Functional Pearl: La Tour D'Hanoï

Ralf Hinze

Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England

ralf.hinze@comlab.ox.ac.uk

Abstract

This pearl aims to demonstrate the ideas of wholemeal and projective programming using the Towers of Hanoi puzzle as a running example. The puzzle has its own beauty, which we hope to expose along the way.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Haskell; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks, patterns, recursion

General Terms Algorithms, Design, Languages

Keywords Towers of Hanoi, wholemeal programming, projective programming, Hanoi graph, Sierpiński graph, Sierpiński gasket graph, Gray code

1. Introduction

Functional languages excel at *wholemeal programming*, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of *projective programming*: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones. This pearl aims to demonstrate the techniques using the popular Towers of Hanoi puzzle as a running example. This puzzle has its own beauty, which we hope to expose along the way.

2. The Hanoi graph

The Towers of Hanoi puzzle was invented by the French number theorist Édouard Lucas more than a hundred years ago. It consists of three vertical pegs, on which discs of mutually different diameters can be stacked. For reference, we call the pegs A, B and C and let a, b and c range over pegs.

$$\mathbf{data}\ \mathit{Peg} = \mathrm{A} \mid \mathrm{B} \mid \mathrm{C}$$

I own a version of the puzzle where the pegs are arranged in a row. However, the mathematical structure of the puzzle becomes clearer, if we arrange them in a circle. Initially, the discs are placed on one peg in decreasing order of diameter. The task is then to move the disks, one at a time, to another peg subject to the rule that a larger disk must not be placed on a smaller one.

This restriction implies that a configuration can be represented by a list of pegs: the first element determines the position of the largest disc, the second element the position of the second largest disc, and so forth. Consequently, there are 3^n possible configurations where n is the total number of discs. Lucas' original puzzle contained 8 discs. The instructions of the puzzle refer to an old Indian legend, attributed to the French mathematician De Parville, according to which monks were given the task of moving a total of 64 discs. Since the day of the world's creation, they transfer the discs, one per day. According to the legend, once they complete their sacred task, the world will come to an end.

Now, taking a wholemeal approach, let us first develop the big picture. The set of all configurations together with the set of legal moves defines a graph, which turns out to enjoy a nice inductive definition. If there are no discs to move around, the graph consists of a singleton node: $\circ$. For the inductive step we reason as follows: the largest disc can only be moved, if all the smaller discs reside on *one* other peg. The smaller discs, however, can be moved independent of the largest one. As the largest disk may rest on one of three pegs, the graph for $n+1$ discs consequently incorporates three copies of the graph for n discs linked by three edges. The diagram in Fig. 1 illustrates the construction. The graph has the shape of a triangle; the dashed lines indicate the sub-graphs (for $n = 0$ the sub-graphs collapse to singleton nodes); the three solid lines connect the sub-graphs. The inductive construction shows that

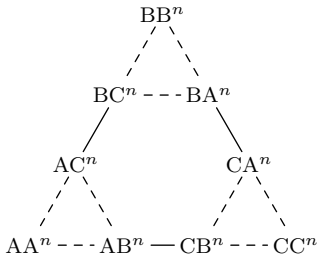

Figure 1. Inductive construction of the Hanoi graph.

the graph is planar: it can be drawn so that no edges intersect. Fig. 2 displays the graph for 4 discs. To reduce clutter, the peg of the largest disc is always written in the centre of the respective sub-graph, with the size of the font indicating the size of the disc. Can you find the configuration $[\mathrm{B}, \mathrm{A}, {}_{\mathrm{B}}, {}_{\mathrm{c}}]$? The corners of the triangle correspond to *perfect configurations*: all the discs reside on one peg. The example graph shows that every configuration permits three different moves, except for the three perfect configurations, where only two moves are possible.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

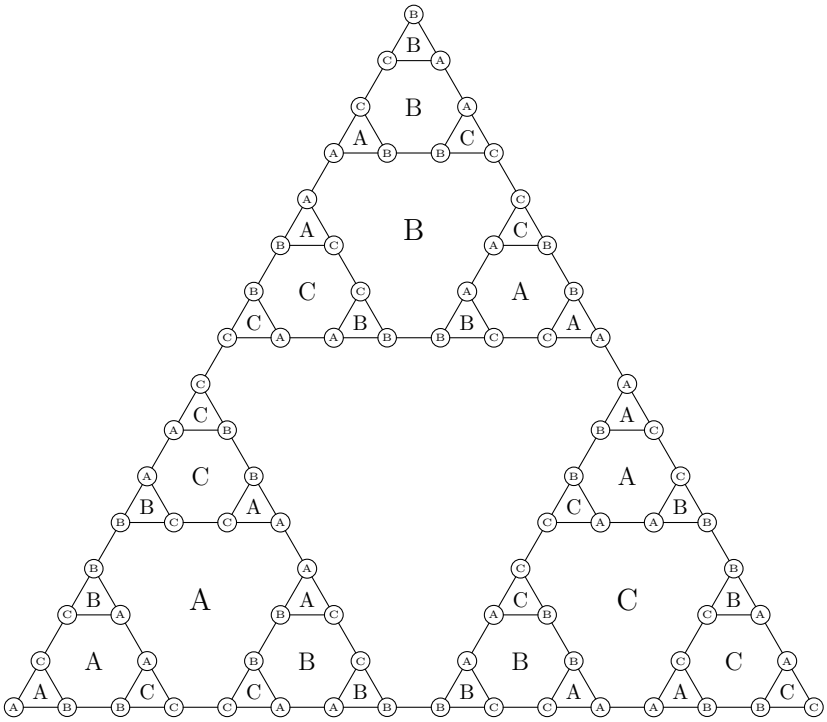

Figure 2. The Hanoi graph for 4 discs.

Let us turn our attention to the layout of the sub-graphs. The following notation proves to be useful: the *arrangement* ${}_x{}^y{}_z$ denotes a permutation of x, y and z, which by assumption are pairwise different. Using ${}_a{}^b{}_c$ to indicate the position of the largest disc—the largest disc in the left triangle resides on a and so forth—we observe that if the corners of the graph are ${}_a{}^b{}_c$, then the corners of the sub-graphs are ${}_a{}^c{}_b$, ${}_c{}^b{}_a$ and ${}_b{}^a{}_c$, respectively. Using this observation, we can capture the informal description of the construction as a pseudo-Haskell program.

$$
\begin{array}{l}
\mathit{graph}_0 \quad ({}_a{}^b{}_c) = \circ \\
\mathit{graph}_{n+1}\ ({}_a{}^b{}_c) = b \triangleleft \mathit{graph}_n\ ({}_c{}^b{}_a) \\
\qquad\qquad\qquad\quad / \qquad\qquad\qquad\quad \backslash \\
\qquad\qquad a \triangleleft \mathit{graph}_n\ ({}_a{}^c{}_b) \ \ — \ \ c \triangleleft \mathit{graph}_n\ ({}_b{}^a{}_c)
\end{array}
$$

The function graph_n maps an arrangement ${}_a{}^b{}_c$ to an undirected graph, whose vertices are lists of pegs of length n. The notation $a \triangleleft g$ means prepend a to all the vertices of g; $\circ$ denotes a singleton node labelled with the empty list. We leave the type of graphs unspecified. If the type were a functor, then $a \triangleleft g$ would be $\mathit{fmap}\ (a :)\ g$. The call $\mathit{graph}_4\ ({}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}})$ yields the graph in Fig. 2.

