le n$.

E2. (Visit) Visit the permutation $a_1a_2\ldots a_n$.

E3. (Prepare for change) Set $j \leftarrow n$ and $s \leftarrow 0$.

E4. (Ready to change?) Set $q \leftarrow c_j + o_j$. If $q < 0$, go to E7. If $q = j$, go to E6.

E5. (Change) Exchange $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$. Then set $c_j \leftarrow q$ and return to E2.

E6. (Increase s) Terminate if $j = 1$. Otherwise set $s \leftarrow s+1$.

[1] Knuth (2005a) 40–44, based upon Johnson (1963) and Trotter (1962).

1	$\bar{1}\bar{2}$	$1\bar{2}\bar{3}$	$12\bar{3}\bar{4}$	$\bar{4}\bar{3}21$
	$\bar{2}\bar{1}$	$\bar{1}\bar{3}2$	$1\bar{2}\bar{4}3$	$3\bar{4}\bar{2}1$
		$3\bar{1}\bar{2}$	$\bar{1}\bar{4}23$	$32\bar{4}\bar{1}$
		$\bar{3}\bar{2}1$	$41\bar{2}\bar{3}$	$\bar{3}\bar{2}14$
		$2\bar{3}\bar{1}$	$\bar{4}\bar{1}32$	$23\bar{1}\bar{4}$
		$\bar{2}\bar{1}3$	$1\bar{4}\bar{3}2$	$2\bar{3}\bar{4}1$
			$13\bar{4}\bar{2}$	$\bar{2}\bar{4}31$
			$\bar{1}\bar{3}24$	$42\bar{3}\bar{1}$
			$31\bar{2}\bar{4}$	$\bar{4}\bar{2}13$
			$3\bar{1}\bar{4}2$	$24\bar{1}\bar{3}$
			$\bar{3}\bar{4}12$	$21\bar{4}\bar{3}$
			$43\bar{1}\bar{2}$	$\bar{2}\bar{1}34$

Figure 12.1. Plain changes.

E7. (Switch direction) Set $o_j \leftarrow -o_j$ and $j \leftarrow j-1$. Return to E4.

The sequence $c_1c_2\ldots c_n$ is known as the inversion of a permutation $a_1a_2\ldots a_n$. Every interchange of adjacent elements changes the number of elements that are out of order by ±1. For $n=4$, we have inversions as follows:

$000\bar{0}$	$001\bar{3}$	$002\bar{0}$	$012\bar{3}$	$01\bar{1}0$	$010\bar{3}$
$000\bar{1}$	$001\bar{2}$	$002\bar{1}$	$012\bar{2}$	$011\bar{1}$	$010\bar{2}$
$000\bar{2}$	$001\bar{1}$	$002\bar{2}$	$012\bar{1}$	$011\bar{2}$	$010\bar{1}$
$00\bar{0}3$	$00\bar{1}0$	$0\bar{0}23$	$012\bar{0}$	$01\bar{1}3$	$0\bar{1}00$

Recalling Section 9.5, this is clearly the reflected mixed-radix Gray code for radices $(1,2,3,4)$. Therefore, we can use the method employed by Algorithm C to generate $c_1c_2\ldots c_n$ looplessly and obtain a loopless imperative algorithm for plain changes.

We can avoid the calculation of the offset variable s in Algorithm E by keeping track of the inverse permutations $a'_1a'_2\ldots a'_n$ and letting the values of $c_1c_2\ldots c_n$ count only downwards. Then we have a more straightforward imperative algorithm:

ALGORITHM F (Plain changes with inverse permutations). This algorithm generates all permutations of a given sequence $a_1a_2\ldots a_n$ of n distinct elements and their corresponding inverse permutations $a'_1a'_2\ldots a'_n$ such that $a'_k=j$ if and only if $a_j=k$. We interchange adjacent pairs of elements in each step and use an array $c_1c_2\ldots c_n$ to represent pairs of not necessarily adjacent elements that are out of order, with $0\le c_j<j$ for $1\le j\le n$. An array of directions $o_1o_2\ldots o_n$ indicates how these c_j change. [2]

F1. (Initialize) Set $a_j\leftarrow a'_j\leftarrow j$, $c_j\leftarrow j-1$ and $o_j\leftarrow-1$ for $1\le j\le n$. Also set $c_0=-1$.

F2. (Visit) Visit the permutation $a_1a_2\ldots a_n$ and its inverse $a'_1\ldots a'_n$.

F3. (Find k) Set $k\leftarrow n$. Then, if $c_k=0$, set $c_k\leftarrow k-1$, $o_k\leftarrow-o_k$, $k\leftarrow k-1$ and repeat until $c_k\neq0$. Terminate if $k=0$.

F4. (Change) Set $c_k\leftarrow c_k-1$, $j\leftarrow a'_k$ and $i\leftarrow j+o_k$. Then set $t\leftarrow a_i$, $a_i\leftarrow k$, $a_j\leftarrow t$, $a'_t\leftarrow j$ and $a'_k\leftarrow i$. Return to F2.

[2] Knuth (2005a) 103–104, based upon Ehrlich (1973b) 505–506.

Figure 12.2. A graphical representation of plain changes.

It is also possible to generate these values of $c_1 c_2 \ldots c_n$ looplessly, therefore giving us another loopless imperative algorithm for permutation generation.

While loopless versions of Algorithms E and F guarantee that each successive permutation takes $O(1)$ time, in terms of total running time they are slower than the original algorithms containing loops. This is because the loops in these algorithms are used relatively infrequently, and the gain in efficiency that would normally result from their removal is outweighed by the overhead of maintaining additional arrays. In Algorithm E, for example, $n-1$ out of n permutations are generated without using the loop defined in steps E4, E6 and E7.

