JFP **23** (2): 145–160, 2013. © Cambridge University Press 2013
doi:10.1017/S0956796813000014

FUNCTIONAL PEARL
Solving the snake cube puzzle in Haskell

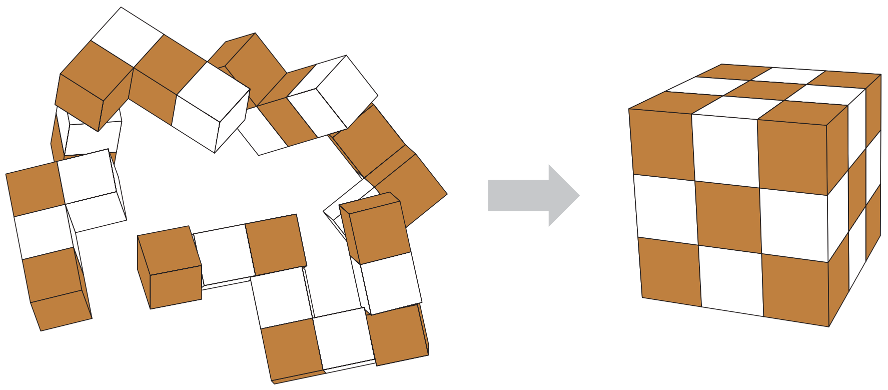
MARK P. JONES
Department of Computer Science, Portland State University, Portland, Oregon, USA
(*e-mail:* mpj@cs.pdx.edu)

Abstract

We describe a concise and elegant functional program, written in Haskell, that computes solutions for a classic puzzle known as the "snake cube." The program reflects some of the fundamental characteristics of the functional style, identifying key abstractions, and defining a small collection of operators for manipulating and working with the associated values. Well-suited for an introductory course on functional programming, this example highlights the use of visualization tools to explain and demonstrate the choices of data structures and algorithms that are used in the development.

1 Introduction

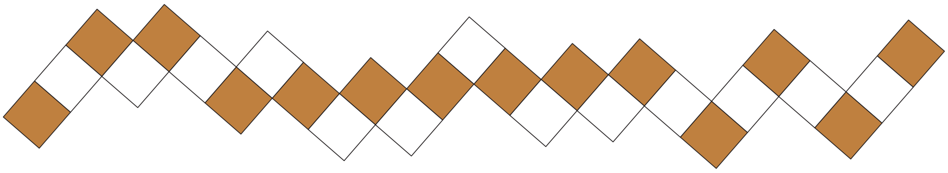
A popular wooden puzzle, the "snake cube" comprises a string of 27 small cubes, typically alternating between dark and light colours, that is solved by folding the puzzle in on itself so that the pieces form a single, large, 3×3×3 cube. The following diagram illustrates a partially folded version of the puzzle on the left and the solved form on the right:



Although the task can be accomplished in just a few seconds with prior knowledge, figuring out a solution from scratch can be quite difficult. (The author writes from the experience of his own struggles as well as the experience of watching others attempt to solve it.) In this paper we present a concise and elegant functional program written in Haskell (Peyton Jones, 2003) that computes solutions to the standard snake cube, and is readily adapted to other variants. The development is well-suited for use in an introductory course on Haskell programming: beyond the obvious visual appeal of the problem, and the ability for students to hold and experiment with a physical puzzle while they are working through the code, the program also provides a good introduction to important tools and techniques of functional programming. This includes, for example, built-in data structures such as tuples and lists; basic language constructs, particularly list comprehensions; and a novel demonstration of Wadler's (1985) programming technique for "replacing failure by a list of successes."

2 Constructing the snake cube

Before we set about the process of computing solutions, it is useful to capture the structure of the puzzle in more detail. Although other methods are possible, the snake cube is usually constructed by threading the small cubes together with an elasticated cord that stretches from one end to the other, entering and exiting individual pieces of the puzzle through centred holes in the faces of the small cubes. In some cases, the entry and exit are on opposite faces, but in others they are on adjacent faces, effectively creating a 90° change in direction. The result of this is to break the snake into straight sections, each of which includes either two or three neighbouring cubes. (These sections must fit within the final 3×3×3 cube and cannot be folded, so they can be at most three cubes in length.) The following diagram shows the structure of a standard snake cube once it has been flattened out, and makes it easy to see the sections of two or three cubes along the length of the puzzle:



The essential details of this structure can be captured by a list of integers. In the following, we choose arbitrarily to list the sections of the snake from left to right; reversing the list would, of course, result in an equivalent description of the same puzzle.

$$\begin{array}{l} \mathit{snake} :: [\mathit{Int}] \\ \mathit{snake} = [3, 2, 2, 3, 2, 3, 2, 2, 3, 3, 2, 2, 2, 3, 3, 3, 3] \end{array}$$

As a sanity check, we can verify that $\mathit{sum}\ \mathit{snake} - (\mathit{length}\ \mathit{snake} - 1) = 27$, the total number of small cubes in the puzzle. This formula works because *sum snake* overcounts the total by including each of the ($\mathit{length}\ \mathit{snake} - 1$) corner pieces twice.