A few observations are in order. The definition of graph_n implies that the graph has 3^n nodes ($a_0 = 1$, $a_{n+1} = 3 \cdot a_n$) and $(3^{n+1}-3)/2$ edges ($a_0 = 0$, $a_{n+1} = 3 \cdot a_n + 3$). Furthermore, the length of a side of the triangle is $2^n - 1$ ($a_0 = 0$, $a_{n+1} = 2 \cdot a_n + 1$). Since there are only $3! = 6$ permutations of three different items, the graph contains at most six different mini-triangles: ${}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}}$, ${}_{\mathrm{C}}{}^{\mathrm{A}}{}_{\mathrm{B}}$, ${}_{\mathrm{B}}{}^{\mathrm{C}}{}_{\mathrm{A}}$, ${}_{\mathrm{A}}{}^{\mathrm{C}}{}_{\mathrm{B}}$, ${}_{\mathrm{C}}{}^{\mathrm{B}}{}_{\mathrm{A}}$ and ${}_{\mathrm{B}}{}^{\mathrm{A}}{}_{\mathrm{C}}$. Note that the first three arrange the pegs clockwise and the last three counterclockwise. Inspecting the definition of graph_n, we see that the direction changes with every recursive step. The diagram in Fig. 3 illustrates the change of direction. This observation implies that the recursion pattern is quite regular: at any depth there are only three different recursive calls. Fig. 4 visualises the call structure using three different colours.

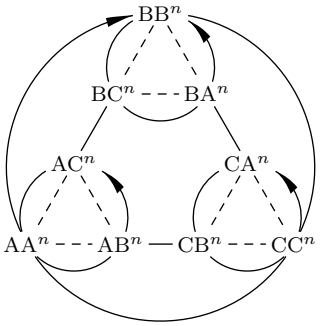

Figure 3. Change of direction.

3. Towers of Hanoi, recursively

Solving the Towers of Hanoi puzzle amounts to finding a shortest path in the corresponding graph. Clearly, the shortest path between two corners is along the side of the triangle. Projecting onto the lower side, we transform graph_n to

$$
\begin{array}{l}
\mathit{hanoi}'_0 \quad ({}_a{}^b{}_c) = [[\,]] \\
\mathit{hanoi}'_{n+1}\ ({}_a{}^b{}_c) = a \triangleleft \mathit{hanoi}'_n\ ({}_a{}^c{}_b) +\!\!+ c \triangleleft \mathit{hanoi}'_n\ ({}_b{}^a{}_c) \ .
\end{array}
$$

Now, $a \triangleleft x$ is shorthand for $\mathit{fmap}\ (a :)\ x$. The call $\mathit{hanoi}'_n\ ({}_a{}^b{}_c)$ returns a list of configurations solving the problem of 'moving n discs from a to c using b'.

Instead of configurations (vertices of the graph) we can alternatively return a list of moves (corresponding to edges).

$$
\begin{array}{l}
\mathit{hanoi}_0 \quad ({}_a{}^b{}_c) = [\,] \\
\mathit{hanoi}_{n+1}\ ({}_a{}^b{}_c) = \mathit{hanoi}_n\ ({}_a{}^c{}_b) +\!\!+ [(a, c)] +\!\!+ \mathit{hanoi}_n\ ({}_b{}^a{}_c)
\end{array}
$$

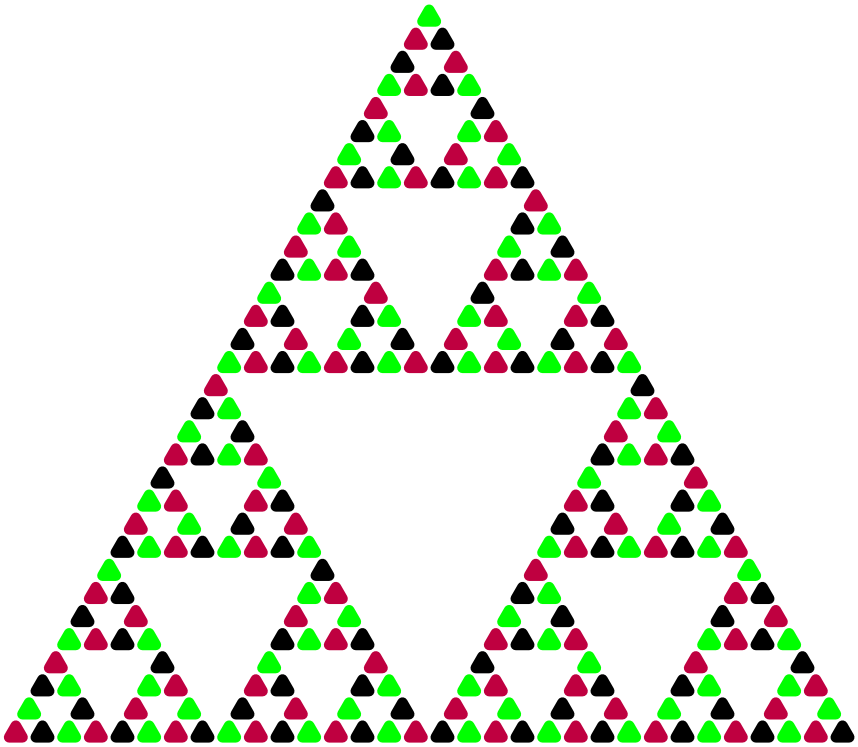

Figure 4. Call structure of $hanoi$ at recursion depth 4.