12.2. Mixing and permutations

We use *mix* to define a function *johnson* which expresses the transitions between permutations with adjacent pairs of elements interchanged: [3]

$$\begin{array}{lcl} \mathit{johnson} & :: & \mathit{Int} \to [\mathit{Int}] \\ \mathit{johnson}\ 1 & = & [\,] \\ \mathit{johnson}\ n & = & \mathit{mix}\ (\mathit{bump}\ 1\ (\mathit{johnson}\ (n-1)))\ [n-1\,..\,1] \end{array}$$

where

$$\begin{array}{lcl} \mathit{bump}\ k\ [\,] & = & [\,] \\ \mathit{bump}\ k\ [a] & = & [a+k] \\ \mathit{bump}\ k\ (a:b:ns) & = & (a+k) : b : \mathit{bump}\ k\ ns \end{array}$$

The function *bump* k adds k to every element in an even position in the sequence. We can define our transitions as the indices j of each element $a_0 a_1 \ldots a_{n-1}$ that should be interchanged with its adjacent element a_{j-1}. Therefore, as shown in Figure 12.1, the transitions between permutations of four elements are

$$3, 2, 1, 3, 1, 2, 3, 1, 3, 2, 1, 3, 1, 2, 3, 1, 3, 2, 1, 3, 1, 2, 3.$$

To express *johnson* in terms of *mixall* and take advantage the loopless functional algorithms we have derived, we generalize it to a function *code* such that

$$\mathit{johnson}\ n \quad = \quad \mathit{code}\ (0, n)$$

This is defined by

$$\mathit{code}\ (k, n) \quad = \quad \mathit{bump}\ k\ (\mathit{johnson}\ n)$$

Using the simple identities

$$\begin{array}{lcl} \mathit{bump}\ k\ (\mathit{mix}\ xs\ ys) & = & \textbf{if}\ \mathit{even}\ (\mathit{length}\ ys)\ \textbf{then}\ \mathit{mix}\ (\mathit{bump}\ k\ xs)\ (\mathit{bump}\ k\ ys) \\ & & \textbf{else}\ \mathit{mix}\ xs\ (\mathit{bump}\ k\ ys) \end{array}$$

$$\begin{array}{lcl} \mathit{bump}\ k\ (xs +\!\!+ y : ys) & = & \textbf{if}\ \mathit{even}\ (\mathit{length}\ xs)\ \textbf{then}\ \mathit{bump}\ k\ xs +\!\!+ \mathit{bump}\ k\ (y : ys) \\ & & \textbf{else}\ \mathit{bump}\ k\ xs +\!\!+ y : \mathit{bump}\ k\ ys \end{array}$$

[3] Bird (2005b).

provable by simple induction, we can derive a recursive definition of *code*:

$$\begin{array}{lcl} code\ (k,1) & = & [\,] \\ code\ (k,n) & = & \textbf{if}\ odd\ n\ \textbf{then}\ mix\ (code\ (k+1,n-1))\ (down\ (k,n)) \\ & & \textbf{else}\ mix\ (code\ (1,n-1))\ (down\ (k,n)) \end{array}$$

where

$$down\ (k,n) \quad = \quad bump\ k\ [n-1,n-2..1]$$

Now, assuming that

$$code \quad = \quad mixall \cdot map\ down \cdot list$$

we can use the definitions of *code* and *map* and the identity

$$mix\ (mixall\ xss)\ xs \quad = \quad mixall\ (xss +\!\!+ [xs])$$

to obtain

$$\begin{array}{lcl} code\ (k,n) & = & \textbf{if}\ odd\ n\ \textbf{then}\ (mixall \cdot map\ down)\ (list\ (k+1,n-1) +\!\!+ [(k,n)]) \\ & & \textbf{else}\ (mixall \cdot map\ down)\ (list\ (1,n-1) +\!\!+ [(k,n)]) \end{array}$$

It follows that we may define *list* recursively by

$$\begin{array}{lcl} list\ (k,1) & = & [\,] \\ list\ (k,n) & = & \textbf{if}\ odd\ n\ \textbf{then}\ list\ (k+1,n-1) +\!\!+ [(k,n)] \\ & & \textbf{else}\ list\ (1,n-1) +\!\!+ [(k,n)] \end{array}$$

We can improve this definition to reduce the number of steps from $O(n^2)$ to $O(n)$ using the prelude function *zip* with infinite lists:

$$\begin{array}{lcl} list\ (k,1) & = & [\,] \\ list\ (k,n) & = & \textbf{if}\ odd\ n\ \textbf{then}\ zip\ twoones\ [2..n-2] +\!\!+ [(k+1,n-1),(k,n)] \\ & & \textbf{else}\ zip\ twoones\ [2..n-2] +\!\!+ [(1,n-1),(k,n)] \\ & & \textbf{where}\ twoones \quad = \quad 2:1:twoones \end{array}$$

If $k = 0$ then $list\ (k,n)$ is the same in both odd and even cases. Therefore, we have

$$johnson \quad = \quad mixall \cdot map\ down \cdot list'$$

where

$$\begin{array}{lcl} list'\ 1 & = & [\,] \\ list'\ n & = & zip\ twoones\ [2..n-2] +\!\!+ [(1,n-1),(0,n)] \\ & & \textbf{where}\ twoones \quad = \quad 2:1:twoones \end{array}$$

Applying the loopless version of *mixall* using forests from Chapter 6, a loopless form for *johnson* follows:

$$johnson \quad = \quad unfoldr\ step \cdot prolog$$

where

$$prolog \quad = \quad wrapQueue \cdot fst \cdot foldr\ tmix\ (empty, empty) \cdot map\ down \cdot list'$$

We do not yet have a loopless functional algorithm since $map\ down \cdot list$ takes $O(n)$ time under strict evaluation.

In Section 6.4, we derived an implementation of *mixall* that, subject to certain conditions, was loopless and took as its argument a list generated by a loopless functional algorithm. Therefore, to make progress, we will try to express *down* looplessly.

