JFP **19** (5): 491–508, 2009. © 2009 Cambridge University Press
doi:10.1017/S0956796809990116 Printed in the United Kingdom

FUNCTIONAL PEARLS
The Bird Tree

RALF HINZE
Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
(*e-mail:* ralf.hinze@comlab.ox.ac.uk)

1 Introduction

Sadly, Richard Bird is stepping down as the editor of the 'Functional Pearls' column. As a farewell present, I would like to dedicate a tree to him. A woody plant is appropriate for at least two reasons: Richard has been preoccupied with trees in many of his pearls, and where else would you find a bird's nest? Actually, there is a lot of room for nests, as the tree is infinite. Figure 1 displays the first five levels. The Bird tree, whose nodes are labelled with rational numbers, enjoys several remarkable properties.

Firstly, it is a *fractal* object, in the sense that parts of it are similar to the whole. The Bird tree can be transformed into its left subtree by first incrementing and then reciprocalising the elements. To obtain the right subtree, we have to interchange the order of the two steps: the elements are first reciprocalised and then incremented. This description can be nicely captured by a *co-recursive* definition, given in the purely functional programming language Haskell (Peyton Jones 2003):

$$\begin{aligned}&\mathit{bird} :: \mathit{Tree}\ \mathit{Rational}\\ &\mathit{bird} = \mathit{Node}\ 1\ (1\ /\ (\mathit{bird} + 1))\ ((1\ /\ \mathit{bird}) + 1).\end{aligned}$$

The definitions that make this work are introduced in the next section. For the moment, it suffices to know that the arithmetic operations are lifted pointwise to trees. For instance, $\mathit{bird} + 1$ is the same as $\mathit{map}\ (+1)\ \mathit{bird}$.

Returning to the tree properties, the picture suggests that mirroring the tree yields its reciprocal, $\mathit{mirror}\ \mathit{bird} = 1\ /\ \mathit{bird}$, and this is indeed the case. Furthermore, consider the sequence of rationals along the left (or the right) spine of the Bird tree. We discover some old friends: each fraction consists of two consecutive *Fibonacci* numbers. In other words, we approximate the *golden ratio* $\phi = (1 + \sqrt{5})/2$ as we go down the right spine. The tree also contains the natural numbers. For those, we have to descend in a zigzag fashion: right, left, right, left and so forth. On the other hand, if we list the numerators (or denominators) level-wise, we obtain a somewhat obscure sequence, which is not even listed in Sloane's 'On-Line Encyclopedia of Integer Sequences' (2009).[1]

[1] I have submitted the sequences, numerators and denominators of *bird* and its bit-reversal permutation tree (see Section 3) to the 'On-Line Encyclopedia of Integer Sequences' (preliminary *A*-numbers: *A*162909–*A*162912).

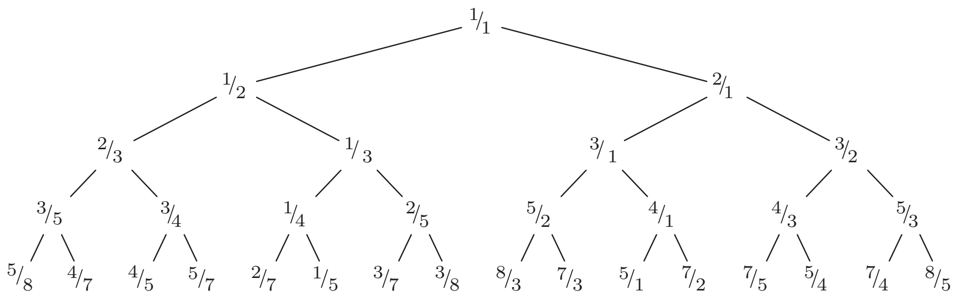


Fig. 1. The Bird tree.

The most intriguing property of the Bird tree, however, is the following: Like the Stern–Brocot tree (Graham *et al.* 1994) or the Calkin–Wilf tree (Calkin & Wilf 2000), it enumerates all the positive rationals. In other words, the tree contains every positive rational exactly once.

The purpose of this pearl is twofold. First, we shall, of course, justify the claims made above. Sections 2 and 3 work towards this goal, reviewing the main proof technique and relating recursive and iterative tree definitions. Section 4 then shows that the Bird tree and the Stern–Brocot tree are level-wise permutations of each other. Second, we aim to derive a loopless algorithm for linearising the Bird tree. Section 6 develops a general algorithm, with Section 5 preparing the ground.

2 Infinite trees, idioms and unique fixed points

In a lazy functional language such as Haskell, infinite trees are easy to define:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Node}\ \{\mathit{root} :: \alpha, \mathit{left} :: \mathit{Tree}\ \alpha, \mathit{right} :: \mathit{Tree}\ \alpha\}.$$

The type *Tree* α is a so-called co-inductive datatype. Its definition is similar to the standard textbook definition of binary trees, except that there is no base constructor, so we cannot build a finite tree. Since there is no base case, *mirror* is a one-liner:

```
mirror              :: Tree α → Tree α
mirror (Node a l r) = Node a (mirror r) (mirror l).
```

The function *mirror* like many more to come relies critically on lazy evaluation.

The definition of *bird* uses $+$ and $/$ lifted to trees. We obtain these liftings almost for free, as *Tree* is a so-called applicative functor or idiom (McBride & Paterson 2008):

```
infixl 9 ⋄
class Idiom φ where
  pure :: α → φ α
  (⋄)  :: φ (α → β) → (φ α → φ β)

instance Idiom Tree where
  pure a = t where t = Node a t t
  t ⋄ u  = Node ((root t) (root u)) (left t ⋄ left u) (right t ⋄ right u).
```

The call *pure a* constructs an infinite tree of *a*s; the idiomatic application $\diamond$ takes a tree of functions and a tree of arguments to a tree of results.

Every instance of *Idiom* must satisfy

$$\begin{aligned}
&\mathit{pure}\ \mathit{id} \diamond u &&= u && \text{(identity)}\\
&\mathit{pure}\ (\circ) \diamond u \diamond v \diamond w &&= u \diamond (v \diamond w) && \text{(composition)}\\
&\mathit{pure}\ f \diamond \mathit{pure}\ x &&= \mathit{pure}\ (f\ x) && \text{(homomorphism)}\\
&u \diamond \mathit{pure}\ x &&= \mathit{pure}\ (\lambda f \to f\ x) \diamond u, && \text{(interchange)}
\end{aligned}$$

which allow us to rewrite every idiom expression into the form $\mathit{pure}\ f \diamond a_1 \diamond \cdots \diamond a_n$. So idioms capture the idea of applying a pure function to ‘impure’ arguments.

We single out two special cases that we will need time and again: $\mathit{map}\ f\ t = \mathit{pure}\ f \diamond t$ and $\mathit{zip}\ g\ t\ u = \mathit{pure}\ g \diamond t \diamond u$. The function *zip* lifts a binary operator to an idiomatic structure; for instance, $(\star) = \mathit{zip}\ (,)$ turns a pair of trees into a tree of pairs. In general, $\mathit{pure}\ f \diamond a_1 \diamond \cdots \diamond a_n$ lifts an *n*-ary function pointwise to an idiomatic structure. Using this ‘idiom’ we can define a generic instance of *Num*:[2]

$$\begin{aligned}
&\mathbf{instance}\ (\mathit{Idiom}\ \phi, \mathit{Num}\ \alpha) \Rightarrow \mathit{Num}\ (\phi\ \alpha)\ \mathbf{where}\\
&\quad (+) &&= \mathit{zip}\ (+)\\
&\quad (-) &&= \mathit{zip}\ (-)\\
&\quad (*) &&= \mathit{zip}\ (*)\\
&\quad \mathit{negate} &&= \mathit{map}\ \mathit{negate}\\
&\quad \mathit{fromInteger} &&= \mathit{pure} \circ \mathit{fromInteger}.
\end{aligned}$$

In this pearl, we consider two idioms, infinite trees and streams. In both cases, the familiar arithmetic laws also hold for the lifted operators.