It is fairly easy to see that the last cube on the right of the diagram above must end up in one of the eight corners of the final 3×3×3 cube in any solution; otherwise there is no way to fit the last four sections (each of length 3) into the final cube. In fact, the first cube on the left must also end up in a corner of the final solution. Readers with good spatial reasoning skills may be able to deduce this from the

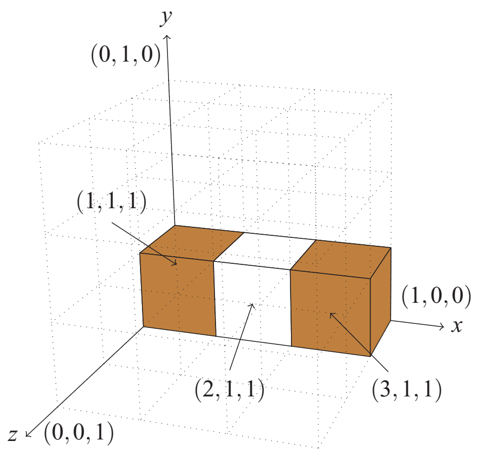


Fig. 1. Using vectors to represent *Position*s and *Direction*s within the snake cube puzzle.

diagram alone, but it quickly becomes obvious once you have the puzzle in your hands and begin to experiment.

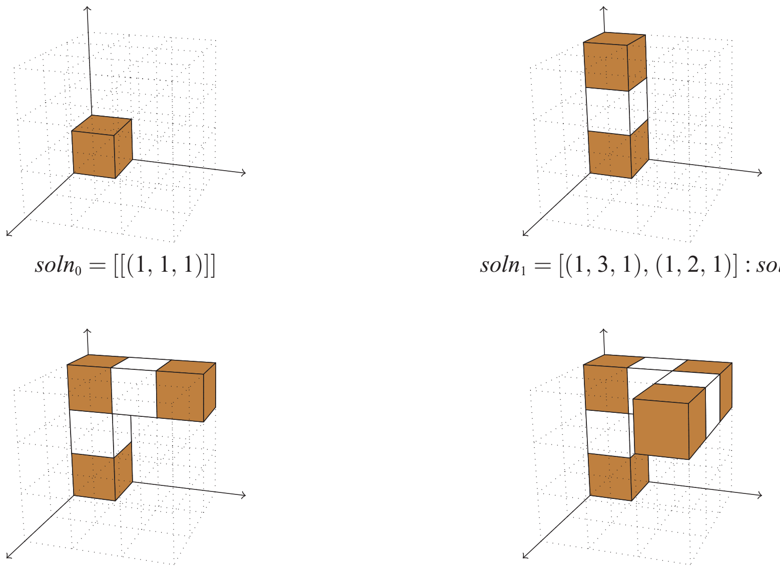
3 Moving into three dimensions

As we start to think about assembling the puzzle in three dimensions, it is natural to adopt a conventional coordinate system as illustrated in Figure 1. Note that we can use vectors such as $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ to represent directions within the large cube corresponding to the x, y, and z axes, respectively. We can also use vectors to reference each of the smaller cube positions within the assembled puzzle. In what follows, we have chosen, somewhat arbitrarily, to represent each small cube by the coordinates of its furthest point from the origin. Thus, the corner cube that is closest to the origin is $(1, 1, 1)$ and its immediate neighbours along the positive x-axis are $(2, 1, 1)$ and $(3, 1, 1)$. Note that all of the vectors that we are dealing with here have integer coordinates, so they can be represented by values of the following types:

```
type Direction = (Int, Int, Int)
type Position  = (Int, Int, Int)
```

In this paper, we will only use six different *Direction* values, corresponding in Figure 1 to right/left (x-axis), up/down (y-axis), and forward/backward (z-axis). In numerical terms, these are the vectors in which one coordinate is either 1 or −1 and the others are both zero. To describe which *Position* values correspond to valid locations within the solved puzzle, we will use a predicate, *inCube* 3, that is defined as follows:

```
inCube              :: Int → Position → Bool
inCube n (x, y, z) = inRange x && inRange y && inRange z
                     where inRange i = 1 ⩽ i && i ⩽ n
```



$soln_0 = [[(1,1,1)]]$

$soln_1 = [(1,3,1),(1,2,1)] : soln_0$

$soln_2 = [(3,3,1),(2,3,1)] : soln_1$

$soln_3 = [(3,3,3),(3,3,2)] : soln_2$

Fig. 2. A sequence of steps placing three sections, each of length 3, within a 3×3×3 cube.

Making n a parameter in this definition will allow us to generalize later to more complex versions of the puzzle, including the "king snake," which has a string of 64 small cubes that can be assembled into a large 4×4×4 cube.