We consider types *Direction* and *Instruction* given by

$$\begin{aligned} \textbf{data}\ \mathit{Direction} &= \mathit{Down} \mid \mathit{DownSkip} \mid \mathit{Up} \mid \mathit{UpSkip} \\ \textbf{type}\ \mathit{Instruction} &= (\mathit{Direction}, \mathit{Int}, \mathit{Int}, \mathit{Int}) \end{aligned}$$

It is then easy to derive the following loopless functional algorithms:

$$\begin{aligned} \mathit{down} &= \mathit{unfoldr}\ \mathit{step}' \cdot \mathit{fst} \cdot \mathit{prolog}' \\ \mathit{reverse} \cdot \mathit{down} &= \mathit{unfoldr}\ \mathit{step}' \cdot \mathit{snd} \cdot \mathit{prolog}' \end{aligned}$$

where

$$\begin{aligned} \mathit{prolog}'\ (k, n) &= \textbf{if}\ \mathit{even}\ n\ \textbf{then}\ ((\mathit{Down}, k, n-1, 1), (\mathit{Up}, k, 1, n-1)) \\ &\quad\ \textbf{else}\ ((\mathit{Down}, k, n-1, 1), (\mathit{UpSkip}, k, 1, n-1)) \\ \mathit{step}'\ (\mathit{Down}, k, m, n) &= \textbf{if}\ m < n\ \textbf{then}\ \mathit{Nothing} \\ &\quad\ \textbf{else}\ \mathit{Just}\ (m+k, (\mathit{DownSkip}, k, m-1, n)) \\ \mathit{step}'\ (\mathit{DownSkip}, k, m, n) &= \textbf{if}\ m < n\ \textbf{then}\ \mathit{Nothing} \\ &\quad\ \textbf{else}\ \mathit{Just}\ (m, (\mathit{Down}, k, m-1, n)) \\ \mathit{step}'\ (\mathit{Up}, k, m, n) &= \textbf{if}\ m > n\ \textbf{then}\ \mathit{Nothing} \\ &\quad\ \textbf{else}\ \mathit{Just}\ (m+k, (\mathit{UpSkip}, k, m+1, n)) \\ \mathit{step}'\ (\mathit{UpSkip}, k, m, n) &= \textbf{if}\ m > n\ \textbf{then}\ \mathit{Nothing} \\ &\quad\ \textbf{else}\ \mathit{Just}\ (m, (\mathit{Up}, k, m+1, n)) \end{aligned}$$

Substituting this new definition of *down* into *johnson* we have

$$\mathit{johnson} = \mathit{mixall} \cdot \mathit{map}\ (\mathit{unfoldr}\ \mathit{step}' \cdot \mathit{fst} \cdot \mathit{prolog}') \cdot \mathit{list}'$$

Then we define a function length' by

$$\mathit{length}'\ (k, n) = n - 1$$

when satisfies the condition

$$\mathit{length}' = \mathit{unfoldr}\ \mathit{step}' \cdot \mathit{prolog}'$$

Since we also we have step' and prolog' such that

$$\mathit{reverse} \cdot \mathit{unfoldr}\ \mathit{step}' \cdot \mathit{fst} \cdot \mathit{prolog}' = \mathit{unfoldr}\ \mathit{step}' \cdot \mathit{snd} \cdot \mathit{prolog}'$$

it follows that we can use these functions with our generalized loopless functional algorithm

$$\mathit{mixall} \cdot \mathit{map}\ (\mathit{unfoldr}\ \mathit{sp} \cdot \mathit{fst} \cdot \mathit{pg}) = \mathit{unfoldr}\ (\mathit{step}\ \mathit{sp}) \cdot \mathit{prolog}\ \mathit{sp}\ \mathit{lg}\ \mathit{pg}$$

to obtain a loopless functional algorithm for *johnson*:

$$\mathit{johnson} = \mathit{unfoldr}\ (\mathit{step}\ \mathit{step}') \cdot \mathit{prolog}\ \mathit{step}'\ \mathit{length}'\ \mathit{prolog}' \cdot \mathit{list}'$$

The definitions of *step* and *prolog* are shown in Figure 12.5 at the end of this chapter.

1234	2341	3412	4123	(1234)		
2314	3142	1423	4231	(2314)		
3124	1243	2431	4312	(3124)	(1234)	
2134	1342	3421	4213	(2134)		
1324	3241	2413	4132	(1324)		
3214	2143	1432	4321	(3214)	(2134)	(1234)

Figure 12.3. Permutations generated by prefix shifts.

12.3. Prefix shifts and star transpositions

Interchanging adjacent elements may be the simplest way to change from one permutation to the next, but there are several other easily characterizable operations that can be used to generate permutations. The first of these is the prefix shifting of elements to the left between each successive permutation. This procedure is particularly efficient when permutations are kept in a machine register instead of an array.

The following imperative algorithm is possibly the most straightforward for generating permutations, especially in terms of minimality of program length:

ALGORITHM G (Permutation generation by prefix shifts). This algorithm generates all permutations $a_1 a_2 \dots a_n$ of n distinct elements $\{x_1, x_2, \dots, x_n\}$, starting with $x_1 x_2 \dots x_n$ and prefix shifting elements to the left in each step. [4]

G1. (Initialize) Set $a_j \leftarrow x_j$ for $1 \le j \le n$.

G2. (Visit) Visit the permutation $a_1 \dots a_n$.

G3. (Prepare to shift) Set $k \leftarrow n$.

G4. (Shift) Replace $a_1 a_2 \dots a_k$ by the prefix shift $a_2 \dots a_k a_1$. If $a_k \neq x_k$, return to G2.

G5. (Decrease k) Set $k \leftarrow k - 1$. Terminate if $k = 1$. Otherwise return to G4.

This algorithm considers permutations that can be obtained from each other by prefix shifts of the whole configuration as being in some way related. These objects are said to have the same 'circular ordering'. Then the problem of generating all permutations reduces to finding distinct circular orderings. From an initial ordering of length k, we can find another circular ordering by a prefix shift to the left of length $k - 1$. Repeating this shift a total of $k - 1$ times, we generate $k - 1$ distinct circular orders. It follows that for each of these new orders, we can generate a further $k - 2$ circular orders by prefix shifts of length $k - 2$. Continuing this idea, we eventually generate all circular orderings, obtaining k actual permutations from each.