The pair (a, c) represents the instruction 'move the topmost disc from a to c'. Here is the sequence of moves for $n = 4$:

$$\begin{array}{l} \gg hanoi_4\ (_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}}) \\ [(\mathrm{A},\mathrm{B}),(\mathrm{A},\mathrm{C}),(\mathrm{B},\mathrm{C}),(\mathrm{A},\mathrm{B}),(\mathrm{C},\mathrm{A}),(\mathrm{C},\mathrm{B}),(\mathrm{A},\mathrm{B}),(\mathrm{A},\mathrm{C}), \\ \ (\mathrm{B},\mathrm{C}),(\mathrm{B},\mathrm{A}),(\mathrm{C},\mathrm{A}),(\mathrm{B},\mathrm{C}),(\mathrm{A},\mathrm{B}),(\mathrm{A},\mathrm{C}),(\mathrm{B},\mathrm{C})]\ . \end{array}$$

Note that the lower part of the list can be obtained from the upper part via a clockwise rotation of the pegs: $_a{}^c{}_b$ becomes $_b{}^a{}_c$.

There is at least one further variation: instead of an arrangement one can pass a source and a target peg.

$$\begin{array}{l} hanoi_0 \quad a\ c = [\,] \\ hanoi_{n+1}\ a\ c \\ \quad = hanoi_n\ a\ (a \perp c) +\!\!+ [(a,c)] +\!\!+ hanoi_n\ (a \perp c)\ c \end{array}$$

The function $\perp$, which determines 'the other peg', is given by

$$\begin{array}{lll} \mathrm{A} \perp \mathrm{A} = \mathrm{A}; & \mathrm{A} \perp \mathrm{B} = \mathrm{C}; & \mathrm{A} \perp \mathrm{C} = \mathrm{B} \\ \mathrm{B} \perp \mathrm{A} = \mathrm{C}; & \mathrm{B} \perp \mathrm{B} = \mathrm{B}; & \mathrm{B} \perp \mathrm{C} = \mathrm{A} \\ \mathrm{C} \perp \mathrm{A} = \mathrm{B}; & \mathrm{C} \perp \mathrm{B} = \mathrm{A}; & \mathrm{C} \perp \mathrm{C} = \mathrm{C}\ . \end{array}$$

We will find some use for $\perp$ later on. For the moment, we just note that the operation is commutative and idempotent, but not associative.

4. Towers of Hanoi, parallelly

Imagine that the monastery always accommodates as many monks as there are discs. The tallest monk is responsible for moving the largest disc, the second tallest monk for moving the second largest disc, and so forth. Can we set up a work schedule for the monastery?

Inspecting Fig. 2, we notice that, somewhat unfairly, the smallest monk is the busiest. Since the smallest triangles correspond to moves of the smallest disc, he is active every other day. We can extract his work plan from $hanoi_n$ by omitting all the moves, except when n equals 1.

$$\begin{array}{l} cycle_0 \quad (_a{}^b{}_c) = [\,] \\ cycle_1 \quad (_a{}^b{}_c) = [(a,c)] \\ cycle_{n+1}\ (_a{}^b{}_c) = cycle_n\ (_a{}^c{}_b) +\!\!+ cycle_n\ (_b{}^a{}_c) \end{array}$$

The function is called $cycle$, because the smallest disc cycles around the pegs: in the recursive call it is moved from a to b and then from b to c. Whether it cycles clockwise or counterclockwise depends on the parity of n—the direction changes with every recursive call of $cycle$.

Of course, the smallest disc is by no means special: all the discs cycle around the pegs, albeit at a slower pace and in alternating directions. In fact, $hanoi$ satisfies the following 'fractal' property:

$$hanoi_{n+1}\ (_a{}^b{}_c) = cycle_{n+1}\ (_a{}^b{}_c)\ \curlyvee\ hanoi_n\ (_a{}^b{}_c)\ , \qquad (1)$$

where $\curlyvee$ denotes the interleaving of two lists.

$$\begin{array}{l} [\,] \qquad \curlyvee\ bs = bs \\ (a : as)\ \curlyvee\ bs = a : (bs\ \curlyvee\ as) \end{array}$$

The fractal property suggests the following alternative definition of $hanoi$, which has a strong parallel flavour.

$$\begin{array}{l} phanoi_0 \quad (_a{}^b{}_c) = [\,] \\ phanoi_{n+1}\ (_a{}^b{}_c) = cycle_{n+1}\ (_a{}^b{}_c)\ \curlyvee\ phanoi_n\ (_a{}^b{}_c) \end{array}$$

In words, the smallest monk starts to work on the first day; he is active every second day and moves his disc, say, clockwise around the pegs. The second smallest monk starts on the second day; he is active every fourth day and moves his disc counterclockwise. And so forth. Actually, only the smallest monk must remember the direction; for the larger discs there is no choice, as one of the other two pegs is blocked by the smallest disc.

There is an intriguing cross-connection to binary numbers: the activity diagram of the monks (Which monk has to work on a given day?)

$$\begin{aligned} discs_0 &= [\,] \\ discs_{n+1} &= discs_n +\!\!+ [n] +\!\!+ discs_n \end{aligned}$$

yields the *binary carry sequence* or *ruler function* as n goes to infinity (Hinze, 2008). This sequence gives the number of trailing zeros in the binary representations of the positive numbers, most significant bit first. Or put differently, it specifies the running time of the binary increment. We will make use of this observation later on.

The fractal property (1) enjoys a simple inductive proof, which makes essential use of the following *abide law*. If x_1 and x_2 are of the same length, then

$$(x_1 +\!\!+ y_1) \curlyvee (x_2 +\!\!+ y_2) = (x_1 \curlyvee x_2) +\!\!+ (y_1 \curlyvee y_2) \ .$$

The basis of the induction is straightforward. Here is the inductive step.

$$\begin{aligned} & cycle_{n+2}\ ({}_a{}^b{}_c) \curlyvee hanoi_{n+1}\ ({}_a{}^b{}_c) \\ = \quad & \{ \text{ definition of } cycle \text{ and definition of } hanoi \ \} \\ & (cycle_{n+1}\ ({}_a{}^c{}_b) +\!\!+ cycle_{n+1}\ ({}_b{}^a{}_c)) \\ & \quad \curlyvee (hanoi_n\ ({}_a{}^c{}_b) +\!\!+ [(a,c)] +\!\!+ hanoi_n\ ({}_b{}^a{}_c)) \\ = \quad & \{ \text{ abide law } \} \\ & (cycle_{n+1}\ ({}_a{}^c{}_b) \curlyvee hanoi_n\ ({}_a{}^c{}_b)) +\!\!+ [(a,c)] \\ & \quad +\!\!+ (cycle_{n+1}\ ({}_b{}^a{}_c) \curlyvee hanoi_n\ ({}_b{}^a{}_c)) \\ = \quad & \{ \text{ ex hypothesi } \} \\ & hanoi_{n+1}\ ({}_a{}^c{}_b) +\!\!+ [(a,c)] +\!\!+ hanoi_{n+1}\ ({}_b{}^a{}_c) \\ = \quad & \{ \text{ definition of } hanoi \ \} \\ & hanoi_{n+2}\ ({}_a{}^b{}_c) \end{aligned}$$

The abide law is applicable in the second step, because the two lists, $cycle_{n+1}\ ({}_a{}^c{}_b)$ and $hanoi_n\ ({}_a{}^c{}_b) +\!\!+ [(a,c)]$, have the same length, namely, 2^n. Speaking of the length of lists, note that $2^{n+1} - 1 = \sum_{i=0}^{n} 2^i$ is a simple consequence of the fractal property.