Every structure comes equipped with structure-preserving maps; so do idioms: a map $h :: \phi\ \alpha \to \psi\ \alpha$ is an *idiom homomorphism* iff

$$\begin{aligned}
h\ (\mathit{pure}\ a) &= \mathit{pure}\ a && (1)\\
h\ (x \diamond y) &= h\ x \diamond h\ y. && (2)
\end{aligned}$$

The map *mirror* is an example of an idiom homomorphism; it is even an *idiom isomorphism*, since $\mathit{mirror} \circ \mathit{mirror} = \mathit{id}$. This fact greatly simplifies reasoning, as we can, for instance, effortlessly rewrite $\mathit{mirror}\ ((1\ /\ \mathit{bird}) + 1) = \mathit{mirror}\ (\mathit{pure}\ (+) \diamond (\mathit{pure}\ (/) \diamond \mathit{pure}\ 1 \diamond \mathit{bird}) \diamond \mathit{pure}\ 1)$ to $\mathit{pure}\ (+) \diamond (\mathit{pure}\ (/) \diamond \mathit{pure}\ 1 \diamond \mathit{mirror}\ \mathit{bird}) \diamond \mathit{pure}\ 1 = (1\ /\ \mathit{mirror}\ \mathit{bird}) + 1$.

This is all very well, but how do we prove the idiom and the homomorphism laws in the first place? It turns out that the type of infinite trees enjoys an attractive and easy-to-use proof principle. Consider the recursion equation $x = \mathit{Node}\ a\ l\ r$, where l and r possibly contain the variable x but *not* the expressions $\mathit{root}\ x$, $\mathit{left}\ x$ or $\mathit{right}\ x$. Equations of this syntactic form possess a *unique solution.* (Rutten (2003) shows an analogous statement for streams; the proof, however, can be readily adapted to infinite trees.) Uniqueness can be exploited to prove that two infinite

[2] Unfortunately, this doesn’t quite work with the Standard Haskell libraries, as *Num* has two superclasses, *Eq* and *Show*, which can’t sensibly be defined generically.

trees are equal: if they satisfy the same recursion equation, then they are. The proof of $1 \,/\, \mathit{bird} = \mathit{mirror}\ \mathit{bird}$ illustrates the idea:

$$
\begin{array}{ll}
& 1 \,/\, \mathit{bird} \\
= & \quad \{ \text{ definition of } \mathit{bird} \} \\
& 1 \,/\, \mathit{Node}\ 1\ (1 \,/\, (\mathit{bird} + 1))\ ((1 \,/\, \mathit{bird}) + 1) \\
= & \quad \{ \text{ arithmetic } \} \\
& \mathit{Node}\ 1\ ((1 \,/\, (1 \,/\, \mathit{bird})) + 1)\ (1 \,/\, ((1 \,/\, \mathit{bird}) + 1)) \\
\subset & \quad \{\ x = \mathit{Node}\ 1\ ((1 \,/\, x) + 1)\ (1 \,/\, (x + 1)) \text{ has a unique solution } \} \\
& \mathit{Node}\ 1\ ((1 \,/\, \mathit{mirror}\ \mathit{bird}) + 1)\ (1 \,/\, (\mathit{mirror}\ \mathit{bird} + 1)) \\
= & \quad \{\ \mathit{mirror} \text{ is an idiom homomorphism } \} \\
& \mathit{Node}\ 1\ (\mathit{mirror}\ ((1 \,/\, \mathit{bird}) + 1))\ (\mathit{mirror}\ (1 \,/\, (\mathit{bird} + 1))) \\
= & \quad \{ \text{ definition of } \mathit{mirror} \text{ and } \mathit{bird} \} \\
& \mathit{mirror}\ \mathit{bird}.
\end{array}
$$

The link $\subset$ indicates that the proof rests on the *unique fixed-point principle*; the recursion equation is given within the curly braces. The upper part shows that $1 \,/\, \mathit{bird}$ satisfies the equation $x = \mathit{Node}\ 1\ ((1 \,/\, x) + 1)\ (1 \,/\, (x + 1))$; the lower part establishes that *mirror bird* satisfies the same equation. The symbol $\subset$ links the two parts, effectively proving the equality of both expressions. As regards contents, the proof relies on the facts that 1 is a fixed point of the reciprocal function and that reciprocal is an *involution*.

Exercise 1 *Using the unique fixed-point principle, show that 'Tree' satisfies the idiom laws and that 'mirror' is an idiom homomorphism.*

3 Recursion and iteration

The combinator *recurse* captures *recursive* or *top-down* tree constructions; the functions f and g are repeatedly mapped over the whole tree:

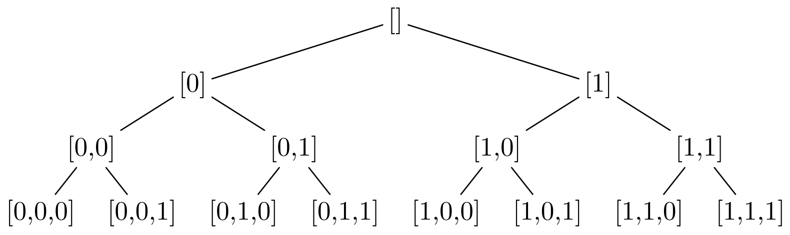
$$
\begin{array}{ll}
\mathit{recurse} & :: (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \mathit{Tree}\ \alpha) \\
\mathit{recurse}\ f\ g\ a = t\ \mathbf{where}\ t = \mathit{Node}\ a\ (\mathit{map}\ f\ t)\ (\mathit{map}\ g\ t). &
\end{array}
$$

Thus, an alternative definition of *bird* is $\mathit{recurse}\ (\mathit{recip} \circ \mathit{succ})\ (\mathit{succ} \circ \mathit{recip})\ 1$, where *recip* is the reciprocal function and *succ* is the successor function.

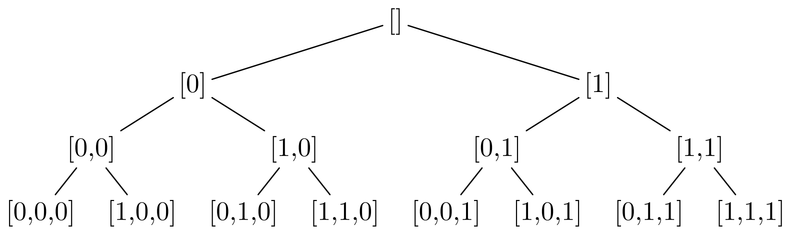
We can also construct a tree in an *iterative* or *bottom-up* fashion; the functions f and g are repeatedly applied to the given initial seed a:

$$
\begin{array}{ll}
\mathit{iterate} & :: (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \mathit{Tree}\ \alpha) \\
\mathit{iterate}\ f\ g\ a = \mathit{loop}\ a\ \mathbf{where}\ \mathit{loop}\ x = \mathit{Node}\ x\ (\mathit{loop}\ (f\ x))\ (\mathit{loop}\ (g\ x)). &
\end{array}
$$

The type α can be seen as a type of states and the infinite tree as an enumeration of the state space. One could argue that *iterate* is more natural than *recurse*. This intuition is backed up by the fact that $\mathit{map}\ h \circ \mathit{iterate}\ f\ g$ is the unfold of the *Tree* co-datatype.



(a) *recurse* ([0]++) ([1]++) []



(b) *iterate* ([0]++) ([1]++) []

Fig. 2. A tree that contains all bit strings and its bit-reversal permutation tree.

The goal of this section is to turn a recursive definition, such as the one for *bird*, into an iterative one, which can be executed manually to grow a tree. Before we tackle this problem, we note that both *recurse* and *iterate* satisfy a *fusion* law:

$$\begin{array}{rcl} \mathit{map}\ h \circ \mathit{recurse}\ f_1\ g_1 & = & \mathit{recurse}\ f_2\ g_2 \circ h \\ & \Uparrow & \\ h \circ f_1 = f_2 \circ h & \wedge & h \circ g_1 = g_2 \circ h \\ & \Downarrow & \\ \mathit{map}\ h \circ \mathit{iterate}\ f_1\ g_1 & = & \mathit{iterate}\ f_2\ g_2 \circ h. \end{array}$$

Exercise 2 *Prove the fusion laws, and then use fusion to give an alternative proof that* $1\ /\ \mathit{bird} = \mathit{mirror}\ \mathit{bird}$.