For example, if $n = 4$, the permutations $1234, 2341, 3412$ and 4123 have the same circular ordering. Prefix shifting the first three elements of 1234 leftwards, we have 2314, which has the same circular ordering as $3142, 1423$ and 4231. Continuing, we generate 3124 and related elements from a prefix shift of the first three elements of 2314, but another prefix shift of the same length gives us 1234 again. Therefore, we shift by two elements to obtain 1324 and permutations with equivalent circular ordering. This is illustrated in Figure 12.3. Unvisited intermediate permutations are indicated by parentheses.

A different approach to permutation generation is by 'star transpositions'. These are the interchange of the leftmost element a_0 of a permutation $a_0 a_1 \dots a_k a_{k+1} \dots a_{n-1}$ with another

[4] Knuth (2005a) 56–57, based upon Langdon (1967).

$\bar{1}\bar{2}34$ $\bar{4}2\bar{1}3$ $\bar{3}\bar{4}12$ $\bar{2}4\bar{3}1$
$\bar{2}1\bar{3}4$ $\bar{1}\bar{2}43$ $\bar{4}3\bar{1}2$ $\bar{3}\bar{4}21$
$\bar{3}\bar{1}24$ $\bar{2}1\bar{4}3$ $\bar{1}\bar{3}42$ $\bar{4}3\bar{2}1$
$\bar{1}3\bar{2}4$ $\bar{4}\bar{1}23$ $\bar{3}1\bar{4}2$ $\bar{2}\bar{3}41$
$\bar{2}\bar{3}14$ $\bar{1}4\bar{2}3$ $\bar{4}\bar{1}32$ $\bar{3}2\bar{4}1$
$\bar{3}21\bar{4}$ $\bar{2}4\bar{1}\bar{3}$ $\bar{1}43\bar{2}$ $\bar{4}23\bar{1}$

Figure 12.4. Permutations generated by star transpositions.

element a_k. This changes each permutation in a minimal way using only $n-1$ different transpositions for permutations of length n. For example when $n = 4$, using the interchanges $a_0 \leftrightarrow a_1$, $a_0 \leftrightarrow a_2$ and $a_0 \leftrightarrow a_3$, we obtain the configuration in Figure 12.4.

Star transpositions have an advantage over adjacent interchanges as we do not need to read the value of a_0 from memory since we know it from the previous transposition.

The following imperative algorithm generates all permutations by star transpositions:

ALGORITHM H (Permutation generation by star transpositions). This algorithm generates all permutations of a given sequence $a_0a_1 \dots a_{n-1}$ of n distinct elements, exchanging the leftmost element with another in each step. We use two arrays $b_0b_1 \dots b_{n-1}$ and $c_1c_2 \dots c_n$. [5]

H1. (Initialize) Set $b_j \leftarrow j$ and $c_{j+1} \leftarrow 0$ for $0 \le j < n$.

H2. (Visit) Visit the permutation $a_0 \dots a_{n-1}$.

H3. (Find k) Set $k \leftarrow 1$. Then, if $c_k = k$, set $c_k \leftarrow 0$, $l \leftarrow k+1$ and repeat until $c_k < k$. Terminate if $k = n$. Otherwise set $c_k \leftarrow c_k + 1$.

H4. (Swap) Exchange $a_0 \leftrightarrow a_{b_k}$.

H5. (Flip) Set $j \leftarrow 1$ and $k \leftarrow k-1$. If $j < k$, interchange $b_j \leftrightarrow b_k$, set $j \leftarrow j+1$, $k \leftarrow k-1$ and repeat until $j \ge k$. Return to H2.

If the value of n is relatively small, we can calculate the indices of the elements to be transposed at each step in advance.

[5] Knuth (2005a) 57–58, based upon a previously unpublished algorithm discovered by G. Ehrlich in 1987.

```
data Rose a          = Node a (Queue (Rose a))
data Delay a b       = Hold a b (Queue (Delay a b), Queue (Delay a b))
data Direction       = Down | DownSkip | Up | UpSkip
type Instruction     = (Direction, Int, Int, Int)

johnson              = unfoldr (step step') · prolog step' length' prolog' · list'

list' 1              = [ ]
list' n              = zip twoones [2..n − 2] ++ [(1, n − 1), (0, n)]
                       where twoones = 2 : 1 : twoones

prolog sp lg pg      = wrapQueue · fst · foldr (tmix sp) (empty, empty) · list lg pg

list lg pg xs        = [(lg x, a, b) | x ← xs, (a, b) ← [pg x]]

tmix sp (n, a, b) (ytq, qty)
                     = if even n then (fmix sp a (ytq, qty), fmix sp b (qty, ytq))
                       else (fmix sp a (ytq, qty), fmix sp b (ytq, qty))

fmix sp a (ytq, qty) = case sp a of
                         Nothing     →  ytq
                         Just (x, b) →  insert ytq (Hold x b (ytq, qty))

prolog' (k, n)       = if even n then ((Down, k, n − 1, 1), (Up, k, 1, n − 1))
                       else ((Down, k, n − 1, 1), (UpSkip, k, 1, n − 1))

length' (k, n)       = n − 1

wrapQueue xtq        = consQueue xtq [ ]

consQueue xtq xtqs   = if isEmpty xtq then xtqs
                       else xtq : xtqs

step sp [ ]          = Nothing
step sp (xtq : xtqs) = Just (x, consQueue (fmix sp a (qty, ytq)) (consQueue ztq xtqs))
                       where (Hold x a (ytq, qty), ztq) = remove xtq

step' (Down, k, m, n)= if m < n then Nothing
                       else Just (m + k, (DownSkip, k, m − 1, n))
step' (DownSkip, k, m, n)
                     = if m < n then Nothing
                       else Just (m, (Down, k, m − 1, n))
step' (Up, k, m, n)  = if m > n then Nothing
                       else Just (m + k, (UpSkip, k, m + 1, n))
step' (UpSkip, k, m, n)
                     = if m > n then Nothing
                       else Just (m, (Up, k, m + 1, n))
```

Figure 12.5. A loopless functional algorithm for johnson using forests.