5. When will the world come to an end?

Many visitors come to the monastery. Looking at the configuration of discs, they often wonder how many days have passed since the creation of the world. Or, when will the world come to an end?

We can answer these questions by locating the current configuration in the Hanoi graph. If we use as positions the three digits ${}_0{}^2{}_1$—that is, 0 for the left triangle, 2 for the upper triangle and 1 for the right triangle—then we obtain the answer to the first question in binary. Fig. 5 displays the Hanoi graph for 4 discs suitably re-labelled—this graph is also known as the Sierpiński graph. The 2^4 nodes on the lower side of the triangle, and only those, are labelled with binary numbers. Consequently, if the current position contains a 2, we know that the monks have lost track. (To answer the second question, we use as positions ${}_1{}^2{}_0$ instead of ${}_0{}^2{}_1$. If we are only interested in the distance to the final configuration, we simply replace the digit 2 by a 1.)

$$\begin{aligned} & pos\ ({}_a{}^b{}_c)\ [\,] = [\,] \\ & pos\ ({}_a{}^b{}_c)\ (p : ps) \\ & \quad \mid p == a \quad = 0 : pos\ ({}_a{}^c{}_b)\ ps \\ & \quad \mid p == b \quad = 2 : pos\ ({}_c{}^b{}_a)\ ps \\ & \quad \mid p == c \quad = 1 : pos\ ({}_b{}^a{}_c)\ ps \end{aligned}$$

For instance, $pos\ ({}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}})\ [\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{A}]$ yields $[0,1,0,1]$, the binary representation of 5, most significant bit first. The function $pos\ ({}_a{}^b{}_c)$ defines a bijection between $\{\mathrm{A},\mathrm{B},\mathrm{C}\}^n$ and $\{0,2,1\}^n$ for any given initial arrangement ${}_a{}^b{}_c$. This arrangement can be seen as representing the bijective mapping $\{a \mapsto 0, b \mapsto 2, c \mapsto 1\}$. Alternatively, we can use an arrangement, say, ${}_l{}^t{}_r$ as a representation of the 'inverse' mapping $\{\mathrm{A} \mapsto l, \mathrm{B} \mapsto t, \mathrm{C} \mapsto r\}$ obtaining the following slightly more succinct variant of pos.

$$\begin{aligned} & pos'\ ({}_l{}^t{}_r)\ [\,] = [\,] \\ & pos'\ ({}_l{}^t{}_r)\ (\mathrm{A} : ps) = l : pos'\ ({}_l{}^r{}_t)\ ps \\ & pos'\ ({}_l{}^t{}_r)\ (\mathrm{B} : ps) = t : pos'\ ({}_r{}^t{}_l)\ ps \\ & pos'\ ({}_l{}^t{}_r)\ (\mathrm{C} : ps) = r : pos'\ ({}_t{}^l{}_r)\ ps \end{aligned}$$

The two variants are related by $pos\ ({}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}}) = pos'\ ({}_0{}^2{}_1)$. The latter definition is particularly easy to invert.

$$\begin{aligned} & conf\ ({}_a{}^b{}_c)\ [\,] = [\,] \\ & conf\ ({}_a{}^b{}_c)\ (0 : ps) = a : conf\ ({}_a{}^c{}_b)\ ps \\ & conf\ ({}_a{}^b{}_c)\ (2 : ps) = b : conf\ ({}_c{}^b{}_a)\ ps \\ & conf\ ({}_a{}^b{}_c)\ (1 : ps) = c : conf\ ({}_b{}^a{}_c)\ ps \end{aligned}$$

The call $conf\ ({}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}})\ [0,1,0,1]$ yields $[\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{A}]$, the configuration we obtain after 5 days. The functions pos' and $conf$ are actually identical, if we identify A with 0, B with 2 and C with 1. Then pos' is an involutive graph isomorphism between Hanoi graphs and Sierpiński graphs of the same order.

If the monks have lost track—a 2 appears in the answer list—then we can use the idea of locating a configuration in the Hanoi graph to determine the shortest path to the final configuration c^n. This leads to the following generalised version of $hanoi$.

$$\begin{aligned} & ghanoi_0\ \ [\,]\ \ ({}_a{}^b{}_c) = [\,] \\ & ghanoi_{n+1}\ (p : ps)\ ({}_a{}^b{}_c) \\ & \quad \mid p == a = ghanoi_n\ ps\ ({}_a{}^c{}_b) +\!\!+ [(a,c)] +\!\!+ hanoi_n\ ({}_b{}^a{}_c) \\ & \quad \mid p == b = ghanoi_n\ ps\ ({}_b{}^c{}_a) +\!\!+ [(b,c)] +\!\!+ hanoi_n\ ({}_a{}^b{}_c) \\ & \quad \mid p == c = ghanoi_n\ ps\ ({}_a{}^b{}_c) \end{aligned}$$

(The index n is actually redundant: it always equals the length of the peg list.) Depending on the location of p, we either walk within the left triangle and then along the lower side, or within the upper triangle and then along the right side, or within the right triangle. The reader should convince herself that $ghanoi_n\ ps\ ({}_a{}^b{}_c)$ indeed yields the *shortest path* between ps and c^n. Note that the first two equations are perfectly symmetric, in fact, $ghanoi_n\ ps\ ({}_a{}^b{}_c) = ghanoi_n\ ps\ ({}_b{}^a{}_c)$. As an aside, if we fuse $ghanoi$ with $length$, then we obtain a function that yields the distance to the final configuration.

6. Towers of Hanoi, iteratively

The generalised version of the puzzle—moving from an arbitrary configuration to a perfect configuration—serves as an excellent starting point for the derivation of an iterative solution, that is, a function that maps a configuration to the next move or to the next configuration.