How are *recurse* $f\ g\ a$ and *iterate* $f\ g\ a$ related? Consider Figure 2, which displays the trees *recurse* ([0]++) ([1]++) [] and *iterate* ([0]++) ([1]++) []. Since f and g are applied in different orders – inside out and outside in – each level of *recurse* $f\ g\ a$ is the *bit-reversal permutation* of the corresponding level of *iterate* $f\ g\ a$. For brevity's sake, one tree is called the *bit-reversal permutation tree* of the other. Can we transform an instance of *recurse* into an instance of *iterate*? Yes, if the two functions are pre- or post-multiplications of elements of some given *monoid*. Let us introduce a suitable type class:

$$\begin{array}{l} \textbf{infixr}\ 5\ \cdot \\ \textbf{class}\ \mathit{Monoid}\ \alpha\ \textbf{where} \\ \quad \epsilon\ \ \ :: \alpha \\ \quad (\cdot) :: \alpha \to \alpha \to \alpha. \end{array}$$

The *recursion–iteration lemma* then states

$$recurse\ (a\cdot)\ (b\cdot)\ \epsilon \quad = \quad iterate\ (\cdot a)\ (\cdot b)\ \epsilon, \tag{3}$$

where a and b are elements of some monoid $(M, \cdot, \epsilon)$. To establish the lemma, we show that *iterate* $(\cdot a)$ $(\cdot b)$ ϵ satisfies the defining equation of *recurse* $(a\cdot)$ $(b\cdot)$ ϵ, that is $t = Node\ \epsilon\ (map\ (a\cdot)\ t)\ (map\ (b\cdot)\ t)$:

$$\begin{aligned}
& iterate\ (\cdot a)\ (\cdot b)\ \epsilon \\
= & \quad \{\ \text{definition of } iterate\ \} \\
& Node\ \epsilon\ (iterate\ (\cdot a)\ (\cdot b)\ (\epsilon \cdot a))\ (iterate\ (\cdot a)\ (\cdot b)\ (\epsilon \cdot b)) \\
= & \quad \{\ \epsilon \cdot x = x = x \cdot \epsilon\ \} \\
& Node\ \epsilon\ (iterate\ (\cdot a)\ (\cdot b)\ (a \cdot \epsilon))\ (iterate\ (\cdot a)\ (\cdot b)\ (b \cdot \epsilon)) \\
= & \quad \{\ \text{fusion: } (x\cdot) \circ (\cdot y) = (\cdot y) \circ (x\cdot)\ \} \\
& Node\ \epsilon\ (map\ (a\cdot)\ (iterate\ (\cdot a)\ (\cdot b)\ \epsilon))\ (map\ (b\cdot)\ (iterate\ (\cdot a)\ (\cdot b)\ \epsilon)).
\end{aligned}$$

At first sight, it seems that the applicability of the lemma is somewhat hampered by the requirement on the form of the two arguments. However, since *endomorphisms*, functions of type $\tau \to \tau$ for some τ, form a monoid, we can easily rewrite an arbitrary instance of *recurse* into the required form ($\diamond$ is function application below, the 'apply' of the identity idiom):

$$\begin{aligned}
& recurse\ (recip \circ succ)\ (succ \circ recip)\ 1 \\
= & \quad \{\ \text{fusion: } id \diamond x = x \text{ and } (\diamond x) \circ (f\ \circ) = f \circ (\diamond x)\ \} \\
& recurse\ (recip \circ succ\ \circ)\ (succ \circ recip\ \circ)\ id \diamond 1 \\
= & \quad \{\ \text{recursion-iteration lemma}\ \} \\
& iterate\ (\circ\ recip \circ succ)\ (\circ\ succ \circ recip)\ id \diamond 1.
\end{aligned}$$

Hooray, we have succeeded in transforming *bird* into an iterative form! Well, not quite; one could argue that using functions as the 'internal state' is cheating. Fortunately, we can provide a concrete representation of these functions by viewing a rational as a pair of numbers. To this end, we introduce a type of vectors:

$$\mathbf{data}\ Vector = \begin{pmatrix} Integer \\ Integer \end{pmatrix}; \qquad \mathbf{i} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The function *rat* maps the concrete to the abstract representation:

$$\begin{aligned}
& rat && :: Vector \to Rational \\
& rat \begin{pmatrix} a \\ b \end{pmatrix} && = a \div b,
\end{aligned}$$

where $\div$ constructs a rational from two integers.

Both *recip* and *succ* can be easily expressed as vector transformations. In fact, since they correspond to linear transformations, we can phrase them as matrix multiplications:

$$\mathbf{data}\ Matrix = \begin{pmatrix} Integer & Integer \\ Integer & Integer \end{pmatrix}.$$

We assume the standard vector and matrix operations and take the opportunity to introduce a handful of matrices that we need later on:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix};\ \mathbf{F} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix};\ \text{▟} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix};\ \text{▙} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix};\ \text{▜} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix};\ \text{▛} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Now, the concrete counterpart of *recip* is $(\mathbf{F}*)$ and that of *succ* is $(\text{▜}*)$. Here, $*$ is matrix multiplication. As an aside, $\mathbf{F}$ is mnemonic for *flip*, as $\mathbf{F} * \mathbf{X}$ flips $\mathbf{X}$ vertically, and $\mathbf{X} * \mathbf{F}$ flips $\mathbf{X}$ horizontally:

$$rat \circ (\mathbf{F}*) \quad = \quad recip \circ rat \tag{4}$$

$$rat \circ (\text{▜}*) \quad = \quad succ \circ rat \tag{5}$$

Since square matrices with matrix multiplication form a monoid, we can redo the derivation above in more concrete terms:

$$\begin{aligned}
& recurse\ (recip \circ succ)\ (succ \circ recip)\ 1 \\
= \quad & \{\ \text{fusion: } rat\ \mathbf{i} = 1\text{, (4) and (5)}\ \} \\
& map\ rat\ \left(recurse\ ((\mathbf{F}*) \circ (\text{▜}*))\ ((\text{▜}*) \circ (\mathbf{F}*))\ \mathbf{i}\right) \\
= \quad & \{\ (\mathbf{X}*) \circ (\mathbf{Y}*) = ((\mathbf{X} * \mathbf{Y})*),\ \mathbf{F} * \text{▜} = \text{▟} \text{ and } \text{▜} * \mathbf{F} = \text{▛}\ \} \\
& map\ rat\ \left(recurse\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{i}\right) \\
= \quad & \{\ \text{fusion: } \mathbf{I} * \mathbf{v} = \mathbf{v} \text{ and } (*\mathbf{v}) \circ (\mathbf{X}*) = (\mathbf{X}*) \circ (*\mathbf{v})\ \} \\
& map\ rat\ \left(map\ (*\mathbf{i})\ \left(recurse\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{I}\right)\right) \\
= \quad & \{\ \text{functor and define } mediant = rat \circ (*\mathbf{i})\ \} \\
& map\ mediant\ \left(recurse\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{I}\right) \\
= \quad & \{\ \text{recursion-iteration lemma}\ \} \\
& map\ mediant\ \left(iterate\ (*\text{▟})\ (*\text{▛})\ \mathbf{I}\right).
\end{aligned}$$

If we unfold the definition of *mediant*, we obtain

$$\begin{aligned}
& mediant && :: Matrix \to Rational \\
& mediant \begin{pmatrix} a & b \\ c & d \end{pmatrix} && = (a + b) \div (c + d)\,.
\end{aligned}$$

The rational ${}^{a+b}/_{c+d}$ is the so-called mediant of ${}^{a}/_{c}$ and ${}^{b}/_{d}$, hence the name of the function. The matrix $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$ can be seen as representing the interval $({}^{a}/_{c}, {}^{b}/_{d})$, which contains the mediant if ${}^{a}/_{c} \leqslant {}^{b}/_{d}$.

The iterative formulation of *bird* explains why the Fibonacci numbers appear on the two spines. The initial state is $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$; the state is updated as follows:

$$\begin{pmatrix} b & a+b \\ d & c+d \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \text{▟} \ \leftarrow\!\shortmid\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \text{▛} = \begin{pmatrix} a+b & a \\ c+d & c \end{pmatrix}.$$

Each row implements the iterative Fibonacci algorithm, which maintains two consecutive Fibonacci numbers. After n steps, we obtain

$$\text{▟}^{n} = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix} \qquad \text{▛}^{n} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix},$$

where F_n is the nth Fibonacci number with $F_{-n} = (-1)^{n+1} * F_n$.