Chapter 13

Generating Combinations

> *Nature is an endless combination and repetition of a very few laws. She hums the old well-known air through innumerable variations.*
>
> RALPH WALDO EMERSON, *'History', Essays, First Series, 1841*

A t-combination is a subset of size t taken from a given set of size n. This is equivalent to choosing the $n - t$ elements not selected. Therefore, we can also call these subsets (s, t)-combinations, where

$$n = s + t.$$

We will represent (s, t)-combinations in two ways:

(i) As an index list of the elements $c_t \dots c_2 c_1$ that have been selected, where

$$n > c_t > \cdots > c_2 > c_1 \geq 0.$$

(ii) As a bit string $a_n \dots a_2 a_1$ where a 1 bit refers to a selected element and a 0 bit to an unselected element. Clearly, there will be s bits set to 0 and t bits set to 1, so it follows that

$$a_n + \cdots + a_1 = t.$$

13.1. Gray codes for combinations and the Liu-Tang algorithm

We can extract a Gray code for (s, t)-combinations from the binary reflected Gray code for n bit numbers by deleting all those elements corresponding to subsets which do not have exactly t elements. The bit strings that remain form a sequence of successive combinations that differ by exactly one element. For example the bit strings $a_6 a_5 a_4 a_3 a_2 a_1$ of a $(3, 3)$-combination are shown in Figure 13.1(a). This list is cyclic, and we can find the Gray binary code for a $(2, 3)$-combination by taking the first two columns of this array. The corresponding index list forms $c_3 c_2 c_1$ appear in Figure 13.1(b).

This arrangement is known as 'revolving door combinations' since we can consider the situation which is illustrated in Figure 13.2 for $(10, 10)$-combinations. An (s, t)-combination represents two rooms that contain s and t people, respectively. Between the two rooms there is a revolving door and whenever a person moves to the other room, someone from that room replaces them in the opposite direction.

[1] Tang & Liu (1973) 176–177.

(a) Bit string form				(b) Index list form			
$00\bar{0}1\bar{1}1$	$011\bar{0}\bar{1}0$	$1100\bar{0}\bar{1}$	$101\bar{0}\bar{1}0$	$3\bar{2}1$	$54\bar{2}$	$65\bar{1}$	$64\bar{2}$
$0011\bar{0}\bar{1}$	$01\bar{1}10\bar{0}$	$110\bar{0}\bar{1}0$	$10\bar{1}10\bar{0}$	$43\bar{1}$	$54\bar{3}$	$65\bar{2}$	$64\bar{3}$
$001\bar{1}1\bar{0}$	$0101\bar{0}\bar{1}$	$11\bar{0}\bar{1}00$	$1001\bar{0}\bar{1}$	$4\bar{3}2$	$53\bar{1}$	$65\bar{3}$	$63\bar{1}$
$0\bar{0}10\bar{1}1$	$010\bar{1}1\bar{0}$	$1\bar{1}100\bar{0}$	$100\bar{1}1\bar{0}$	$4\bar{2}1$	$5\bar{3}2$	$6\bar{5}4$	$6\bar{3}2$
$0110\bar{0}\bar{1}$	$\bar{0}100\bar{1}1$	$1010\bar{0}\bar{1}$	$\bar{1}00\bar{0}11$	$54\bar{1}$	$5\bar{2}1$	$64\bar{1}$	$\bar{6}21$

Figure 13.1. Revolving door combinations.

We have the following recursive definition for revolving door combinations:

$$L_{st} = \begin{cases} 0^s & \text{if } t = 0 \\ 1^t & \text{if } s = 0 \\ 0L_{(s-1)t},\ 1L^R_{s(t-1)} & \text{otherwise} \end{cases}$$

Clearly, this takes the same form as the recursive definition of binary reflective Gray codes in Section 9.1, except for an additional parameter.

THEOREM 13.1. The sequence L_{st} has the following properties:

(i) Successive bit strings differ by the interchange of one pair of bits.

(ii) The first element of L_{st} is 0^s1^t.

(iii) The last element of L_{st} is 10^s1^{t-1} for $st > 0$.

Additionally, the transition between the last and first strings also satisfies the revolving door constraint. The following sequences show the different cases that can occur and demonstrate these properties:

$s = 1$	otherwise	$t = 1$
	0^s1^t	0^s1
0111^{t-2}	$\vdots$	$\vdots$
	$010^{s-1}11^{t-2}$	010^{s-1}
1101^{t-2}	$110^{s-1}01^{t-2}$	
$\vdots$	$\vdots$	100^{s-1}
101^{t-1}	10^s1^{t-1}	

The imperative Liu-Tang algorithm visits the index list forms of all combinations in revolving door order:

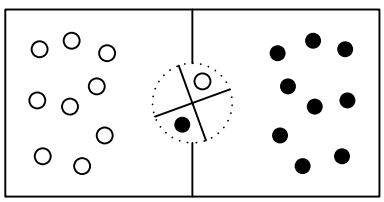


Figure 13.2. Two rooms separated by a revolving door.

Figure 13.3. A graphical representation of revolving door combinations.

ALGORITHM I (Revolving door combinations). This algorithm generates list representations of all (s,t)-combinations $c_t \dots c_2 c_1$ of n elements $\{0,1,\dots,n-1\}$, where $n = s+t$ and $n > t > 1$. Lists with a common prefix occur consecutively. [2]

I1. (Initialize) Set $c_j \leftarrow j-1$ for $t \geq j \geq 1$ Also set $c_{t+1} \leftarrow n$.

I2. (Visit) Visit the combination $c_t \dots c_2 c_1$.