The next move is easy to determine: we fuse $ghanoi$ with the natural transformation

$$\begin{aligned} & first\ \ :: [\alpha] \to Maybe\ \alpha \\ & first\ [\,] = Nothing \\ & first\ (a : as) = Just\ a \end{aligned}$$

that maps a list to its first element. We obtain

$$\begin{aligned} & move_0\ \ [\,]\ \ ({}_a{}^b{}_c) = Nothing \\ & move_{n+1}\ (p : ps)\ ({}_a{}^b{}_c) \\ & \quad \mid p == a = move_n\ ps\ ({}_a{}^c{}_b) +\!\!+ Just(a,c) +\!\!+ first(hanoi_n({}_b{}^a{}_c)) \\ & \quad \mid p == b = move_n\ ps\ ({}_b{}^c{}_a) +\!\!+ Just(b,c) +\!\!+ first(hanoi_n({}_a{}^b{}_c)) \\ & \quad \mid p == c = move_n\ ps\ ({}_a{}^b{}_c) \ . \end{aligned}$$

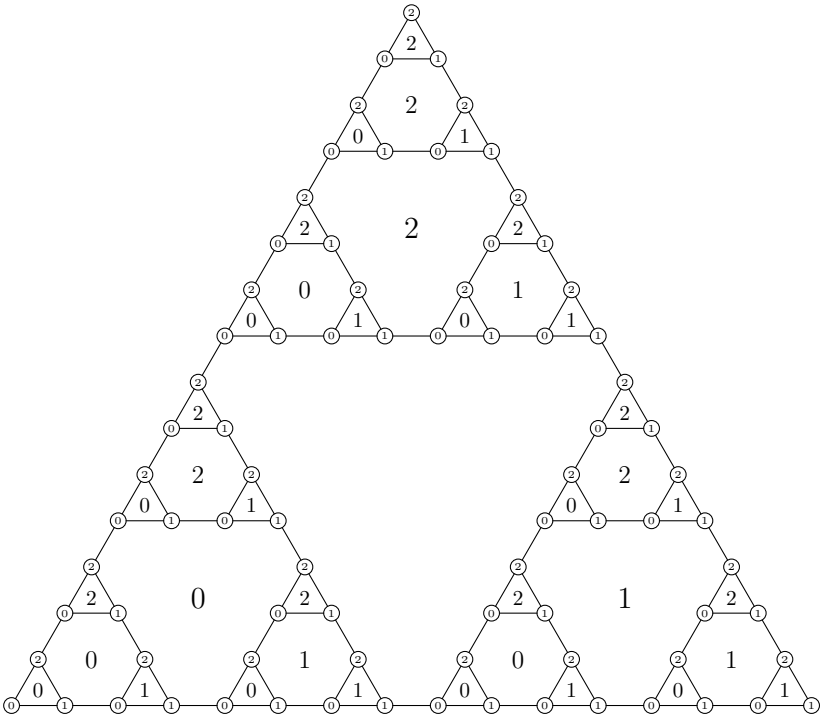

Figure 5. The Sierpiński graph of order 4.

The operator $+\!\!+$ is overloaded to also denote concatenation of *Maybe* values ($+\!\!+$ is really the *mplus* method of *MonadPlus*).

$$\begin{array}{l}(+\!\!+) \qquad\qquad :: \mathit{Maybe}\ \alpha \to \mathit{Maybe}\ \alpha \to \mathit{Maybe}\ \alpha \\ \mathit{Nothing} +\!\!+ m = m \\ \mathit{Just}\ a \quad +\!\!+ m = \mathit{Just}\ a\end{array}$$

We can simplify the definition of *move* drastically: since $\mathit{Just}\ a +\!\!+ m = \mathit{Just}\ a$, the calls to *hanoi* can be eliminated; because of that the index n is no longer needed. Furthermore, the first two equations can be unified through the use of $\perp$, the operator that determines 'the other peg'. Finally, the second argument, ${}_a{}^b{}_c$, can be simplified to c, the target peg.

$$\begin{array}{l}\mathit{move}\ [\,] \qquad\quad c = \mathit{Nothing} \\ \mathit{move}\ (p : ps)\ c \\ \quad \mid p \neq c \qquad = \mathit{move}\ ps\ (p \perp c) +\!\!+ \mathit{Just}\ (p, c) \\ \quad \mid p == c \qquad = \mathit{move}\ ps\ c\end{array}$$

Here is a short interactive session that illustrates the use of *move*.

$$\begin{array}{l}\gg \mathit{move}\ [\mathrm{A},\mathrm{B},\mathrm{B},\mathrm{A},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{B}]\ \mathrm{C} \\ \mathit{Just}\ (\mathrm{B},\mathrm{C}) \\ \gg \mathit{move}\ [\mathrm{A},\mathrm{B},\mathrm{B},\mathrm{A},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C}]\ \mathrm{C} \\ \mathit{Just}\ (\mathrm{A},\mathrm{B}) \\ \gg \mathit{move}\ [\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C}]\ \mathrm{C} \\ \mathit{Nothing}\end{array}$$

Since the pair (B,C) means 'move the topmost disc from peg B to peg C, the configuration following $[\mathrm{A},\mathrm{B},\mathrm{B},\mathrm{A},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{B}]$ is $[\mathrm{A},\mathrm{B},\mathrm{B},\mathrm{A},\mathrm{C},\mathrm{C},\mathrm{C},\mathrm{C}]$. Incidentally, if we start with the initial peg list $[\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A},\mathrm{A}]$ and target C, we obtain these configurations after 110 and 111 steps.

The function *move* implements a two-way algorithm: on the way down the recursion it calculates the target peg for each disc (using $\perp$); on the way up the recursion it determines the smallest disc that is not yet in place (using $+\!\!+$). The following table makes the target pegs explicit. The rows labelled c_i lists the target peg for each disc: the first, user-supplied target is $c_0 = \mathrm{C}$, the next target is $c_1 = p_0 \perp c_0$, and so forth.

	i	0	1	2	3	4	5	6	7	
110	p_i	A	B	B	A	C	C	C	B	
	c_i	C	B	B	B	C	C	C	C	A
111	p_i	A	B	B	A	C	C	C	C	
	c_i	C	B	B	B	C	C	C	C	C

The smallest disc is not in place, so it is moved from B to C. In the next round, disc 3 has to be moved from A to B.

