5 Infinite data structures

This practical is based on my paper *Enumerating the Rationals* (DOI 10.1017/S0956796806005880). You'll find all the answers there—but I suggest you don't look at the paper until you have had a try yourself at these exercises.

We'll use the Haskell library for rational numbers:

$$\textbf{import}\ \mathit{Data.Ratio}$$

The only thing you need to know from here is that the binary operator % constructs a rational; for example, $22\,\%\,7$ is a rational approximation to π.

We'll also use the Prelude function *iterate*, which can generate an infinite list:

$$\begin{array}{l} \mathit{iterate} :: (a \to a) \to a \to [\,a\,] \quad \text{-- generates a stream} \\ \mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x) \end{array}$$

For example,

$$\mathit{take}\ 10\ (\mathit{iterate}\ (2\times)\ 1) = [\,1,2,4,8,16,32,64,128,256,512\,]$$

1. Recall the definition of *unfoldr* in the lecture:

   $$\begin{array}{l} \mathit{unfoldr} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to b \to [\,a\,] \\ \mathit{unfoldr}\ p\ f\ g\ z = \textbf{if}\ p\ z\ \textbf{then}\ [\,]\ \textbf{else}\ f\ z : \mathit{unfoldr}\ p\ f\ g\ (g\ z) \end{array}$$

   Give an alternative definition of *iterate* in terms of *unfoldr*.

2. The first thing you might try to generate all the rationals is with a nested list comprehension:

   $$\begin{array}{l} \mathit{rats}_1 :: [\,\mathit{Rational}\,] \quad \text{-- a stream} \\ \mathit{rats}_1 = [\,a\,\%\,b \mid a \leftarrow [\,1\,..\,], b \leftarrow [\,1\,..\,]\,] \end{array}$$

   (We'll use the word "stream" for a list that is known to be infinite.) Why doesn't this work?

3. The next try is to be more careful about the nested list of lists:

   $$\begin{array}{l} \mathit{rats}_2 :: [\,\mathit{Rational}\,] \quad \text{-- a stream} \\ \mathit{rats}_2 = \mathit{diag}\,[\,[\,a\,\%\,b \mid b \leftarrow [\,1\,..\,]\,] \mid a \leftarrow [\,1\,..\,]\,] \end{array}$$

This requires a function *diag* to convert an infinite lists of infinite lists into a single infinite list.

$$diag :: [[a]] \to [a] \quad \text{-- stream of streams to stream}$$

The trick is that although each row and each column is infinite, each *minor diagonal* is finite. The first minor diagonal consists of just the first element of the first row; the second diagonal has the second element of the first row and the first element of the second row; in general, the kth diagonal contains the ith element of the jth row for every i, j such that $i + j = k$. Define a function

$$diags :: [[a]] \to [[a]] \quad \text{-- stream of streams to stream of lists}$$

to compute the diagonals. Then we can diagonalize the stream of streams by concatenating all the (finite!) diagonals:

$$diag\ xss = concat\ (diags\ xss)$$

For example,

$$take\ 10\ rats_2 = [\,{}^1\!/\!_1, {}^2\!/\!_1, {}^1\!/\!_2, {}^3\!/\!_1, {}^1\!/\!_1, {}^1\!/\!_3, {}^4\!/\!_1, {}^3\!/\!_2, {}^2\!/\!_3, {}^1\!/\!_4\,]$$

This will indeed generate all the rationals; but it produces duplicates, for example both 1 % 2 and 2 % 4. You could then filter it to remove the duplicates (which ones are easy to spot as duplicates?), but we'll explore a better way: one that avoids generating the duplicates in the first place. We'll use a datatype of infinite binary trees:

$$\mathbf{data}\ ITree\ a = Branch\ (ITree\ a)\ a\ (ITree\ a)$$

We'll also use an unfold function for infinite trees:

$$unfold :: (b \to b) \to (b \to a) \to (b \to b) \to b \to ITree\ a$$
$$unfold\ f\ g\ h\ b = Branch\ (unfold\ f\ g\ h\ (f\ b))\ (g\ b)\ (unfold\ f\ g\ h\ (h\ b))$$

This is similar to the unfold for lists in the lectures, except of course with two generators for subtrees; but now there is no predicate or sum type, because the generated trees have only one possible shape—the infinite one.