4 Describing solutions

We will represent solutions to the snake cube puzzle as sequences of steps that fit each of the puzzle pieces into its final place, moving from the first section to the last. With a physical puzzle in hand, it may actually be necessary to perform the steps in a slightly different order to avoid conflicts between the pieces of the puzzle that have already been placed and the "dangling tail" that comprises the remaining puzzle sections. Although this can occur in practice, it is usually easy to work around with the puzzle in hand. And in some cases, just reversing the order of the list of sections—so that we start working from the opposite end—can make the puzzle easier to assemble (see the description of *reversePuzzle* in Section 7). For these reasons, we will not worry about modeling the dangling tail in our attempts to compute puzzle solutions.

The diagram in Figure 2 illustrates a sequence of steps for fitting three puzzle sections, each containing three small cubes, into the space of the large cube. As a special case, we begin by placing the first small cube at $(1,1,1)$ in $soln_0$. Then, for each of the three sections, we add a separate list that describes the two new pieces in that section with the position of the most recently placed small cube at the front. This will allow us to build up solutions incrementally: any solution s that fits $(n+1)$ sections will include a solution, *tail* s, that fits the first n sections (assuming $n \geqslant 1$).

```
type Solution = [Section]
type Section  = [Position]
```

The pieces within each section are also ordered with the most recently placed small cube first, so we can always find the *Position* of the most recently placed small cube in a full solution, s :: *Solution*, by using *head* (*head* s). This also means that we can obtain a list of all the positions that have been filled in a given solution s from the first to last (and with alternating dark and light colours if we wish to model that aspect of the puzzle) by using *reverse* (*concat* s). For example, with the steps illustrated in Figure 2, we have:

```
reverse (concat soln₃)
  = [(1, 1, 1), (1, 2, 1), (1, 3, 1), (2, 3, 1), (3, 3, 1), (3, 3, 2), (3, 3, 3)]
```

5 Building sections

The sequence of cubes that appears in any given section is determined: by the position, *start*, of the first cube (which is also the last cube in the previous section); by the direction, (u, v, w), of the section; and by its length, *len*.

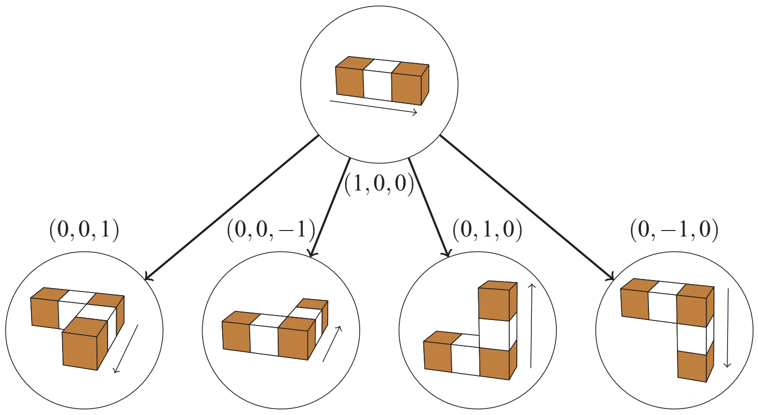
```
section                     :: Position → Direction → Int → Section
section start (u, v, w) len = reverse (tail (take len pieces))
  where pieces = iterate (\(x, y, z) → (x + u, y + v, z + w)) start
```

The local definition here produces an infinite list, *pieces*, of positions with the required start and direction. Using *take len*, we truncate the list to the length of the section, and then use *tail* to discard the first position (which will, again, have already been included in the previous section). Finally, we use *reverse* to ensure that the list is ordered with the most recently placed cube first. For example, the first two full sections from Figure 2, which appear at the heads of $soln_1$ and $soln_2$, respectively, can be calculated as follows:

```
section (1, 1, 1) (0, 1, 0) 3 = [(1, 3, 1), (1, 2, 1)]
section (1, 3, 1) (1, 0, 0) 3 = [(3, 3, 1), (2, 3, 1)]
```

6 Changing directions

It remains to account for the change of direction between adjacent puzzle sections. As an example, if we begin with a section that has direction $(1, 0, 0)$, then there are four possible directions for the next section—forwards $(0, 0, 1)$, backwards $(0, 0, -1)$, up $(0, 1, 0)$, and down $(0, -1, 0)$—as illustrated in the following diagram:



More generally, in any given case, the new directions can be obtained from the old by rotating the coordinates of the old direction to the left (using the function *left* $(x, y, z) = (y, z, x)$ to permute the tuple coordinates) or to the right (using *right* $(x, y, z) = (z, x, y)$), and then either optionally flipping the sign (using *inv* $(x, y, z) = (-x, -y, -z)$, or else the identity function, *id*, if no sign change is required). In the diagram, for example, the two new directions on the left are obtained using a left rotation of the coordinates, while those on the right use a right rotation. In addition, the second and fourth new directions involve a change of sign, while the first and third keep the same sign as the original. Given this observation, we can define the following function that computes all of the new directions that are possible after a section with direction *dir*.

```
newDirs     :: Direction → [Direction]
newDirs dir = [sig (rot dir) | rot ← [left, right], sig ← [id, inv]]
  where left, right, inv :: Direction → Direction
        left (x, y, z)  = (y, z, x)
        right (x, y, z) = (z, x, y)
        inv (x, y, z)   = (−x, −y, −z)
```

Note that we do not require this definition to work for arbitrary inputs, just for the six specific *Direction* vectors in which one coordinate is either 1 or −1 and the others are zero.