The smaller discs are moved more frequently, so it is actually prudent to reverse the list of pegs so that the peg on which the smallest disc is located comes first. The situation is similar to the binary increment: visiting the digits from least to most significant bit is more efficient than the other way round. Let us assume for the moment that we know the 'last target', the value of c that is discarded in the first equation of *move* (the pegs that stick out in the example above). Since $\perp$ is reversible, $p \perp c = c'$ iff $c = p \perp c'$, we can reconstruct the previous target c from the next target c'. The following variant of *move* makes use of this fact—the suffix 'i' indicates that the input list is now arranged in *i*ncreasing order of diameter.

$$\begin{array}{l}\mathit{movei}\ [\,] \qquad\quad c' = \mathit{Nothing} \\ \mathit{movei}\ (p : ps)\ c' \\ \quad \mid p \neq (p \perp c') = \mathit{Just}\ (p, p \perp c') +\!\!+ \mathit{movei}\ ps\ (p \perp c') \\ \quad \mid p == (p \perp c') = \qquad\qquad\qquad\quad \mathit{movei}\ ps\ c'\end{array}$$

We consistently changed the second argument of *move* to reflect the fact that we compute the previous from the next and additionally replaced the remaining occurrences of c by $p \perp c'$. Again, we can simplify the code: $p \neq (p \perp c')$ is the same as $p \neq c'$. Applying $\mathit{Just}\ a +\!\!+ m = \mathit{Just}\ a$ once more, we can eliminate the first recursive call to *movei* so that the function

stops as soon as the smallest displaced disc is found—this was the purpose of the whole exercise. We obtain

$$\begin{array}{ll} \mathit{movei}\ [\,] & c' = \mathit{Nothing} \\ \mathit{movei}\ (p : ps)\ c' & \\ \quad \mid p \neq c' & = \mathit{Just}\ (p, p \perp c') \\ \quad \mid p == c' & = \mathit{movei}\ ps\ c'\ . \end{array}$$

In a nutshell, movei determines the smallest disc that does not reside on the last target.

So, for an iterative version of hanoi we have to maintain two pieces of information: the current configuration and the current 'last target'. It remains to determine the initial last target and how the last target changes after each move. If the list of pegs is given in the original decreasing order, then we can transform move to

$$\begin{array}{ll} \mathit{last}\ [\,] & c = c \\ \mathit{last}\ (p : ps)\ c & = \mathit{last}\ ps\ (p \perp c)\ , \end{array}$$

which yields the last target. It is not hard to see that last is an instance of the famous foldl: $\mathit{last}\ \ ps\ \ c = \mathit{foldl}\ \ (\perp)\ \ c\ \ ps$. If we reverse the list, we simply have to replace foldl by foldr additionally using the fact that $\perp$ is commutative: $\mathit{foldr}\ (\perp)\ c\ ps = \mathit{foldl}\ (\perp)\ c\ (\mathit{reverse}\ ps)$.

Next, we augment movei so that it returns the next configuration instead of the next move and additionally the next 'last target'.

$$\begin{array}{ll} \mathit{step}\ ([\,], & c') = \mathit{Nothing} \\ \mathit{step}\ (p : ps, c') & \\ \quad \mid p \neq c' & = \mathit{Just}\ ((p \perp c') : ps, p \perp c') \\ \quad \mid p == c' & = \mathbf{do}\ \{(ps', c) \leftarrow \mathit{step}\ (ps, c'); \\ & \qquad \mathit{return}\ (p : ps', p \perp c)\} \end{array}$$

Consider the second equation: p is moved to $p \perp c'$. After the move, the disc resides on the target peg. Consequently, the next target is also $p \perp c'$—recall that $\perp$ is idempotent. This target is then updated in the third equation 'on the way back' mimicking move's mode of operation. The function step runs in constant amortised time, since it performs the same number of steps as the binary increment—recall that the activity diagram of the monks coincides with the binary carry sequence.

The next last target can, in fact, be easily calculated by hand. Consider the following two subsequent moves (as usual, a, b and c are pairwise different).

	n	$n-1$	$n-2$		1	0	
$\cdots$	a	c	c	$\cdots$	c	c	
$\cdots$	b	c	c	$\cdots$	c	c	c
$\cdots$	b	c	c	$\cdots$	c	c	
$\cdots$	b	b	a	$\cdots$	b	a	b

Assume that the first configuration ends with an even number of cs. The topmost disc of a is then moved to b. The new succession of target pegs consequently alternates between b and a: since the number of cs is even, the new last target is b; for an odd number, it is a. So, the monks can be instructed as follows: determine the smallest disc that is not on c. Transfer it from a to b. If the disc's number is even, the new last target is b; otherwise, it is a.

If we solve the original puzzle, that is, if the configurations lie on one of the sides of the triangle, then the next last target is even easier to determine: like the discs, it cycles around the pegs. If the pegs are arranged ${}_{\mathrm{A}}{}^{\mathrm{B}}{}_{\mathrm{C}}$ and we move the discs from A to C, then the last target moves counterclockwise for an even number of discs and clockwise for an odd number.

Summing up, we obtain the following iterative implementation for solving the *generalised* Hanoi puzzle.

$$\begin{array}{ll} \mathit{ihanoi}\ ps\ c = \mathit{map}\ \mathit{fst}\ (\mathit{iterate}\ \mathit{step}\ (ps, \mathit{foldr}\ (\perp)\ c\ ps)) & \\ \mathit{iterate} & :: (a \to \mathit{Maybe}\ a) \to (a \to [a]) \\ \mathit{iterate}\ f\ x & = x : \mathbf{case}\ f\ x\ \mathbf{of}\ \{\mathit{Nothing} \to [\,]; \\ & \qquad\qquad \mathit{Just}\ x' \ \to \mathit{iterate}\ f\ x'\} \end{array}$$

7. Longest paths and Sierpiński's triangle

So far we have considered shortest paths in the Hanoi graph. Since the destruction of the world hangs in the balance, as a gift to future generations, we might want to look for the *longest path*. In the following variant of hanoi the largest disc makes a detour. (According to the Indian legend, the temple is actually in Bramah rather than in Hanoi, hence the name of the function.)

$$\begin{array}{ll} \mathit{bramah}_0 & ({}_a{}^b{}_c) = [\,] \\ \mathit{bramah}_{n+1} & ({}_a{}^b{}_c) = \mathit{bramah}_n\ ({}_a{}^b{}_c) +\!\!+ [(a, b)] \\ & \qquad +\!\!+ \mathit{bramah}_n\ ({}_c{}^b{}_a) +\!\!+ [(b, c)] \\ & \qquad +\!\!+ \mathit{bramah}_n\ ({}_a{}^b{}_c) \end{array}$$

The largest disc is first moved from a to b, and then from b to c. Since bramah_n returns $3^n - 1$ moves ($a_0 = 0$, $a_{n+1} = 3 \cdot a_n + 2$), we have actually found a *Hamiltonian path*. The path has another interesting property: if the pegs are arranged in a row, $a\ b\ c$, then discs are only moved between adjacent pegs.