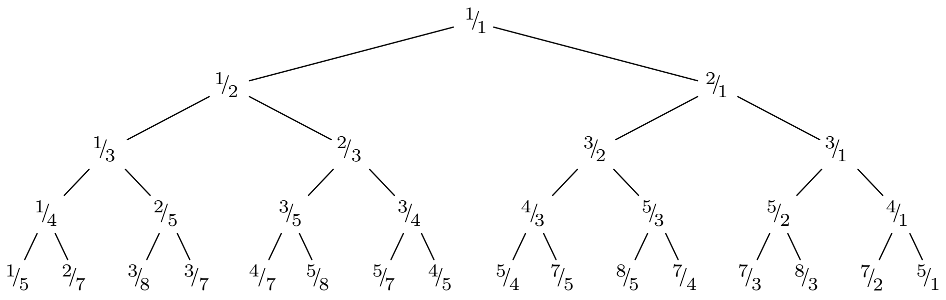


Fig. 3. The Stern-Brocot tree

Exercise 3 *Formalise the claim above. You may want to peek at Section 5 first, which introduces infinite lists.* (Hint: *define a function spine* :: *Tree* $\alpha \rightarrow$ *Stream* α *that maps a tree on to its left or right spine.*)

4 The Stern–Brocot tree

There are many ways to enumerate the positive rationals. Probably the oldest method was found around 1850 by the German mathematicians Eisenstein and Stern. It is deceptively simple: start with the two 'boundary rationals' $^0/_1$ and $^1/_0$, which are not included in the enumeration, and then repeatedly insert the mediant $^{a+b}/_{c+d}$ between two adjacent rationals $^a/_c$ and $^b/_d$.

Since the number of inserted rationals doubles with every step, the process can be pictured by an infinite binary tree, the so-called Stern–Brocot tree[3] (see Figure 3). Quite remarkably, each level shown is a permutation of the corresponding level of the Bird tree. The purpose of this section is to verify this observation, which implies that the Bird tree also contains every positive rational once.

Before we work out the relationship, let us turn the informal description of the Stern–Brocot tree into a program. This is most easily accomplished if we first construct a tree of intervals, represented by 2×2 matrices, and then map the intervals to their mediants. The start interval is now $\left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$; the interval is updated as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \text{▜} = \begin{pmatrix} a & a+b \\ c & c+d \end{pmatrix} \;\longleftarrow\!\!\shortmid\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \;\longmapsto\; \begin{pmatrix} a+b & b \\ c+d & d \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \text{▙}.$$

The left bound of the left descendent is the original left bound; the right bound is the mediant of the two original bounds. Likewise for the right descendent.

So the verbal description corresponds to an iterative construction, in which the state is an interval. Using a derivation inverse to the one in the preceding section, we can turn the verbal description into a compact co-recursive definition:

$$\begin{aligned} & \mathit{map\ mediant}\ \big(\mathit{iterate}\ ({*}\text{▜})\ ({*}\text{▙})\ \mathbf{F}\big) \\ = \quad & \{\ \text{fusion: } \mathbf{I} * \mathbf{F} = \mathbf{F},\ \mathbf{F} * \text{▜} = \text{▙} * \mathbf{F} \text{ and } \mathbf{F} * \text{▙} = \text{▜} * \mathbf{F}\ \} \\ & \mathit{map\ mediant}\ \big(\mathit{map}\ ({*}\mathbf{F})\ \big(\mathit{iterate}\ ({*}\text{▙})\ ({*}\text{▜})\ \mathbf{I}\big)\big) \end{aligned}$$

[3] The French clockmaker Brocot discovered the method around 1860, independent of Eisenstein and Stern.

$$
\begin{array}{ll}
= & \{\ \text{functor and } \mathit{mediant} \circ (*\mathbf{F}) = \mathit{mediant}\ \} \\
 & \mathit{map\ mediant}\ \big(\mathit{iterate}\ (*\text{▙})\ (*\text{▜})\ \mathbf{I}\big) \\
= & \{\ \text{recursion-iteration lemma}\ \} \\
 & \mathit{map\ mediant}\ \big(\mathit{recurse}\ (\text{▙}*)\ (\text{▜}*)\ \mathbf{I}\big) \\
= & \{\ \text{fusion: } \mathbf{I} * \mathbf{v} = \mathbf{v} \text{ and } (*\mathbf{v}) \circ (\mathbf{X}*) = (\mathbf{X}*) \circ (*\mathbf{v})\ \} \\
 & \mathit{map\ rat}\ \big(\mathit{recurse}\ (\text{▙}*)\ (\text{▜}*)\ \mathbf{i}\big) \\
= & \{\ \mathbf{F} * \text{▜} * \mathbf{F} = \text{▙}\ \} \\
 & \mathit{map\ rat}\ \big(\mathit{recurse}\ (\mathbf{F} * \text{▜} * \mathbf{F}*)\ (\text{▜}*)\ \mathbf{i}\big) \\
= & \{\ \text{fusion: } \mathit{rat}\ \mathbf{i} = 1,\ (4) \text{ and } (5)\ \} \\
 & \mathit{recurse}\ (\mathit{recip} \circ \mathit{succ} \circ \mathit{recip})\ \mathit{succ}\ 1.
\end{array}
$$

If we unfold the definitions, we obtain the following co-recursive program:

$$
\begin{array}{l}
\mathit{stern\text{-}brocot} :: \mathit{Tree\ Rational} \\
\mathit{stern\text{-}brocot} = \mathit{Node}\ 1\ (1\ /\ (1\ /\ \mathit{stern\text{-}brocot} + 1))\ (\mathit{stern\text{-}brocot} + 1).
\end{array}
$$

The definition is tantalisingly close to the definition of *bird*. As an aside, the derivation above also provides a formula for the bit-reversal permutation tree of *stern-brocot*, the so-called Calkin–Wilf or Eisenstein–Stern tree. We simply replace *recurse* by *iterate*, obtaining *iterate* (*recip* ∘ *succ* ∘ *recip*) *succ* 1.

We have already observed that *bird* and *stern-brocot* are level-wise permutations of each other. Looking a bit closer, we notice that the natural numbers are located on the right spine of the Stern–Brocot tree, whereas the Fibonacci fractions that approach the golden ratio ${}^{1}/_{1}$, ${}^{2}/_{1}$, ${}^{3}/_{2}$, ${}^{5}/_{3}$, ${}^{5}/_{8}$, etc. appear on a zigzag path. Recalling that it was the other way round in the Bird tree, we conjecture

$$
\begin{array}{lcl}
\mathit{odd\text{-}mirror\ bird} & = & \mathit{stern\text{-}brocot} \qquad (6) \\
\mathit{odd\text{-}mirror\ stern\text{-}brocot} & = & \mathit{bird}, \qquad (7)
\end{array}
$$

where *odd-mirror* swaps the immediate subtrees of a node but only on odd levels:

$$
\begin{array}{l}
\mathit{even\text{-}mirror}, \mathit{odd\text{-}mirror} \quad :: \mathit{Tree}\ \alpha \rightarrow \mathit{Tree}\ \alpha \\
\mathit{odd\text{-}mirror}\ \ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ a\ (\mathit{even\text{-}mirror}\ l)\ (\mathit{even\text{-}mirror}\ r) \\
\mathit{even\text{-}mirror}\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ a\ (\mathit{odd\text{-}mirror}\ \ r)\ (\mathit{odd\text{-}mirror}\ \ l).
\end{array}
$$