In practice, by expanding the list comprehension and inlining the uses of *left*, *right*, *id*, and *inv*, we can show that the definition of *newDirs* can be written more compactly as follows:

```
newDirs (x, y, z) = [(y, z, x), (−y, −z, −x), (z, x, y), (−z, −x, −y)]
```

Nevertheless, as a matter of (admittedly subjective) programming style, we prefer the original definition because it provides useful structural information that is lost in the process of deriving the shorter version. For example, the use of a list comprehension in the first definition makes it easy to see, at a glance, that each of the results is produced by combining a rotation (either *left* or *right*) with an optional sign change (either *id* or *inv*). By comparison, the form of the second definition provides no direct insight as to why each of the four elements in the result list was chosen, and it requires a more careful inspection to check that the details are correct.

7 Describing complete puzzles

Building on the definitions in the previous sections, we can now give a general framework for describing instances of the snake cube puzzle as values of the following datatype.

```
data Puzzle = Puzzle { sections :: [Int],
                       valid    :: Position → Bool,
                       initSoln :: Solution,
                       initDir  :: Direction }
```

Each puzzle specifies a list of *sections* that describes the flattened puzzle structure as well as a predicate that identifies the *valid* positions within the solved three-dimensional puzzle. We also include a field, *initSoln*, that will be used as the starting point for any solutions that we compute. This will typically only be used to specify the position of the first small cube, but it can also be used to constrain a puzzle by fixing the positions of some initial sections. (See Section 9 for one application of this.) Finally, each puzzle specifies an initial direction, *initDir*, that should fit the initial solution. In particular, the first section of a puzzle *p* must always be placed along one of the directions in *newDirs* (*initDir p*).

For example, the standard snake puzzle that is shown in the preceding illustrations can be described as follows (we have already argued that the first small cube must be placed in one of the corners of the large cube, hence the choice of [[(1, 1, 1)]] as an initial solution):

```
standard :: Puzzle
standard = Puzzle { valid    = inCube 3,
                    initSoln = [[(1, 1, 1)]],
                    initDir  = (0, 0, 1),
                    sections = snake }
```

Other variations of the puzzle can be described as modifications to this basic structure. For example, Creative Crafthouse, a Florida company that distributes a wide range of wooden puzzles, manufactures a “Mean Green” variant that they characterize as being more difficult than the standard snake. One possible explanation for the increased difficulty is that this version has only 16 sections instead of the 17 sections in the standard snake, potentially giving less flexibility for folding.

```
meanGreen :: Puzzle
meanGreen = standard { sections = [3, 3, 2, 3, 2, 3, 2, 2, 2, 3, 3, 3, 2, 3, 3, 3] }
```

This example, like most of the others in this section, uses Haskell’s update syntax; in this case, we define *meanGreen* as a variant of *standard* that differs only in its list of *sections*.

The previously mentioned king cube variant of the puzzle can also be described in this framework. It differs from the *standard* by targeting a 4×4×4 cube with 46 individual sections, each of which contains either 2, 3, or 4 small cubes.

```
king :: Puzzle
king = standard { valid    = inCube 4,
                  sections = [3, 2, 3, 2, 2, 4, 2, 3, 2, 3, 2, 3, 2, 2, 2,
                              2, 2, 2, 2, 2, 3, 3, 2, 2, 2, 2, 2, 3, 4, 2,
                              2, 2, 4, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2] }
```

Unlike *standard*, where we can be sure that the first small cube will be placed in the corner of the large cube in any valid solution, it is possible to find solutions for the king cube where neither the first nor the last small cube is a corner in the solved puzzle. To allow for this in our framework, we must define distinct *Puzzle* values for each different starting position. Given the specific numbers in *sections king*, however, we can argue by symmetry that there are only two other starting positions that need to be considered in a search for all possible solutions (and, in fact, it turns out that $king_2$ has no solutions):

```
king₁, king₂ :: Puzzle
king₁        = king { initSoln = [[(2, 1, 1)]] }
king₂        = king { initSoln = [[(1, 2, 2)]] }
```

Another way to create a variant of a puzzle is by reversing the order of the sections:

```
reversePuzzle   :: Puzzle → Puzzle
reversePuzzle p = p { sections = reverse (sections p)}
```