4. Using *unfold*, define the infinite binary tree of positive integers that has a 1 on the first level, two 2s on the second level, four 3s on the third level, and so on.

   $$t_1 :: ITree\ Integer$$

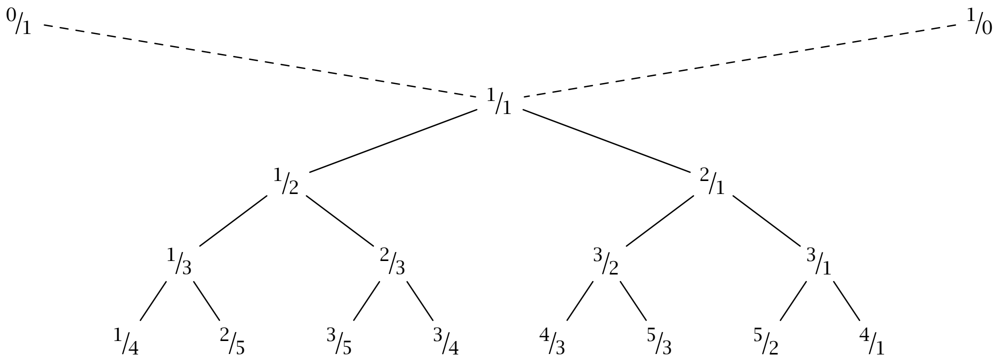

Figure 1: The top of the Stern–Brocot Tree

5. More interestingly, define the infinite binary tree of positive integers that has 1 at the root, and each node labelled n has children labelled $2\ n$ and $2\ n + 1$. This tree contains every positive natural precisely once; to find where positive integer n is in the tree, take the binary numeral for n, ignore the initial *True* bit, and treat the remainder as a path, with each subsequent bit specifying whether to take the left or the right branch.

   $t_2 :: \mathit{ITree\ Integer}$

6. It's not straightforward to test functions that generate infinite trees (how would you use QuickCheck?). As a simple expedient, define a function that extracts the first n levels of a tree, as a list of lists.

   $\mathit{levelsTo} :: \mathit{Int} \to \mathit{ITree}\ a \to [[\,a\,]]$ -- generates a list of lists

   For example,

   $$\mathit{levelsTo}\ 3\ t_1 = [[1],[2,2],[3,3,3,3]]$$
   $$\mathit{levelsTo}\ 3\ t_2 = [[1],[2,3],[4,5,6,7]]$$

7. The *Stern–Brocot Tree* is an infinite binary search tree: the label at any node is strictly greater than everything to the left, and strictly smaller than everything to the right. The top few levels are shown in Figure 1.

   Each node is the *mediant* ${}^{m+m'}/_{n+n'}$ of its rightmost left ancestor ${}^{m}/_{n}$ and its leftmost right ancestor ${}^{m'}/_{n'}$. For example, the node labelled ${}^{3}/_{4}$ has ancestors ${}^{2}/_{3}$, ${}^{1}/_{2}$, ${}^{1}/_{1}$, ${}^{0}/_{1}$, ${}^{1}/_{0}$, of which ${}^{1}/_{1}$ and ${}^{1}/_{0}$ are to the right and the others to the left. The rightmost left ancestor is $^2/_3$, and the leftmost right ancestor $^1/_1$, and indeed $^3/_4 = {}^{2+1}/_{3+1}$. The two "pseudo-nodes" $^0/_1$ and $^1/_0$ are there to make this relationship work also for nodes on the boundary of the tree; for example, $^1/_4$ is the mediant of $^0/_1$ and $^1/_3$. This also explains how to generate the tree: the seed from which the tree is grown consists of its rightmost left ancestor and leftmost right ancestor initially the two pseudo-nodes (better represented as a pair of integers, since $^1/_0$ isn't actually a rational). The tree root is their mediant, which then acts as one half of the seed for each subtree. Use this perspective to define the Stern–Brocot Tree.

$$\mathit{sternBrocot} :: \mathit{ITree}\ \mathit{Rational}$$

8. In fact, the Stern–Brocot Tree contains every positive rational precisely once: none are missing, and none are duplicated. Do you see why? And what does this have to do with Euclid's Algorithm for computing greatest common divisor?

9. We can therefore generate all the rationals directly, without duplicates, by flattening the Stern–Brocot Tree to a stream. Define a function

   $$\mathit{levels} :: \mathit{ITree}\ a \to [[a]] \quad \text{-- generates a stream of lists}$$

   to compute a stream of lists from an infinite binary tree, one per level.