Since *odd-mirror* and *even-mirror* are involutions, it suffices to prove one of the equations above; we pick the first one (6). Let us introduce some shortcuts so that the expressions still fit on one line: We abbreviate *map recip* by *r*, *map* (*recip* ∘ *succ*) by *rs* and so forth. Furthermore, *e* is shorthand for *even-mirror bird* and likewise *o* for *odd-mirror bird*. Finally, we abbreviate *stern-brocot* by *sb*:

$$
\begin{array}{ll}
 & o \\
= & \{\ \text{definition of } \mathit{bird} \text{ and } \mathit{odd\text{-}mirror}, \text{ naturality of } \mathit{even\text{-}mirror}\ \} \\
 & \mathit{Node}\ 1\ (\mathit{rs}\ \ e)\ (\mathit{sr}\ \ e)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{definition of } \mathit{bird} \text{ and } \mathit{even\text{-}mirror}, \text{ naturality of } \mathit{odd\text{-}mirror} \} \\
&\quad \mathit{Node}\ 1\ (\mathit{rs}\ (\mathit{Node}\ 1\ (\mathit{sr}\ o)\ (\mathit{rs}\ o)))\ (\mathit{sr}\ (\mathit{Node}\ 1\ (\mathit{sr}\ o)\ (\mathit{rs}\ o))) \\
&= \quad \{ \text{definition of } \mathit{rs} \text{ and } \mathit{sr} \} \\
&\quad \mathit{Node}\ 1\ (\mathit{Node}\ {}^1\!/_2\ (\mathit{rssr}\ o)\ (\mathit{rsrs}\ o))\ (\mathit{Node}\ {}^2\!/_1\ (\mathit{srsr}\ o)\ (\mathit{srrs}\ o)) \\
&\subset \quad \{ x = \mathit{Node}\ 1\ (\mathit{Node}\ {}^1\!/_2\ (\mathit{rssr}\ x)\ (\mathit{rsrs}\ x))\ (\mathit{Node}\ {}^2\!/_1\ (\mathit{srsr}\ x)\ (\mathit{srrs}\ x)) \} \\
&\quad \mathit{Node}\ 1\ (\mathit{Node}\ {}^1\!/_2\ (\mathit{rssr}\ \mathit{sb})\ (\mathit{rsrs}\ \mathit{sb}))\ (\mathit{Node}\ {}^2\!/_1\ (\mathit{srsr}\ \mathit{sb})\ (\mathit{srrs}\ \mathit{sb})) \\
&= \quad \{ \mathit{recip} \text{ is an involution: } \mathit{rssr} = \mathit{rsrrsr} \text{ and } \mathit{srrs} = \mathit{ss} \} \\
&\quad \mathit{Node}\ 1\ (\mathit{Node}\ {}^1\!/_2\ (\mathit{rsrrsr}\ \mathit{sb})\ (\mathit{rsrs}\ \mathit{sb}))\ (\mathit{Node}\ {}^2\!/_1\ (\mathit{srsr}\ \mathit{sb})\ (\mathit{ss}\ \mathit{sb})) \\
&= \quad \{ \text{definition of } \mathit{rsr} \text{ and } s \} \\
&\quad \mathit{Node}\ 1\ (\mathit{rsr}\ (\mathit{Node}\ 1\ (\mathit{rsr}\ \mathit{sb})\ (s\ \mathit{sb})))\ (s\ (\mathit{Node}\ 1\ (\mathit{rsr}\ \mathit{sb})\ (s\ \mathit{sb}))) \\
&= \quad \{ \text{definition of } \mathit{sb} \} \\
&\quad \mathit{Node}\ 1\ (\mathit{rsr}\ \mathit{sb})\ (s\ \mathit{sb}) \\
&= \quad \{ \text{definition of } \mathit{sb} \} \\
&\quad \mathit{sb}.
\end{aligned}
$$

5 Linearising the Stern–Brocot tree

Now, let us consider linearising the Bird tree. As a warm-up exercise, this section demonstrates how to linearise the Stern–Brocot tree. This has been done several times before (Gibbons *et al.* 2006; Backhouse & Ferreira 2008), but we believe that the co-data framework is particularly suited to this task, so it is worthwhile to repeat the exercise. Technically, we aim to derive a *loopless algorithm* (Bird 2006) for enumerating the elements of *stern-brocot*. An enumeration is called *loopless* if the next element is computed from the previous one in constant time and, for this pearl, in constant space.

Since we have to transform an infinite tree into an infinite list, let us introduce a tailor-made co-datatype for the latter (Hinze 2008):

```
data Stream α = Cons {head :: α, tail :: Stream α}

infixr 5 ≺
(≺)     :: α → Stream α → Stream α
a ≺ s = Cons a s.
```

The type of streams is similar to Haskell's predefined type of lists, except that there is no empty list constructor, so we cannot form a finite list.

Like infinite trees, streams are an idiom, which means that we can effortlessly lift functions to streams:

```
instance Idiom Stream where
  pure a = s where s = a ≺ s
  s ⋄ t    = (head s) (head t) ≺ (tail s) ⋄ (tail t).
```

Like infinite trees, streams can be built recursively or iteratively. We overload *recurse* and *iterate* to also denote the combinators on streams:

$$\begin{aligned}
&\mathit{recurse} && :: (\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\ \alpha)\\
&\mathit{recurse}\ f\ a = s\ \mathbf{where}\ s = a \prec \mathit{map}\ f\ s\\[1ex]
&\mathit{iterate} && :: (\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\ \alpha)\\
&\mathit{iterate}\ f\ a && = a \prec \mathit{iterate}\ f\ (f\ a).
\end{aligned}$$

Unlike the tree combinators, *recurse* and *iterate* construct exactly the same stream: we have $\mathit{recurse}\ f\ a = \mathit{iterate}\ f\ a$.

Exercise 4 *Show that iterate f a satisfies the recursion equation of recurse f a.* (Hint: *Formulate a fusion law first.*)

To convert a tree to a stream, we define a helper function that chops the root off a tree:

$$\begin{aligned}
&\mathit{stream} && :: \mathit{Tree}\ \alpha \to \mathit{Stream}\ \alpha\\
&\mathit{stream}\ t = \mathit{root}\ t \prec \mathit{stream}\ (\mathit{chop}\ t)\\[1ex]
&\mathit{chop} && :: \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha\\
&\mathit{chop}\ t = \mathit{Node}\ (\mathit{root}\ (\mathit{left}\ t))\ (\mathit{right}\ t)\ (\mathit{chop}\ (\mathit{left}\ t)).
\end{aligned}$$

In a sense, *root* is the counterpart of *head* and *chop* is the counterpart of *tail*. Infinite trees and streams are both very regular structures, so it probably comes as little surprise that *stream* is an idiom isomorphism.

Exercise 5 *Show that 'stream' is an idiom isomorphism and that 'chop' is an idiom homomorphism.*

Let's have a closer look at the workings of *stream*: it outputs the elements of its argument tree level by level. However, since *chop* repeatedly swaps the left and the right subtree, each level is output in *bit-reversal permutation* order. In other words, *stream stern-brocot* actually linearises the Calkin–Wilf tree. We return to this point later on. The enumeration *stream t* is not loopless: to produce the next element, *stream t* takes time linear in the depth of the element in the tree and space proportional to the width of the current level. So turning *stream stern-brocot* into a loopless algorithm requires some effort.

As a first step towards this goal, let us disentangle *stern-brocot* into a tree of numerators and denominators. Given the specification

$$\mathit{num} \div \mathit{den} \quad = \quad \mathit{stern\text{-}brocot}, \tag{8}$$

where $\div$ is lifted to trees, we reason as follows:

$$\begin{aligned}
&\mathit{num} \div \mathit{den}\\
=\ &\quad \{\ \text{specification and definition of } \mathit{stern\text{-}brocot}\ \}\\
&\mathit{Node}\ 1\ (1\ /\ (1\ /\ (\mathit{num} \div \mathit{den}) + 1))\ (\mathit{num} \div \mathit{den} + 1)\\
=\ &\quad \{\ \text{arithmetic}\ \}\\
&\mathit{Node}\ (1 \div 1)\ (\mathit{num} \div (\mathit{num} + \mathit{den}))\ ((\mathit{num} + \mathit{den}) \div \mathit{den})
\end{aligned}$$

$$\begin{aligned} &= \quad \{ \text{ definition of } \div \text{ lifted to trees } \} \\ &\quad (\mathit{Node}\ 1\ \mathit{num}\ (\mathit{num} + \mathit{den})) \div (\mathit{Node}\ 1\ (\mathit{num} + \mathit{den})\ \mathit{den}). \end{aligned}$$