I3. (Easy case?) If t is odd: If $c_1 + 1 < c_2$, set $c_1 \leftarrow c_1 + 1$ and return to I2, otherwise set $j \leftarrow 2$ and go to I4. If t is even: If $c_1 > 0$, set $c_1 \leftarrow c_1 - 1$ and return to I2, otherwise set $j \leftarrow 2$ and go to I5.

I4. (Try to decrease c_j) If $c_k \geq j$, set $c_j \leftarrow c_{j-1}$, $c_{j-1} \leftarrow j-2$ and return to I2. Otherwise set $j \leftarrow j+1$.

I5. (Try to increase c_j) If $c_j + 1 < c_{j+1}$, set $c_{j-1} \leftarrow c_j$, $c_j \leftarrow c_j + 1$ and return to I2. Otherwise set $j \leftarrow j+1$ and go to I4 if $k \leq t$.

A graphical representation of revolving door $(5,5)$-combinations is displayed in Figure 13.3.

13.2. Combination generation by homogenous transitions

While the revolving door Gray code changes only one element of the combination at each step, it must often change two of the indices of c_j simultaneously to preserve the listing $c_t > \cdots > c_2 > c_1$. This is because the recursive definition of C_{st} involves transitions of the form $110^a0 \leftrightarrow 010^a1$ for $a \geq 1$. However, it is possible modify our definition so that only one index changes at each step. A Gray code for combinations with this property is known as 'homogeneous', characterized by having only transitions of the form $10^a \leftrightarrow 0^a1$. This suggests that a suitable sequence would begin with 0^s1^t and end with 1^t0^s. It follows that we may define a homogenous sequence of combinations by

$$K_{st} = \begin{cases} \epsilon & \text{if } t = -1 \\ 0^s & \text{if } t = 0 \\ 1^t & \text{if } s = 0 \\ 0K_{(s-1)t},\ 10K^R_{(s-1)(t-1)},\ 11K_{s(t-2)} & \text{otherwise} \end{cases}$$

This relation results from only a small modification to the recursive definition of binary reflected Gray codes.

THEOREM 13.2. The sequence K_{st} has the following properties:

(i) Successive bit strings differ by an interchange of the form $10^a \leftrightarrow 0^a1$ for $a \geq 1$.

(ii) The first element of K_{st} is 0^s1^t.

[2] Knuth (2005b) 8–10, based upon Payne & Ives (1979) and Liu & Tang (1973).

(a) Homogeneous

$00\bar{0}\bar{1}11$ $010\bar{1}\bar{0}1$ $101\bar{1}0\bar{0}$ $1\bar{0}00\bar{1}1$
$001\bar{0}\bar{1}1$ $01\bar{0}0\bar{1}1$ $1010\bar{0}\bar{1}$ $1100\bar{0}\bar{1}$
$0011\bar{0}\bar{1}$ $0110\bar{0}\bar{1}$ $10\bar{1}\bar{0}10$ $1100\bar{0}\bar{1}0$
$0\bar{0}\bar{1}110$ $011\bar{0}\bar{1}0$ $1001\bar{1}\bar{0}$ $11\bar{0}\bar{1}00$
$0101\bar{1}\bar{0}$ $\bar{0}\bar{1}1100$ $100\bar{1}\bar{0}1$ 111000

(b) Adjacent interchange

$00\bar{0}\bar{1}11$ $0\bar{1}\bar{0}101$ $1\bar{0}\bar{1}010$ $011\bar{0}\bar{1}0$
$0\bar{0}\bar{1}011$ $0011\bar{0}\bar{1}$ $1100\bar{1}\bar{0}$ $\bar{0}\bar{1}1100$
$\bar{0}\bar{1}0011$ $0\bar{0}\bar{1}110$ $1100\bar{0}\bar{1}$ $1\bar{0}\bar{1}100$
$100\bar{0}\bar{1}11$ $\bar{0}\bar{1}0110$ $\bar{1}\bar{0}1001$ $11\bar{0}\bar{1}00$
$\bar{1}\bar{0}0101$ $100\bar{1}\bar{0}10$ $011\bar{0}\bar{1}$ 111000

Figure 13.4. Homogeneous and adjacent interchange combinations.

(iii) The last element of K_{st} is 1^t0^s.

In this case, the last and first strings do not share the properties of successive strings and so do not differ by a homogeneous transition. The properties that K_{st} does possess, are illustrated by the following sequence for $n > t > 1$:

$$\left.\begin{array}{l} 00^{s-1}111^{t-2} \\ \vdots \\ 0111^{t-2}0^{s-1} \end{array}\right\} 0K_{(s-1)t}$$
$$\left.\begin{array}{l} 1011^{t-2}0^{s-1} \\ \vdots \\ 100^{s-1}11^{t-2} \end{array}\right\} 10K^R_{(s-1)(t-1)}$$
$$\left.\begin{array}{l} 110^{s-1}01^{t-2} \\ \vdots \\ 111^{t-2}00^{s-1} \end{array}\right\} 11K_{s(t-2)}$$

Therefore, for a bit string indexed $a_n \ldots a_2a_1$, the change at the first interface between subsequences interchanges the bits a_n and a_{n-1}, while at the second interface the interchanges are between the bits a_{n-1} and a_{t-1}. The homogeneous Gray code for $(3, 3)$-combinations is shown in Figure 13.4(a), while a graphical illustration of $(5, 5)$ appears in Figure 13.5.

We can also generate Gray codes for combinations in which each step causes an index c_j to change by at most 2. This corresponds to the transitions $01 \leftrightarrow 10$ or $001 \leftrightarrow 100$.

Unfortunately, it is not possible in general to limit our transitions further to only allow adjacent interchanges and generate all combinations in an analogous way to the Johnson-Trotter algorithm for permutations in Chapter 12:

THEOREM 13.3. The generation of all (s, t)-combinations $a_{s+t-1} \ldots a_1a_0$ by adjacent interchanges $01 \leftrightarrow 10$ is possible if and only if $s \leq 1$ or $t \leq 1$ or st is odd.