10. Hence give another definition of the stream of all positive rationals.

    $$\mathit{rats}_3 :: [\mathit{Rational}] \quad \text{-- a stream}$$

    For example,

    $$\mathit{take}\ 10\ \mathit{rats}_3 = [\,^1/_1, {}^1/_2, {}^2/_1, {}^1/_3, {}^2/_3, {}^3/_2, {}^3/_1, {}^1/_4, {}^2/_5, {}^3/_5\,]$$

11. A data structure of type *ITree A* can be seen as a binary *trie*: a representation of a function from boolean sequences to values of type *A*, in which application of the function is represented by treating the boolean sequence as a path in the tree. In particular, the Stern–Brocot Tree represents a function from boolean sequences to rationals; this is the function that, for example, takes [*False*, *True*, *True*]

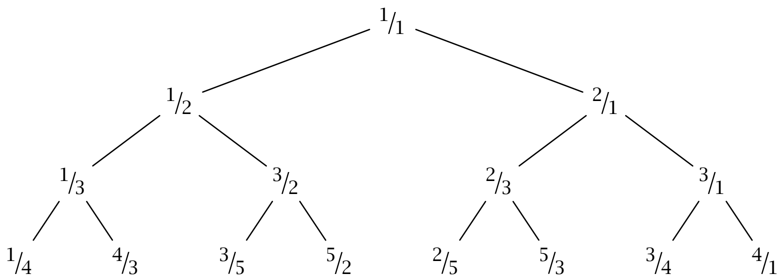

Figure 2: The top of the Calkin–Wilf Tree

to $^3/_4$, because following one left branch then two right branches from the root takes you to the node labelled $^3/_4$. Clearly, the function is total, because every node of the tree is populated.

But this means that the function from *reversed* boolean sequences can also be represented by the same type *ITree A*: the values end up on the same level, but generally in a different position on that level. The first few levels of this tree are shown in Figure 2. For example, $^3/_4$ is on the third level, as before (because it is the image of a sequence of length three); but now it is reached by two right branches then one left branch, ie following the reversed path [*True*, *True*, *False*].

This new tree is known as the *Calkin–Wilf Tree*, and of course it too contains every positive rational precisely once. It is even easier to generate: a node labelled $^m/_n$ has a left child labelled $^m/_{n+m}$ and a right child labelled $^{n+m}/_n$. Use this information to define the Calkin–Wilf Tree, and hence yet another enumeration of the positive rationals.

$$\begin{array}{l} \mathit{calkinWilf} :: \mathit{ITree\ Rational} \\ \mathit{rats}_4 :: [\mathit{Rational}] \quad \text{-- a stream} \end{array}$$

For example,

$$\mathit{take}\ 10\ \mathit{rats}_4 = [\,^1/_1, ^1/_2, ^2/_1, ^1/_3, ^3/_2, ^2/_3, ^3/_1, ^1/_4, ^4/_3, ^3/_5\,]$$

12. The Calkin–Wilf Tree is no longer a binary search tree, as the Stern–Brocot Tree was: the left-to-right ordering has been lost. But in return, there is an even closer relationship with Euclid's Algorithm, manifested in the parent-to-child ordering. Do you see what this relationship is?

13. Even better, the Calkin–Wilf has an additional remarkable property, as observed by Moshe Newman: you can generate the stream $rats_4$ without needing the tree at all! In fact, it is an instance of *iterate*: each element can be computed as a function purely of its predecessor element. That function takes a rational x to $1/_{\lfloor x \rfloor + 1 - \{x\}}$, where $\lfloor x \rfloor$ computes the "floor" if x (the largest integer at most x) and $\{x\}$ computes the "fractional part" $x - \lfloor x \rfloor$. Use this to define

$$rats_5 :: [\,Rational\,] \quad \text{-- a stream}$$

using *iterate*, so that $rats_5 = rats_4$. (Hint: Haskell provides functions

$$\begin{array}{ll} fromIntegral & :: Integer \to Rational \\ floor & :: Rational \to Integer \end{array}$$

In fact, the types are more general than these:

$$\begin{array}{ll} fromIntegral & :: (Integral\ a, \ \ Num\ b) \quad \Rightarrow a \to b \\ floor & :: (RealFrac\ a, Integral\ b) \Rightarrow a \to b \end{array}$$

but we don't need the more general types here.)

14. (not about FP) Can you explain why this works—how does iterating $\lambda x \to 1/_{\lfloor x \rfloor + 1 - \{x\}}$ neatly traverse the infinite Calkin–Wilf Tree in breadth-first order? There are three cases to consider, in order of increasing difficulty: when x is a left child, so its successor is its right sibling; when x is a right child but not on the right boundary of the tree, so its successor is some kind of cousin, and they have a more distant ancestor in common, of which they are both the same generation of descendant; and when x is on the right boundary, so its successor is on the left boundary on the next level down.

The end. . .

. . . however, I am including two longer practical exercises, in case you'd like to explore further.