Thus, *num* and *den* defined by

$$\begin{aligned} \mathit{num} &= \mathit{Node}\ 1\ \mathit{num}\ (\mathit{num} + \mathit{den}) \\ \mathit{den} &= \mathit{Node}\ 1\ (\mathit{num} + \mathit{den})\ \mathit{den} \end{aligned}$$

satisfy the specification. The two definitions are pleasingly symmetric; in fact, we have $\mathit{den} = \mathit{chop}\ \mathit{num}$. In other words, we can confine ourselves to linearising *den*; that is we seek to express *chop den* in terms of *den* and possibly *num*. To see what we are aiming for, let us unroll *chop den*:

$$\mathit{chop}\ \mathit{den} = \mathit{Node}\ 2\ \mathit{den}\ (\mathit{den} + \mathit{chop}\ \mathit{den}).$$

This is almost the sum of *num* and *den*:

$$\mathit{num} + \mathit{den} - \mathit{chop}\ \mathit{den} = \mathit{Node}\ 0\ (2 * \mathit{num})\ (\mathit{num} + \mathit{den} - \mathit{chop}\ \mathit{den})\ .$$

The difference between $\mathit{num} + \mathit{den}$ and *chop den* equals $2 * x$, where x is the solution of $x = \mathit{Node}\ 0\ \mathit{num}\ x$. Have we made any progress? Somewhat surprisingly, the answer is yes. By a stroke of good luck, the unique solution x of the equation above is *num* **mod** *den*, as a quick calculation shows:

$$\begin{aligned} &\mathit{num}\ \mathbf{mod}\ \mathit{den} \\ = &\quad \{ \text{ definition of } \mathit{num} \text{ and definition of } \mathit{den} \ \} \\ &(\mathit{Node}\ 1\ \mathit{num}\ (\mathit{num} + \mathit{den}))\ \mathbf{mod}\ (\mathit{Node}\ 1\ (\mathit{num} + \mathit{den})\ \mathit{den}) \\ = &\quad \{ \text{ definition of } \mathbf{mod} \text{ lifted to trees } \} \\ &\mathit{Node}\ (1\ \mathbf{mod}\ 1)\ (\mathit{num}\ \mathbf{mod}\ (\mathit{num} + \mathit{den}))\ ((\mathit{num} + \mathit{den})\ \mathbf{mod}\ \mathit{den}) \\ = &\quad \{ \text{ properties of } \mathbf{mod} \ \} \\ &\mathit{Node}\ 0\ \mathit{num}\ (\mathit{num}\ \mathbf{mod}\ \mathit{den})\ . \end{aligned}$$

As an intermediate summary, we have derived

$$\begin{aligned} \mathit{fusc} &= 1 \prec \mathit{fusc}' \\ \mathit{fusc}' &= 1 \prec \mathit{fusc} + \mathit{fusc}' - 2 * (\mathit{fusc}\ \mathbf{mod}\ \mathit{fusc}'), \end{aligned}$$

where $\mathit{fusc} = \mathit{stream}\ \mathit{num}$ is the stream of numerators and $\mathit{fusc}' = \mathit{stream}\ \mathit{den} = \mathit{tail}\ \mathit{fusc}$ is the stream of denominators. (The name *fusc* is due to Dijkstra (1976).) Both streams are given by recursive definitions. It is a simple exercise to turn them into iterative definitions. Tupling *fusc* and *fusc′* using $(\star) = \mathit{zip}\ (,)$, we obtain

$$\begin{aligned} &\mathit{fusc} \star \mathit{fusc}' \\ = &\quad \{ \text{ definition of } \mathit{fusc} \text{ and definition of } \mathit{fusc}' \ \} \\ &(1 \prec \mathit{fusc}') \star (1 \prec \mathit{fusc} + \mathit{fusc}' - 2 * (\mathit{fusc}\ \mathbf{mod}\ \mathit{fusc}')) \\ = &\quad \{ \text{ definition of } \star \text{ and definition of } \mathit{zip} \ \} \\ &(1, 1) \prec \mathit{fusc}' \star (\mathit{fusc} + \mathit{fusc}' - 2 * (\mathit{fusc}\ \mathbf{mod}\ \mathit{fusc}')) \end{aligned}$$

$$
\begin{array}{ll}
= & \{\ \text{idioms, and introduce } \mathit{step}\ (n,d) = (d, n + d - 2 * (n\ \mathbf{mod}\ d))\ \} \\
 & (1,1) \prec \mathit{map}\ \mathit{step}\ (\mathit{fusc} \star \mathit{fusc}').
\end{array}
$$

Since *iterate f a* is the unique solution of the recursion equation $x = a \prec \mathit{map}\ f\ x$, we have $\mathit{fusc} \star \mathit{fusc}' = \mathit{iterate}\ \mathit{step}\ (1,1)$. The following calculations summarise our findings:

$$
\begin{array}{ll}
 & \mathit{stream}\ \mathit{stern\text{-}brocot} \\
= & \quad \{\ \text{see above}\ \} \\
 & \mathit{stream}\ (\mathit{num} \div \mathit{den}) \\
= & \quad \{\ \mathit{stream} \text{ is an idiom homomorphism}\ \} \\
 & \mathit{stream}\ \mathit{num} \div \mathit{stream}\ \mathit{den} \\
= & \quad \{\ \text{see above}\ \} \\
 & \mathit{fusc} \div \mathit{fusc}' \\
= & \quad \{\ \text{idioms, and introduce } \mathit{rat}'\ (n,d) = n \div d\ \} \\
 & \mathit{map}\ \mathit{rat}'\ (\mathit{fusc} \star \mathit{fusc}') \\
= & \quad \{\ \text{see above}\ \} \\
 & \mathit{map}\ \mathit{rat}'\ (\mathit{iterate}\ \mathit{step}\ (1,1)).
\end{array}
$$

As a final step, we can additionally fuse rat' with *iterate*, employing the following formula:

$$1 \,/\, (\lfloor n \div d \rfloor + 1 - \{n \div d\}) = d \div (n + d - 2 * (n\ \mathbf{mod}\ d)).$$

Here, $\lfloor r \rfloor$ denotes the integral part of r and $\{r\}$ its fractional part, such that $r = \lfloor r \rfloor + \{r\}$. Continuing the calculation, we obtain

$$
\begin{array}{ll}
= & \quad \{\ \text{fusion, and introduce } \mathit{next}\ r = 1 \,/\, (\lfloor r \rfloor + 1 - \{r\})\ \} \\
 & \mathit{iterate}\ \mathit{next}\ 1.
\end{array}
$$

This intriguing algorithm is attributed to Newman (Aigner & Ziegler 2004); the formula for *step* is apparently due to Stay (Sloane 2009; sequence A002487).

Can we derive a similar algorithm for *stream bird*? The answer is probably no. The next section explains why.

6 Linearising the Bird tree and some others

Now that we have warmed up, let's become more ambitious: the goal of this section is to derive a *loopless algorithm* for enumerating the elements of the infinite tree $ab = \mathit{recurse}\ (a\cdot)\ (b\cdot)\ \epsilon$, where a and b are elements of some given monoid. Unfortunately, we will not quite achieve our goal: the step function will run in *amortised* constant time, based on the assumption that the monoid operation '·' runs in constant time. Or put differently, it will use a constant number of '·' operations amortised over time.

Now, the call *stream ab* yields

$$\epsilon \prec a \prec b \prec a \cdot a \prec b \cdot a \prec a \cdot b \prec b \cdot b \prec a \cdot a \cdot a \prec \cdots .$$

On the face of it, calculating the next element in *stream ab* corresponds to the binary increment: $b^i \cdot a \cdot w$ becomes $a^i \cdot b \cdot w$, and b^i becomes a^{i+1}.

In contrast to the binary increment, we can't examine the elements, since we are not working with the free monoid – after all, the elements a and b could be functions. Of course, if we don't make any additional assumptions about the underlying structure, then we simply can't calculate the next from the previous element. In order to support incremental computations, we assume that each element has an inverse; that is we are working with a group rather than a monoid:

$$\begin{array}{l} \textbf{class } (\mathit{Monoid}\ \alpha) \Rightarrow \mathit{Group}\ \alpha \textbf{ where} \\ \quad \begin{array}{ll} \mathit{inverse} & :: \alpha \to \alpha \\ (\hat{}) & :: \alpha \to \mathit{Integer} \to \alpha. \end{array} \end{array}$$

The class *Group* additionally supports exponentiation, which we assume defaults to a logarithmic implementation. We abbreviate $a\,\hat{}\,n$ by a^n; in particular, $\mathit{inverse}\ a = a^{-1}$.