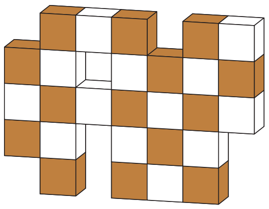
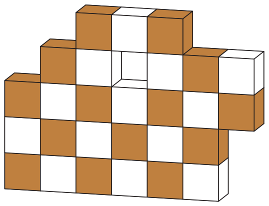
For puzzles like *standard* and *king*, where the only solutions both begin and end with pieces in the corners of the larger cube, applying *reversePuzzle* does not change any fundamental aspects of solvability. However, we sometimes find that the sequences of assembly instructions that we get for *reversePuzzle p* using the methods in Section 8 are easier to follow in practice than those that we get for *p*. (Or, conversely, harder to follow; for example, there is a certain point in our solution for *reversePuzzle standard* that requires some awkward manipulation to avoid a conflict with the 'dangling tail', as suggested in Section 4. The solution that we obtain for *standard*, however, can be followed without any such problems.) For this reason, *reversePuzzle* can be a useful tool in finding practical solutions to snake cube puzzles. Reversing a puzzle can also have a significant impact on the running time of our solver because it forces a different view of the search space. For example, in the next section, we will see that it takes approximately eight times longer to enumerate the solutions to *reversePuzzle king* than it does to enumerate essentially the same set of solutions to *king*.

One more challenge that can be applied to any of the previous puzzles is to find the most compact, flat form that has all of the sections in a single level. We can construct the flat variant of a puzzle by using a *valid* predicate that only allows

*Position*s with $z == 1$.

```
flat       :: Puzzle → Puzzle
flat  p = p { valid = (\(x, y, z) → z == 1)}
```

Using the tools presented in the next section, we can determine that there are 22,768 distinct solutions to *flat standard*, for example, but only 16 that give the most compact layout possible. Once we allow for the inherent symmetries (generated by a reflection, and by rotations through multiples of 90°), we can divide these numbers by a factor of 8, and see that there are really only two distinct, most compact solutions out of 2,846, as shown in the following diagrams.

These diagrams were constructed by (a) using the function described in the next section to enumerate all solutions to *flat standard*; (b) scanning that list to find the most compact solutions (a simple but useful programming exercise for the reader); and then (c) using the methods that will be described in Section 10 to produce the illustrations.

8 Solving puzzles

In this section we describe a method for solving the snake cube and related puzzles using a simple, brute force algorithm.

Suppose that we have a particular puzzle, p; that we have already placed a number of sections to construct a given (partial) solution, *soln*; and that the next section has length *len*. In this setting, we can find the start position for the next section using *start* = *head* (*head soln*). If we pick a particular direction, *dir*, for the new section, then we can calculate the positions *sect* = *section start dir len* that the new section will occupy and use that to produce an extended solution, *sect* : *soln*. Of course, for this to be acceptable, we must ensure that all of the positions in *sect* are *valid* in the given puzzle, and we must also check that none of the positions in *sect* have already been occupied by other sections in the starting solution, *soln*. The following definition captures these ideas:

```
extend                  :: Puzzle → Solution → Direction → Int → [Solution]
extend  p soln dir len = [sect  : soln | let start = head (head soln)
                                              sect  = section start dir len,
                                          all (valid p) sect,
                                          all (‘notElem‘ concat soln) sect]
```

Note that our definition of *extend* uses a special form of list comprehension that has only a local definition and two Boolean guards to the right of the vertical bar.

This is an elegant and compact Haskell idiom for describing a list with at most one element; the extended solution, *sect* : *soln*, is included only when both of the guards are *True*.

We can now construct the desired function that solves an arbitrary puzzle, enumerating the list of all of its solutions:

```
solutions    :: Puzzle → [Solution]
solutions p = solve (initSoln p) (initDir p) (sections p)
  where solve              :: Solution → Direction → [Int] → [Solution]
        solve soln dir [] = [soln]
        solve soln dir (len : lens)
                           = concat [solve soln′ newdir lens
                                       | newdir ← newDirs dir,
                                         soln′ ← extend p soln newdir len]
```

The main function defined here, *solutions*, is simply a wrapper that extracts the necessary components of the input puzzle that are needed as arguments to a worker function, *solve*. The latter maintains a partial solution, *soln*, and a current direction, *dir*, as it iterates through the list of puzzle sections. If there are no remaining puzzle sections, then we have placed all of the puzzle pieces and can output *soln* as a full solution. Otherwise, there is at least one puzzle section of length *len* that must still be placed: we consider each possible direction, *newdir*, for that section; try to *extend* the current solution; and then recurse to find places for the remaining puzzle sections.

Our use of lists and list comprehensions gives us a particularly compact and elegant way to describe these functions. In effect, *solutions* is constructing and searching a large tree structure, but all we see as the output is a lazily generated list of complete solutions. Experienced readers will recognize this approach as an instance of the programming technique that was described by Wadler (1985) as showing "how to replace failure by a list of successes." This same idea has been used in other areas, for example, as an alternative to exception handling; in the implementation of theorem proving tactics; and as a foundation for the construction of parser combinator libraries.