The Hamiltonian path for four discs is displayed in Fig. 6. The picture is quite appealing. Actually, if we move the sub-triangles closer to each other so that the corners touch, we obtain a nice fractal image. Fig. 7 shows the result for 7 discs. The corresponding graph is known as the discrete Sierpiński gasket graph.[1] The picture has been drawn using Functional Metapost's turtle graphics (Korittky, 1998).

$$\begin{array}{ll} \mathit{curve}_0 & d = \mathit{forward}\ \&\ \mathit{turn}\ (2 * d)\ \&\ \mathit{forward} \\ \mathit{curve}_{n+1} & d = \mathit{turn}\ d\ \&\ \mathit{curve}_n\ (-d)\ \&\ \mathit{turn}\ d\ \&\ \mathit{curve}_n\ d \\ & \quad \&\ \mathit{turn}\ d\ \&\ \mathit{curve}_n\ (-d)\ \&\ \mathit{turn}\ d \end{array}$$

The command $\mathit{forward}$ moves the turtle one step forward, $\mathit{turn}\ d$ turns the turtle clockwise by d degrees, and $\&$ sequences two turtle actions. Since turtle graphics is state-based—the turtle has a position and faces a direction—recursive definitions typically maintain an invariant. To draw the 'triangle' ${}_a{}^b{}_c$, we start at a looking at b and stop at c looking away from b. The overall change of direction is twice the second argument of curve, which for equilateral triangles is either $60°$ or $-60°$.

The curve is closely related to Sierpiński's arrowhead curve. In fact, both curves yield Sierpiński's triangle as n goes to infinity. As an aside, Sierpiński's triangle is a so-called *fractal curve*: it has the Hausdorff dimension $\log 3 / \log 2 \cong 1.58496$, as it consists of three self-similar pieces with magnification factor 2.

8. Gray code

The function bramah enumerates the configurations $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}^n$ changing only one peg at a time. In other words, the succession of configurations corresponds to a *ternary* Gray code! To investigate the correspondence a bit further, here is a version of bramah that returns a list of configurations, rather than a list of moves.

$$\begin{array}{ll} \mathit{bramah}'_0 & ({}_a{}^b{}_c) = [[\,]] \\ \mathit{bramah}'_{n+1} & ({}_a{}^b{}_c) = a \triangleleft \mathit{bramah}'_n\ ({}_a{}^b{}_c) \\ & \qquad +\!\!+ b \triangleleft \mathit{bramah}'_n\ ({}_c{}^b{}_a) \\ & \qquad +\!\!+ c \triangleleft \mathit{bramah}'_n\ ({}_a{}^b{}_c) \end{array}$$

[1] The term gasket graph is actually not used consistently. Some authors call the counterpart of the Hanoi graph the Sierpiński gasket graph, and its contracted variant the Sierpiński graph.

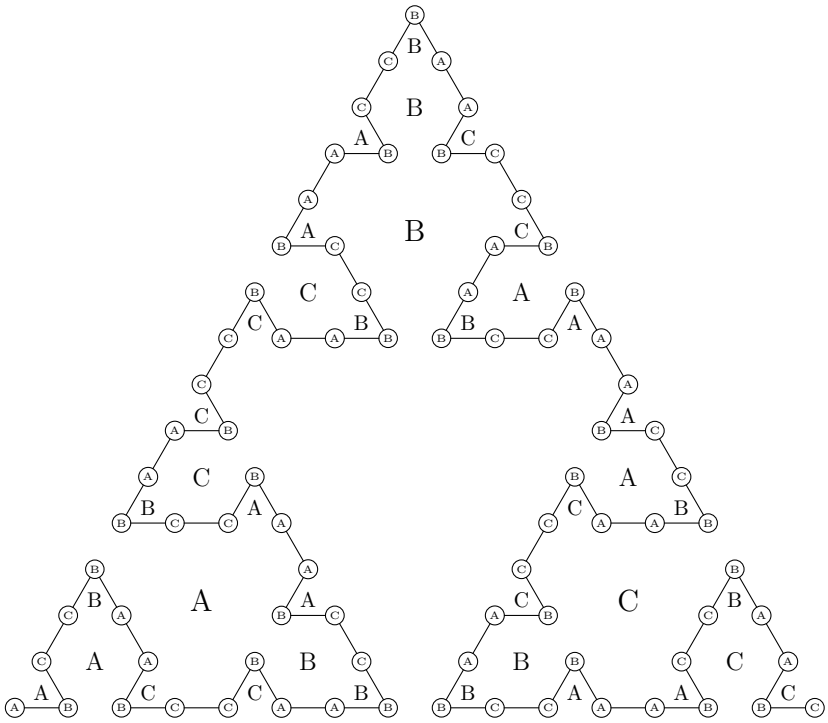

Figure 6. A Hamiltonian path in the Hanoi graph for 4 discs.

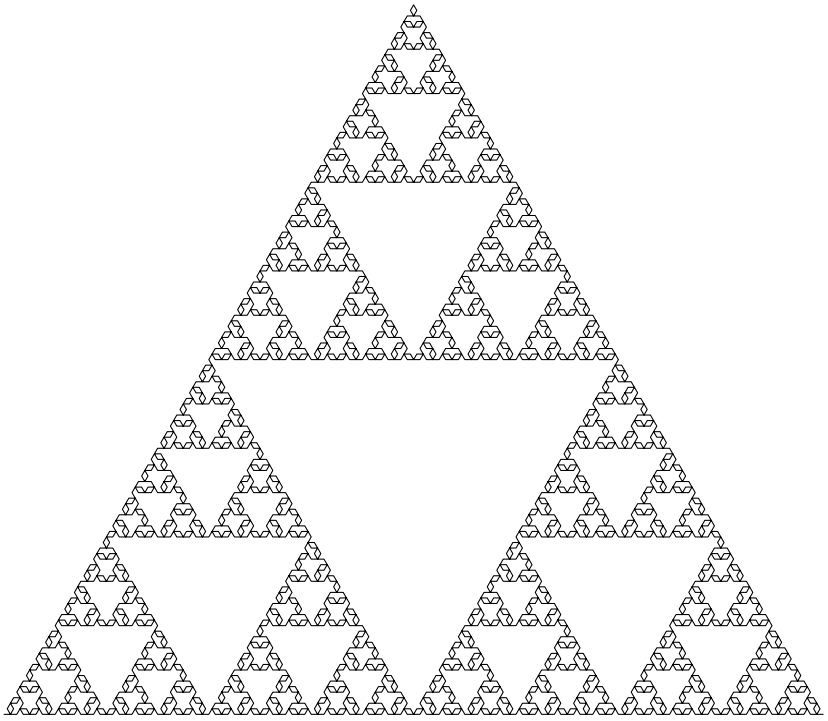

Figure 7. A Hamiltonian path in the Hanoi graph for 7 discs.