Now, to get from $b^i \cdot a \cdot w$ to $a^i \cdot b \cdot w$, we simply pre-multiply the former element with $(a^i \cdot b) \cdot (b^i \cdot a)^{-1}$. If the current element is b^i, then we cannot reuse any information, and we compute a^{i+1} afresh. Still, there is no way to inspect the elements, so we have to maintain some information: the number i of leading bs and whether the current element contains only bs. It turns out that the calculations are slightly more attractive, if we maintain the number of leading as of the *next* element. Given this information, the next element can be computed as follows:

$$\begin{array}{lll} \langle c, i, x \rangle & \mid c & = a^i \\ & \mid \mathit{otherwise} & = a^i \cdot b \cdot a^{-1} \cdot b^{-i} \cdot x. \end{array}$$

The required pieces of information can be easily defined using infinite trees:

$$\begin{array}{ll} \mathit{rim} & = \mathit{Node}\ \mathit{True}\ (\mathit{pure}\ \mathit{False})\ \mathit{rim} \\ \mathit{carry}' & = \mathit{Node}\ 1\ 0\ (1 + \mathit{carry}'). \end{array}$$

Abbreviating $\mathit{map}\ (x\cdot)\ s$ by $x \cdot s$, we have $ab = \mathit{Node}\ \epsilon\ (a \cdot ab)\ (b \cdot ab)$. Only the elements on the right spine of *ab* contain only bs. Consequently, *rim*'s right spine is labelled with *True*s, and all the other elements are *False*. To motivate the definition of *carry′*, let's unfold *chop ab*:

$$\begin{array}{ll} & \mathit{chop}\ ab \\ = & \quad \{ \text{ definition of } \mathit{chop} \} \\ & \mathit{Node}\ (\mathit{root}\ (\mathit{left}\ ab))\ (\mathit{right}\ ab)\ (\mathit{chop}\ (\mathit{left}\ ab)) \\ = & \quad \{ \text{ definition of } ab \} \\ & \mathit{Node}\ a\ (b \cdot ab)\ (\mathit{chop}\ (a \cdot ab)) \\ = & \quad \{ \mathit{chop} \text{ is an idiom homomorphism} \} \\ & \mathit{Node}\ a\ (b \cdot ab)\ (a \cdot \mathit{chop}\ ab). \end{array}$$

The root contains one leading a, the left subtree none and each element of the right subtree one more than the corresponding element in the entire tree. The definition of *carry′* exactly captures this description.

Lifting the ternary operation $\langle -, -, - \rangle$ to infinite trees, we claim that $\langle rim, carry', ab \rangle = chop\ ab$. The proof makes essential use of the *shift* property

$$\langle c, i+1, x \rangle \quad = \quad a \cdot \langle c, i, b^{-1} \cdot x \rangle, \tag{9}$$

which expresses the straightforward fact that we can pull an a to the front if the next element has at least one leading a. Turning to the proof, we show that $\langle rim, carry', ab \rangle$ satisfies the same recursion equation as $chop\ as$, namely $x = Node\ a\ (b \cdot ab)\ (a \cdot x)$:

$$\begin{array}{ll}
& \langle rim, carry', ab \rangle \\
= & \quad \{ \text{definition of } rim \text{, definition of } carry' \text{ and definition of } ab \} \\
& \langle Node\ True\ (pure\ False)\ rim, Node\ 1\ 0\ (carry' + 1), Node\ \epsilon\ (a \cdot ab)\ (b \cdot ab) \rangle \\
= & \quad \{ \text{definition of } \langle -, -, - \rangle \text{ lifted to trees} \} \\
& Node\ \langle True, 1, \epsilon \rangle\ \langle pure\ False, 0, a \cdot ab \rangle\ \langle rim, carry' + 1, b \cdot ab \rangle \\
= & \quad \{ \text{definition of } \langle -, -, - \rangle \text{ and (9)} \} \\
& Node\ a\ (b \cdot a^{-1} \cdot a \cdot ab)\ (a \cdot \langle rim, carry', b^{-1} \cdot b \cdot ab \rangle) \\
= & \quad \{ \text{inverses} \} \\
& Node\ a\ (b \cdot ab)\ (a \cdot \langle rim, carry', ab \rangle.
\end{array}$$

So we have reduced the problem of linearising ab to the problem of linearising rim and $carry'$. Are we any better off? Certainly, rim isn't difficult to enumerate, but what about $carry'$? By a second stroke of good luck, there is an intriguing cross-connection to the Stern–Brocot tree: $carry' = \lfloor stern\text{-}brocot \rfloor = \lfloor num\ /\ den \rfloor = num\ \mathbf{div}\ den$. Here is the straightforward proof:

$$\begin{array}{ll}
& num\ \mathbf{div}\ den \\
= & \quad \{ \text{definition of } num \text{ and definition of } den \} \\
& (Node\ 1\ num\ (num + den))\ \mathbf{div}\ (Node\ 1\ (num + den)\ den) \\
= & \quad \{ \text{definition of } \mathbf{div} \text{ lifted to trees} \} \\
& Node\ (1\ \mathbf{div}\ 1)\ (num\ \mathbf{div}\ (num + den))\ ((num + den)\ \mathbf{div}\ den) \\
= & \quad \{ \text{definition of } \mathbf{div} \text{ and } num \geqslant 1 \leqslant den \} \\
& Node\ 1\ 0\ ((num\ \mathbf{div}\ den) + 1).
\end{array}$$

Exercise 6 *Show that* $rim = den == 1$, *where* $==$ *is equality lifted to trees, that is* $==$ *has type* $(Eq\ \alpha) \Rightarrow Tree\ \alpha \to Tree\ \alpha \to Tree\ Bool$.

In other words, we can reuse the results of the previous section to solve the more general problem of turning ab into a stream:

$$\begin{array}{ll}
& stream\ ab \\
= & \quad \{ \text{definition of } stream \} \\
& root\ ab \prec stream\ (chop\ ab) \\
= & \quad \{ \text{see above and Exercise 6} \}
\end{array}$$

$$\begin{aligned}
&\epsilon \prec \mathit{stream}\ \langle \mathit{den} == 1,\ \mathit{num}\ \mathbf{div}\ \mathit{den},\ \mathit{ab}\rangle\\
=\quad &\quad \{\ \text{introduce } \otimes \text{ with } (n,d) \otimes x = \langle d == 1,\ n\ \mathbf{div}\ d,\ x\rangle\ \}\\
&\epsilon \prec \mathit{stream}\ ((\mathit{num} \star \mathit{den}) \otimes \mathit{ab})\\
=\quad &\quad \{\ \mathit{stream} \text{ is an idiom homomorphism}\ \}\\
&\epsilon \prec \mathit{stream}\ (\mathit{num} \star \mathit{den}) \otimes \mathit{stream}\ \mathit{ab}\\
=\quad &\quad \{\ \text{Section 5}\ \}\\
&\epsilon \prec \mathit{iterate}\ \mathit{step}\ (1,1) \otimes \mathit{stream}\ \mathit{ab}.
\end{aligned}$$

All that is left to do is to express $\mathit{stream}\ \mathit{ab} = \epsilon \prec \mathit{iterate}\ \mathit{step}\ (1,1) \otimes \mathit{stream}\ \mathit{ab}$ as an iteration. This is easy to achieve using tupling. Let $q = (1,1)$; then

$$\begin{aligned}
&\mathit{iterate}\ \mathit{step}\ q \star \mathit{stream}\ \mathit{ab}\\
=\quad &\quad \{\ \text{definition of } \mathit{iterate} \text{ and property of } \mathit{stream}\ \mathit{ab}\ \}\\
&(q \prec \mathit{map}\ \mathit{step}\ (\mathit{iterate}\ \mathit{step}\ q)) \star (\epsilon \prec \mathit{iterate}\ \mathit{step}\ q \otimes \mathit{stream}\ \mathit{ab})\\
=\quad &\quad \{\ \text{definition of } \star \text{ and definition of } \mathit{zip}\ \}\\
&(q,\epsilon) \prec \mathit{map}\ \mathit{step}\ (\mathit{iterate}\ \mathit{step}\ q) \star (\mathit{iterate}\ \mathit{step}\ q \otimes \mathit{stream}\ \mathit{ab})\\
=\quad &\quad \{\ \text{idioms and introduce } \mathit{step}'\ (x,y) = (\mathit{step}\ x, x \otimes y)\ \}\\
&(q,\epsilon) \prec \mathit{map}\ \mathit{step}'\ (\mathit{iterate}\ \mathit{step}\ q \star \mathit{stream}\ \mathit{ab}).
\end{aligned}$$

Recalling that $\mathit{iterate}\ f\ e$ is the unique solution of the equation $x = e \prec \mathit{map}\ f\ x$, we have established that $\mathit{iterate}\ \mathit{step}\ q \star \mathit{stream}\ \mathit{ab} = \mathit{iterate}\ \mathit{step}'\ (q,\epsilon)$ and furthermore that $\mathit{stream}\ \mathit{ab} = \mathit{map}\ \mathit{snd}\ (\mathit{iterate}\ \mathit{step}'\ (q,\epsilon))$.