Using these functions, we can compute *length* (*solutions standard*) = 4, and can enumerate the sequence of steps in any one of those solutions, such as:

```
head (solutions standard)
  = [[(3, 3, 3), (2, 3, 3)], [(1, 3, 3)], [(1, 2, 3)], [(2, 2, 3), (2, 2, 2)],
     [(2, 2, 1)], [(2, 1, 1), (2, 1, 2)], [(2, 1, 3)], [(1, 1, 3)], [(1, 1, 2), (1, 2, 2)],
     [(1, 3, 2), (2, 3, 2)], [(3, 3, 2)], [(3, 2, 2)], [(3, 2, 3)], [(3, 1, 3), (3, 1, 2)],
     [(3, 1, 1), (3, 2, 1)], [(3, 3, 1), (2, 3, 1)], [(1, 3, 1), (1, 2, 1)], [(1, 1, 1)]]
```

These results are produced almost immediately, even in the Hugs interpreter. The other puzzles described in Section 7 can be solved in the same way. Solving the mighty king snake, however, requires considerably more patience: reflecting the larger search space—a search tree of depth 46 rather than 17—it takes a little under seven minutes to compute *solutions king* on a fairly typical laptop, and an astonishing

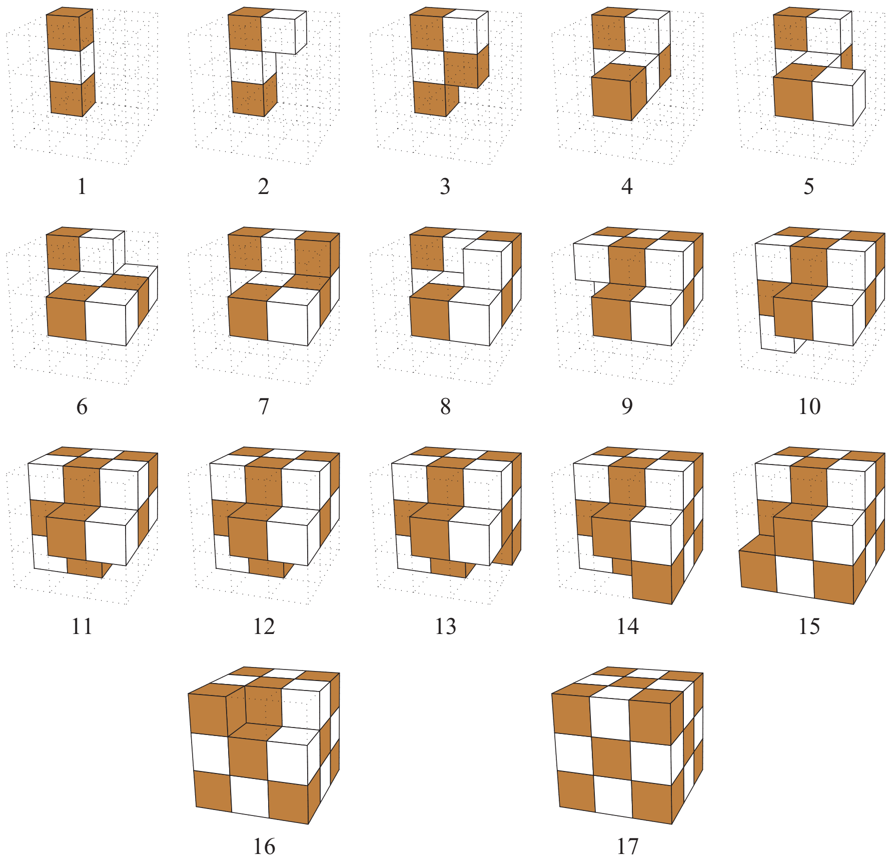


Fig. 3. A solution to the snake cube puzzle.

55 minutes if we switch to *solutions* (*reversePuzzle king*), even when the solver is compiled using GHC.

Of course, it is a little disappointing to present a solution to the snake cube as a list of lists of tuples; we would much prefer to be able to visualize the solution in graphical form. Fortunately, it is not too difficult to convert our computed solution into the sequence of graphics shown in Figure 3, adding one additional puzzle section at each step.

This version is much easier to follow in practice than the original sequence of tuples. Then again, it is hard to capture the precise details of a solution in a sequence of three-dimensional sketches. For example, the reader may note that there is no apparent difference between the diagrams for Steps 11 and 12. But, once again, if you have the puzzle in hand and, in this case, look ahead to the diagram for Step 13, then it is easy to infer the appropriate action for Step 12, and then proceed to complete the puzzle. Success!

Further details about the methods that we use to construct the diagrams shown in Figure 3, as well as some of the other illustrations in this paper, are provided in Section 10.

9 Leveraging symmetry to eliminate solutions

As mentioned in the previous section, our algorithm finds four solutions to the standard snake cube puzzle. But, in fact, these are really just four variations of the same solution that are equivalent under symmetry. To see why this occurs, note that there are two possible directions for the first puzzle section (either $(1, 0, 0)$ or $(0, 1, 0)$, because we have chosen *initDir standard* = (0, 0, 1)). And in each of those two cases, there are two possible choices of direction for the second puzzle section. This gives four distinct ways to start a solution to the puzzle, but all of them include the L-shaped configuration, possibly rotated or reflected, that was shown in Step 2 of Figure 3. As we proceed with each subsequent step on any one of those four configurations, there is a corresponding step in each of the others.