For example, the following listing of $(3, 3)$-combinations in Figure 13.4(b) is possible. A visualization of $(5, 5)$-combinations is shown in Figure 13.6

In the non-trivial cases when st is not odd, problems with the parities of the bit strings prevent generation by adjacent interchanges. All current imperative algorithms to generate

Figure 13.5. A graphical representation of homogeneous combinations.

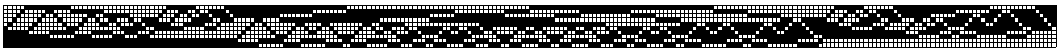


Figure 13.6. A graphical representation of adjacent interchange combinations.

combinations with this minimal change property are difficult to implement efficiently, for example using techniques such recursive coroutines. [3]

13.3. Combination generation by prefix shifts

In Section 12.3, we saw an algorithm for generating all permutations using prefix shifts. It is also possible to generate all combinations in bit string form using this method. Consider the following recursive definition:

$$W'_{st} = 1^t0^s,\ W_{st}$$

where

$$W_{st} = \begin{cases} \epsilon & \text{if } s = 0 \text{ or } t = 0 \\ W_{(s-1)t}0,\ W_{s(t-1)}1,\ 1^{t-1}0^s1 & \text{otherwise} \end{cases}$$

The order of elements produced by this relation is known as 'cool-lex'. We have the following result:

THEOREM 13.4. The sequence W_{st} has the following properties:

(i) Each bit string of an (s,t)-combination appears in W_{st} exactly once, except 1^t0^s.

(ii) Successive bit strings differ by a prefix shift of one position to the right.

(iii) Successive bit strings differ by the interchange of one or two pairs of bits.

(iv) The first element of W_{st} is 01^t0^{s-1}.

(v) The last element of W_{st} is $1^{t-1}0^s1$.

The sequence W_{st} contains each (s,t)-combination exactly once except 1^t0^s because $W_{(s-1)t}0$ is a sequence of all combinations that end with a 0 bit, except for $1^t0^{s-1}0$, while $W_{s(t-1)1}$ is a sequence of all combinations that end with a 1 bit, except $1^{t-1}0^s1$ which is, of course, appended to the end of the sequence.

To show the other properties, we just need to examine the interfaces between the three subsequences of W_{st}. In the sequences below, the general case is shown twice, illustrating prefix shifts on the left and exchanges of bits on the right.

$s = 1$	otherwise		$t = 1$
	$011^{t-2}10^{s-2}0$	$011^{t-2}10^{s-2}0$	$010^{s-2}0$
	$\vdots$	$\vdots$	$\vdots$
	$11^{t-2}00^{s-2}10$	$\bar{1}1^{t-2}\bar{0}0^{s-2}\bar{1}\bar{0}$	$0^{s-2}010$
$011^{t-2}1$	$011^{t-2}00^{s-2}1$	$01^{t-2}10^{s-2}01$	
$\vdots$	$\vdots$	$\vdots$	
$1^{t-2}011$	$1^{t-2}00^{s-2}011$	$1^{t-2}\bar{0}00^{s-2}\bar{1}1$	
$11^{t-2}01$	$11^{t-2}00^{s-2}01$	$1^{t-2}100^{s-2}01$	$00^{s-2}01$

[3] Eades *et al.* (1984).

111000	101010	011001	010011
011100	010110	101001	001011
101100	001110	010101	000111
110100	100110	001101	100011
011010	110010	100101	110001

Figure 13.7. Combinations in cool-lex order.

From this, it is easy to see that at the interfaces between the subsequences, the bit strings differ by a prefix shift of all n positions. Moreover, two bits change between strings at the first interface while just one bit changes at the second.

Two other observations about cool-lex order follow from these properties. Since the last string of W_{st} is equal to the complement of the reverse of the first string of W_{ts}, we could define cool-lex order in other ways. We also have an easy way to generate successive algorithms by identifying the shortest prefix ending in 010 or 011 and then shifting it by one position to right. If no such prefix exists then the whole string is shifted. Formally, a loopless imperative algorithm for generating combinations is as follows:

ALGORITHM J (Combination generation by prefix shifts). This loopless algorithm generates [4] binary string representations of all (s,t)-combinations $a_n \ldots a_2 a_1$, where $n > t > 0$, prefix shifting elements to the right in each step.

J1. (Initialize) Set $n \leftarrow s+t$ and $a_j \leftarrow 0$ for $1 \le j \le t$, and $a_j \leftarrow 1$ for $s < j \le n$. Also set $j \leftarrow k \leftarrow s+1$.

J2. (Visit) Visit the combination $a_n \ldots a_2 a_1$.

J3. (Zero out a_j) Set $a_j \leftarrow 0$ and $j \leftarrow j-1$.

J4. (Easy case?) If $a_j = 1$, set $a_k \leftarrow 1$, $k \leftarrow k-1$ and return to J2.

J5. (Wrap around) Terminate if $j = 0$. Otherwise set $a_j \leftarrow 1$. Then if $k < n$, set $a_k \leftarrow 1$, $a_n \leftarrow 0$, $j \leftarrow n-1$, $k \leftarrow n$ and return to J2.

All $(3,3)$-combinations in cool-lex order are shown in Figure 13.7. A graphical illustration of $(5,5)$-combinations appears in Figure 13.8.