The following definition summarises the derivation – we have additionally inlined the definitions and flattened the nested pair (q,ϵ) to a triple:

```
loopless     :: (Group α) ⇒ α → α → Stream α
loopless a b = map (λ(x, y, z) → z) (iterate step3 (1, 1, ε))
  where
  step3 (n, d, x) = (d, n + d − 2 ∗ m, x′)
    where (i, m) = divMod n d
          x′ | d == 1     = $a^i$
             | otherwise = $a^i \cdot b \cdot a^{-1} \cdot b^{-i} \cdot x$.
```

Assuming that the operation '$\cdot$' has a constant running time, the function *step3* takes $\Theta(\log\log(n+1))$ steps to produce the $(n+1)$st element from the nth element: the exponent i in the definition of *step3* is at most $\lceil \lg(n+1)\rceil$, and fast exponentiation uses at most $2\lceil \lg(i+1) - 1\rceil$ multiplications. The *amortised running time* of *step3* would be, however, constant, even if exponentiation were implemented naively; *step3* would then perform the same number of steps as the binary increment.

Linearising the Bird tree is now simply a matter of applying *loopless*. First of all, the set of all invertible square matrices forms a group, the so-called general linear group $GL_n(\mathbb{R})$ – if the coefficients are drawn from $\mathbb{R}$. Since ▟ and ▛ have the determinant -1, they are both invertible in $GL_2(\mathbb{Z})$, and we have

$$\begin{aligned}
& \mathit{stream}\ \mathit{bird} \\
=\ & \quad \{\ \text{Section 3}\ \} \\
& \mathit{stream}\ (\mathit{map}\ \mathit{mediant}\ (\mathit{recurse}\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{I})) \\
=\ & \quad \{\ \mathit{stream}\ \text{is an idiom homomorphism}\ \} \\
& \mathit{map}\ \mathit{mediant}\ (\mathit{stream}\ (\mathit{recurse}\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{I})) \\
=\ & \quad \{\ \text{see above}\ \} \\
& \mathit{map}\ \mathit{mediant}\ (\mathit{loopless}\ \text{▟}\ \text{▛})\ .
\end{aligned}$$

Done! Well, not quite: because of the way *stream* is defined, the program above actually enumerates the elements of the bit-reversal permutation tree of *bird*. We should really linearise *recurse* $(*\text{▟})\ (*\text{▛})\ \mathbf{I}$ instead of *recurse* $(\text{▟}*)\ (\text{▛}*)\ \mathbf{I}$. Of course, *loopless* can be adapted to work with pre- instead of post-multiplications, but fortunately, there is a more modular approach. Using matrix transposition $(-)^{\top}$ we can put *recurse* $(*\text{▟})\ (*\text{▛})\ \mathbf{I}$ into the required form:

$$\begin{aligned}
& \mathit{stream}\ (\mathit{map}\ \mathit{mediant}\ (\mathit{recurse}\ (*\text{▟})\ (*\text{▛})\ \mathbf{I})) \\
=\ & \quad \{\ \mathbf{I}^{\top} = \mathbf{I},\ \text{▟}^{\top} = \text{▟}\ \text{and}\ \text{▛}^{\top} = \text{▛}\ \} \\
& \mathit{stream}\ (\mathit{map}\ \mathit{mediant}\ (\mathit{recurse}\ (*\text{▟}^{\top})\ (*\text{▛}^{\top})\ \mathbf{I}^{\top})) \\
=\ & \quad \{\ \text{fusion: } (\mathbf{X} * \mathbf{Y})^{\top} = \mathbf{Y}^{\top} * \mathbf{X}^{\top}\ \} \\
& \mathit{stream}\ (\mathit{map}\ \mathit{mediant}\ (\mathit{map}\ \mathit{transpose}\ (\mathit{recurse}\ (\text{▟}*)\ (\text{▛}*)\ \mathbf{I}))) \\
=\ & \quad \{\ \text{functor and see above}\ \} \\
& \mathit{map}\ (\mathit{mediant} \circ \mathit{transpose})\ (\mathit{loopless}\ \text{▟}\ \text{▛}).
\end{aligned}$$

Can we improve the running time of the final program? If we managed to determine $\mathbf{X}^i$ in constant time, then *loopless* could be turned into a true loopless algorithm. Recall the findings of Section 3:

$$\text{▟}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix} \quad \text{▙}^n = \begin{pmatrix} 1 & 0 \\ n & 1 \end{pmatrix} \quad \text{▜}^n = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \quad \text{▛}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

The Stern–Brocot tree and the Calkin–Wilf tree can indeed be enumerated looplessly, as both involve only ▙ and ▜ – Backhouse and Ferreira derive these special cases in (2008). However, since we can't compute the Fibonacci numbers in constant time, this doesn't work for the Bird tree. Indeed, the fastest algorithm for computing F_n involves calculating ▟^n using fast exponentiation.

Acknowledgments

Thanks are due to Roland Backhouse, Jeremy Gibbons, Daniel James and Tom Harper for carefully proofreading a previous version of this paper. I am furthermore grateful to the anonymous referees for many helpful suggestions, in particular, for insisting on a roadmap. Finally, I would like to thank Richard for many years of inspiration and support.

References

Aigner, M. & Ziegler, G. M. (2004) *Proofs from THE BOOK*, 3rd edn. Springer-Verlag.

Backhouse, R. & Ferreira, J. F. (2008) Recounting the rationals: Twice!, In *The 9th International Conference on Mathematics of Program Construction (MPC '08)*, Audebaud, P. & Paulin-Mohring, C. (eds), Lecture Notes in Computer Science, vol. 5133. Springer, pp. 79–91.

Bird, R. S. (2006) Loopless functional algorithms. In *The 8th International Conference on Mathematics of Program Construction (MPC '06)*, Uustalu, T. (ed.), Lecture Notes in Computer Science, vol. 4014. Springer, pp. 90–114.

Calkin, N. & Wilf, H. (2000) Recounting the rationals, *Am. Math. Monthly*, 107 (4): 360–363.

Dijkstra, E. W. (1976) EWD 570: An exercise for Dr. R. M. Burstall. In *Selected Writings on Computing: A Personal Perspective*, Dijkstra, E. W. Springer, pp. 215–216. ISBN 0–387–90652–5.

Gibbons, J., Lester, D. & Bird, R. (2006) Functional pearl: Enumerating the rationals, *J. Funct. Program.*, 16 (3): 281–291.

Graham, R. L., Knuth, D. E. & Patashnik, O. (1994) *Concrete Mathematics*, 2nd ed. Addison-Wesley.

Hinze, R. (2008) Functional pearl: Streams and unique fixed points. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, Thiemann, P. (ed.). ACM Press, pp. 189–200.

McBride, C. & Paterson, R. (2008) Functional pearl: Applicative programming with effects, *J. Funct. Program.*, 18 (1): 1–13.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.

Rutten, J. (2003) Fundamental study: Behavioural differential equations: A coinductive calculus of streams, automata, and power series, *Theoret. Comp. Sci.*, 308: 1–53.

Sloane, N. J. A. (2009) The on-line encyclopedia of integer sequences [online]. Available at: `http://www.research.att.com/~njas/sequences/` (Accessed 17 July 2009).