The redundancy that we see here is a consequence of the inherent symmetries of the cube, or, from a different perspective, of the essentially arbitrary choices that we have made in selecting our coordinate system. One way to avoid this is to fix the positions of the first two puzzle sections before attempting to solve the rest. And although the details can be a little fiddly, it is not too difficult, in principle, to adjust the description of any specific puzzle in this way, changing the *initSoln* field to include specific positions for the initial sections and dropping the corresponding entries from the *sections* field. Happily, we can describe this process in a general manner as a transformation on *Puzzle* values:

```
advance   :: Puzzle → Puzzle
advance p = head [p { initDir  = newdir,
                      initSoln = soln′,
                      sections = tail (sections p) }
                  | newdir ← newDirs (initDir p),
                    soln′ ← extend p (initSoln p) newdir (head (sections p))]
```

The key idea here is that *advance p* represents the same puzzle as *p*, but forces an arbitrary choice for the first step toward a solution. The overall structure seen here is very similar to the definition of the *solutions* function that was described in the previous section, and it uses the same *extend* operator to find valid extensions of the initial solution in *p*. The essential differences are that (a) instead of making a recursive call, we package up the results in a new *Puzzle*; and (b) instead of exploring the list of all possible solutions, we make an "arbitrary" choice by using *head* to pick the first valid extension.

The *advance* operation can be used, for example, to show that there is really only one way to solve the *standard* snake cube puzzle:

$$(\mathit{length} \cdot \mathit{solutions} \cdot \mathit{advance} \cdot \mathit{advance})\ \mathit{standard} = 1$$

Given the intuition that motivates *advance*, this approach seems more appealing than simply noticing that *solutions standard* includes multiple elements and hoping that the *head* will be a good representative for all of them. On the other hand, there are also situations where *advance* is not appropriate because it could inadvertently eliminate portions of the search space, potentially causing us to miss the solutions

```
showCubes        ::  [Position] → String
showCubes        =   unlines · concat · zipWith cube2sk (cycle ["brown", "white"])

cube2sk          ::  String → Position → [String]
cube2sk col a    =   [prefix ++ concat (map show f) | f ← faces]
  where prefix     =   "polygon[fill=" ++ col ++ "]"
        faces      =   [[a, d, g, c], [b, e, h, f], [a, b, f, d],
                        [c, g, h, e], [a, c, e, b], [d, f, h, g]]
        (x, y, z)  =   a
        b          =   (x, y, z − 1)
        c          =   (x, y − 1, z)
        d          =   (x − 1, y, z)
        e          =   (x, y − 1, z − 1)
        f          =   (x − 1, y, z − 1)
        g          =   (x − 1, y − 1, z)
        h          =   (x − 1, y − 1, z − 1)
```

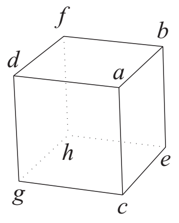


Fig. 4. Functions for generating a Sketch diagram from a list of small cube positions.

that we were seeking. One example of this occurs with the $king_1$ puzzle: it would not be a good idea to make an arbitrary choice between the two possible positions for the first puzzle section in this case because they are not related by a symmetry within the overall cube. As such, *advance* should be used with care.

10 Visualizing solutions to the snake cube puzzle

One of the attractive aspects of working on the problems described in this paper is the ability to hold and play with an actual, physical snake cube puzzle as you think about and develop a program for solving it. This works particularly well in a classroom setting where it is possible to hand out copies of the puzzle for the students to experiment with. But even without a physical copy, the snake cube puzzle still has a strong visual appeal, as illustrated by some of the diagrams in this paper.

In this section, we give a brief overview of how these diagrams were constructed. In most cases, we started by writing some Haskell code—to process computed solutions, for example. But the real work was done using Gene Ressler's Sketch tool (Ressler, 2012) and Tantau's (2010) Ti*k*Z package. The former takes text files containing simple three-dimensional scene or object descriptions and uses those to generate code for the latter, which will render the images in a LaTeX document. For example, the code in Figure 4 shows how we can take a list of *Position* values, each describing a small cube, and generate a corresponding list of `polygon` definitions for use with Sketch. The *showCubes* function essentially just pairs up each small cube with a colour, alternating between dark and light (rendered as brown and white in the code and in the online version of this paper), and then concatenates the resulting list of Sketch commands into a single Haskell *String*. The *cube2sk* function generates the Sketch code for individual cubes, each of which is described by a list of six (square) polygons, one for each face. (Sketch uses hidden surface removal techniques, together with a specification of the camera position and orientation, to determine exactly which of these faces will be visible in each generated diagram.) The only subtlety here is that, by default, Sketch polygons are one-sided, and only visible on the side from which the vertices appear in a counter clockwise order. The small diagram shows our scheme for labelling the eight vertices of the cube, which can also be used to check that each of the faces is described correctly. For example, the expression *showCubes* [(1, 1, 1)] produces a string containing the following Sketch code.

```
polygon[fill=brown](1,1,1)(0,1,1)(0,0,1)(1,0,1)
polygon[fill=brown](1,1,0)(1,0,0)(0,0,0)(0,1,0)
polygon[fill=brown](1,1,1)(1,1,0)(0,1,0)(0,1,1)
polygon[fill=brown](1,0,1)(0,0,1)(0,0,0)(1,0,0)
polygon[fill=brown](1,1,1)(1,0,1)(1,0,0)(1,1,0)
polygon[fill=brown](0,1,1)(0,1,0)(0,0,0)(0,0,1)
```