Clearly, we can consider the transitions between combinations to be a prefix shift of length k. A function *coollex′* for these transitions follows almost directly from the recursive definition:

$$\begin{array}{lcl} \mathit{coollex}' & :: & [\mathit{Int}] \to [\mathit{Int}] \to [\mathit{Int}] \\ \mathit{coollex}'\ s\ t & = & (s+t) : \mathit{coollex}\ s\ t \end{array}$$

where

$$\begin{array}{lcl} \mathit{coollex}\ 1\ 1 & = & [\,] \\ \mathit{coollex}\ 1\ t & = & \mathit{coollex}\ 1\ (t-1) +\!\!+ [t+1] \\ \mathit{coollex}\ s\ 1 & = & \mathit{coollex}\ (s-1)\ 1 +\!\!+ [s+1] \\ \mathit{coollex}\ s\ t & = & \mathit{coollex}\ (s-1)\ t +\!\!+ [s+t] +\!\!+ \mathit{coollex}\ s\ (t-1) +\!\!+ [s+t] \end{array}$$

This gives the following transitions for the sequence of $(3,3)$-combinations in Figure 13.7:

$$6, 3, 4, 5, 3, 4, 3, 4, 5, 6, 3, 4, 3, 4, 5, 3, 4, 5, 6$$

[4] Ruskey & Williams (2005) and Knuth (2005b) 97–98.

Figure 13.8. A graphical representation of combinations in cool-lex order.

These are not unique. For example, the transition from 11100 to 011100 could be achieved with prefix shifts of lengths 4, 5 or 6. However, those generated by *coollex′* correspond to choosing the shortest prefix ending in 010 or 011, or the entire bit string if neither of these prefixes exist. This is always possible. The concatenation of the singleton lists $[t+1]$, $[s+1]$ and $[s+t]$ in the definition follows because we can see from the properties of cool-lex order that in each case the penultimate strings of W_{st}, W_{1t} and W_{s1} only contain the substrings 010 and 011 as their final three elements.

Bibliography

Articles

Bird, R. S. (2005a). The Li-Ruskey algorithm. In preparation.

Bird, R. S. (2005b). Loopless functional algorithms. In preparation.

Bitner, J. R., Ehrlich, G. & Reingold, E. M. (1976). Efficient generation of the binary reflected Gray code and its applications. *Communications of the ACM* **19** (9) 517–521.

Eades, P., Hickey, M. & Read, R. C. (1984). Some Hamilton paths and a minimal change algorithm. *Journal of the ACM* **31** (1) 19–29.

Ehrlich, G. (1973a). Algorithm 466: Four combinatorial algorithms. *Communications of the ACM* **16** (11) 690–691.

Ehrlich, G. (1973b). Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM* **20** (3) 500–513.

Filliâtre, J.-C. & Pottier, F. (2003). Producing all ideals of a forest, functionally. *Journal of Functional Programming* **13** (5) 945–956.

Gibbons, J. & Jones, G. (1998). The under-appreciated unfold. *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, ICFP '98*, 273–279. ACM Press, New York, United States.

Goddyn, L. & Gvozdjak, P. (2003). Binary gray codes with long bit runs. *Electronic Journal of Combinatorics* **10** (1) R27.

Hood, R. & Melville, R. (1981). Real-time queue operations in pure LISP. *Information Processing Letters* **13** (2) 50–54.

Johnson, S. M. (1963). Generation of permutations by adjacent transposition. *Mathematics of Computation* **17** (83) 282–285.

Knuth, D. E. & Ruskey, F. (2004). Efficient coroutine generation of constrained Gray sequences. O. Owe, S. Krogdahl & T. Lyche (eds.) *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, number 2635 in Lecture Notes in Computer Science, 183–204. Springer-Verlag, Heidelberg, Baden-Württemberg, Germany.

Koda, Y. & Ruskey, F. (1993). A Gray code for the ideals of a forest poset. *Journal of Algorithms* **15** (2) 324–340.

Langdon, G. G., Jr (1967). An algorithm for generating permutations. *Communications of the ACM* **10** (5) 298–299.

Liu, C. N. & Tang, D. T. (1973). Algorithm 452: Enumerating combinations of m out of n objects. *Communications of the ACM* **16** (8) 485.

Okasaki, C. (1995). Simple and efficient purely functional queues and deques. *Journal of Functional Programming* **5** (4) 583–592.

Payne, W. H. & Ives, F. M. (1979). Combination generators. *ACM Transactions on Mathematical Software* **5** (2) 163–172.

Ruskey, F. & Williams, A. (2005). Generating combinations by prefix shifts. L. Wang (ed.) *Computing and Combinatorics: Proceedings of the 11th Annual International Conference, COCOON 2005*, number 3595 in Lecture Notes in Computer Science, 570–576. Springer-Verlag, Heidelberg, Baden-Württemberg, Germany.

Savage, C. D. (1997). A survey of combinatorial Gray codes. *SIAM Reviews* **39** (4) 605–629.

Sedgewick, R. (1977). Permutation generation methods. *Computing Surveys* **9** (2) 137–164.

Tang, D. T. & Liu, C. N. (1973). Distance-2 cyclic chaining of constant-weight codes. *IEEE Transactions on Computers* **C-22** (2) 176–180.

Trotter, H. F. (1962). Algorithm 115: PERM. *Communications of the ACM* **5** (8) 434–435.

Books

Bird, R. S. (1998). *Introduction to Functional Programming using Haskell*. Second edition. Prentice Hall Europe, London, United Kingdom.

Bird, R. S. & de Moor, O. (1997). *Algebra of Programming*. Prentice Hall Europe, London, United Kingdom.

Even, S. (1973). *Algorithmic Combinatorics*. Macmillan, New York, United States.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Third edition. Addison-Wesley, Reading, Massachusetts, United States.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3, Sorting and Searching*. Second edition. Addison-Wesley, Reading, Massachusetts, United States.

Knuth, D. E. (2005a). *The Art of Computer Programming, Volume 4, Fascicle 2, Generating All Tuples and Permutations*. Addison-Wesley, Upper Saddle River, New Jersey, United States.

Knuth, D. E. (2005b). *The Art of Computer Programming, Volume 4, Fascicle 3, Generating All Combinations and Partitions*. Addison-Wesley, Upper Saddle River, New Jersey, United States.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, Cambridge, United Kingdom.

Peyton Jones, S. L. (ed.) (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, United Kingdom.

Ruskey, F. (2003). *Combinatorial Generation*. Preliminary working draft. University of Victoria, Victoria, British Columbia, Canada.