There are two standard ternary Gray codes: the so-called modular code, the digits vary $012|201|120\cdots$, and the reflected code, the digits vary $012|210|012\cdots$. The definition above yields the latter, as the discs are only moved between adjacent pegs. In fact, we have $bramah'_n\ ({}_c{}^b{}_a) = reverse\ (bramah'_n\ ({}_a{}^b{}_c))$. Using this property on the second recursive call, we can simplify $bramah'_n\ ({}_0{}^1{}_2)$ somewhat.

$$\begin{aligned} gray3_0 &= [[\,]] \\ gray3_{n+1} &= 0 \lhd gray3_n \mathbin{+\!\!+} 1 \lhd reverse\ gray3_n \mathbin{+\!\!+} 2 \lhd gray3_n \end{aligned}$$

For comparison, here is the definition of the Gray *binary* sequence.

$$\begin{aligned} gray2_0 &= [[\,]] \\ gray2_{n+1} &= 0 \lhd gray2_n \mathbin{+\!\!+} 1 \lhd reverse\ gray2_n \end{aligned}$$

Actually, the *binary* Gray code is also hidden in the Tower of Hanoi puzzle. We only have to work with a different configuration space: instead of $\{\mathrm{A},\mathrm{B},\mathrm{C}\}^n$ we use $\{0,1\}^n$ keeping track whether a disc has been moved an even or an odd number of times. The initial configuration AA^n becomes 00^n, the final configuration CC^n becomes 10^n. Since the ith disc is moved 2^i times, all the discs make an even number of moves, except for the largest, which makes a single move.

We can easily adapt $ihanoi$ to generate binary Gray code. We take as a starting point the first definition of $movei$, slightly modified so that the next configuration is returned instead of the next move (we also applied the $Just\ a \mathbin{+\!\!+} m = Just\ a$ simplification).

$$\begin{aligned} &configi\ [\,] \qquad c' = Nothing \\ &configi\ (p : ps)\ c' \\ &\quad \mid p \neq (p \perp c') \ = Just\ (p \perp c' : ps) \\ &\quad \mid p == (p \perp c') \ = p \lhd configi\ ps\ (p \perp c') \end{aligned}$$

Now, the Gray code equivalent of $\perp$ is the Boolean operation *exclusive or*, that is, inequality of Booleans. This implies that the last target c' corresponds to a *parity bit*. Thus, $configi$ becomes (the code uses Booleans rather than bits)

$$\begin{aligned} &codei\ [\,] \qquad p \ = Nothing \\ &codei\ (b : bs)\ p \\ &\quad \mid b \neq (b \neq p) = Just\ ((b \neq p) : bs) \\ &\quad \mid b == (b \neq p) = b \lhd codei\ bs\ (b \neq p)\ . \end{aligned}$$

Again, we can simplify the code a bit: inequality and equality of Boolean values are associative (Backhouse and Fokkinga, 2001), so $b \neq (b \neq p)$ is simply p. Using $False \neq b = b$ and $True \neq b = \neg\, b$, we obtain

$$\begin{aligned} &codei\ [\,] \qquad p = Nothing \\ &codei\ (b : bs)\ p \\ &\quad \mid p \qquad\quad = Just\ (\neg\, b : bs) \\ &\quad \mid \neg\, p \qquad = b \lhd codei\ bs\ b\ . \end{aligned}$$

In words: we traverse the list $p : bs$ up to the first 1; the following bit, if any, is flipped.

As for $ihanoi$, we have to maintain two pieces of information: the current Gray code and the current *parity bit*. The latter is easy to update: it is flipped in each step. Summing up, we obtain the following Gray code generator.

$$\begin{aligned} &igray\ bs = map\ fst\ (iterate\ step\ (bs, foldr\ (\neq)\ True\ bs)) \\ &\quad \mathbf{where}\ step\ (bs, p) = \mathbf{do}\ \{\, bs' \leftarrow codei\ bs\ p; \\ &\qquad\qquad\qquad\qquad return\ (bs', \neg\, p)\,\} \end{aligned}$$

This is, in fact, the functional version of Knuth's Algorithm G (2005).

9. Conclusion and further reading

We have come to the end of the Tour D'Hanoï. In the spirit of the wholemeal approach we started with an inductive definition of the Hanoi *graph*. From that we derived a series of programs evolving around the Tower of Hanoi theme. Knowing the big picture was jolly useful: for instance, calculating the number of moves could be reduced to the problem of locating a configuration in the graph. Projective program transformations are abundant: $hanoi$ is derived from $graph$ by projecting onto the lower side of the graph, $cycle$ is derived from $hanoi$ by mashing out the moves of the larger discs, and so forth.

The transformations could be made rigorous within the Algebra of Programming framework (Bird and de Moor, 1997). Occasionally, this comes at an additional cost. For instance, to derive $cycle$ from $hanoi$ we would additionally need the disc number, which is not present in $hanoi$'s output.

A lot is left to explore. There are literally hundreds of papers on the subject: Paul Stockmeyer's comprehensive bibliography has a total of 369 entries (2005). From that bibliography I learned that my definition of the Hanoi graph is not original: Er introduced it to analyse the complexity of the generalised Tower of Hanoi problem (1983). The original instructions of the game already alluded to the recursive procedure for solving the puzzle. It has been used since to illustrate the concept of recursion. The parallel version—usually classified as iterative— is due to Buneman and Levy (1980). Backhouse and Fokkinga (2001) show that each disc cycles around the pegs exploiting the associativity of equivalence. To the best of the author's knowledge the iterative, or stepwise variant is original. The connection to Gray codes has first been noticed by Gardner (1972).

Acknowledgments

Special thanks are due to Daniel James for enjoyable discussions and for suggesting a number of stylistic and presentational improvements. Thanks are also due to the anonymous referees for an effort to make the paper less dense.

References

Backhouse, Roland, and Maarten Fokkinga. 2001. The associativity of equivalence and the Towers of Hanoi problem. *Information Processing Letters* 77:71–76.

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

Buneman, Peter, and Leon Levy. 1980. The Towers of Hanoi problem. *Information Processing Letters* 10(4–5):243–244.

Er, M.C. 1983. An analysis of the generalized Towers of Hanoi problem. *BIT* 23:429–435.

Gardner, Martin. 1972. Mathematical games: The curious properties of the Gray code and how it can be used to solve puzzles. *Scientific American* 227(2):106–109. Reprinted, with Answer, Addendum, and Bibliography, as Chapter 2 of *Knotted Doughnuts and Other Mathematical Entertainments*, W. H. Freeman and Co., New York, 1986.

Hinze, Ralf. 2008. Functional Pearl: Streams and Unique Fixed Points. In *Proceedings of the 2008 International Conference on Functional Programming*, ed. Peter Thiemann, 189–200. ACM Press.

Knuth, Donald E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Publishing Company.

Korittky, Joachim. 1998. *Functional* METAPOST. Diplomarbeit, Universität Bonn.

Stockmeyer, Paul K. 2005. The Tower of Hanoi: A bibliography. Available from `http://www.cs.wm.edu/~pkstoc/biblio2.pdf`.