For a given *Solution*, we can construct a list of lists of *Position* values to describe the set of cubes that has been placed at each step in the solution (as seen, of course, in Figure 3). The task of converting a *Solution* into a list of that type can be described using standard Haskell list processing functions.

$$\begin{aligned}&\mathit{steps} :: \mathit{Solution} \rightarrow [[\mathit{Position}]]\\&\mathit{steps} = \mathit{tail} \cdot \mathit{reverse} \cdot \mathit{map}\ \mathit{reverse} \cdot \mathit{scanr1}\ (+\!\!+)\end{aligned}$$

The most important detail here is to ensure that the small cubes in each output list are correctly ordered so that each one appears with the appropriate colour when displayed using *showCubes*. This is accomplished by using *scanr*1 to build up a list of "partial sums," starting from the right of the solution, and then mapping the *reverse* function over each resulting list. The leftmost use of *reverse* ensures that the solutions are displayed in the appropriate order, ending with the completed puzzle. Finally, we use *tail* to drop the very first step in the solution, which, otherwise, would just show the position of the initial small cube (as in the diagram for $soln_0$ in Figure 2). With these tools in place, we can define a function that takes a puzzle as input and generates a sequence of Sketch diagrams, each described by a single string, for the first complete solution:

$$\begin{aligned}&\mathit{showSteps} :: \mathit{Puzzle} \rightarrow [\mathit{String}]\\&\mathit{showSteps} = \mathit{map}\ \mathit{showCubes} \cdot \mathit{steps} \cdot \mathit{head} \cdot \mathit{solutions}\end{aligned}$$

To complete the task, we require some additional Sketch code (to draw grid lines, and set appropriate perspective views, for example) and some more Haskell code (to wrap the outputs from *showSteps* with the Sketch code and write the results to a corresponding sequence of output files). None of this, however, is difficult (or interesting!), and so for further details we refer the reader instead to the supplementary material section at the end of the paper for links to the source code (with more solutions and other related materials).

11 Further development

In this paper, we have described a concise and elegant functional program, written in Haskell, that computes solutions for the snake cube puzzle and is readily adapted to other variants. These tools provide a platform for investigating and understanding a range of snake cube puzzles. For example, we have used the solver to compute four distinct solutions for the 4×4×4 cube (one for *king* and three for $king_1$). Based on available resources and published solutions on the Internet, it is possible that only three of these solutions were previously known. Our program is also attractive in an educational setting. One reason for this is that the code reflects some fundamental characteristics of the functional style, identifying key abstractions such as *Position*s, *Direction*s, *Solution*s, and *Puzzle*s, and defining a small collection of operators for manipulating and working with these values. In addition, we benefit from working on a problem that not only has visual appeal but also a strong tactile component for those with access to a physical copy of the puzzle.

There are several opportunities for building on the ideas presented here. On a practical front, for example, it would be useful, particularly for the larger examples, to show multiple views of a puzzle at each step of assembly so that there are fewer (or, ideally, no) places where details of the next step are hidden from view. One way to accomplish this is by applying a geometric transformation to the cubes in any given solution step. For example, given a list *cubes* :: [*Position*], the following expression will add in the second copy of the same step, but rotated through 90°, and translated to the right of the original.

$$cubes +\!\!+\, map\ (\backslash(x, y, z) \rightarrow (z + 7, y, 4 - x))\ cubes$$

(In practice, we need to do some additional work to account for colouring, but the same basic methods/transformations still apply.) An alternative approach would be to make use of a three-dimensional graphics library, such as OpenGL, to build an interactive viewer for puzzle solutions.

A shortcoming of the approach that we have used in this paper is the need to specify an initial direction and cube position as part of each *Puzzle* data structure. By providing these details explicitly, we encode some geometric insights about the structure of individual puzzles, and we can avoid generating large sets of solutions that are all equivalent up to symmetry. We do not know, however, if a more elegant approach is possible, perhaps starting from a slightly higher level description of a puzzle, and computing a full set of solutions automatically, without the need to explore multiple options by hand.

Another practical concern is the "dangling tail" problem that was described in Section 4. This can occur because our method of finding solutions does not account for the possibility that unplaced sections of the puzzle might conflict with parts of the puzzle that have already been put in position. What is needed here, however, is not a new method of finding solutions, but instead an algorithm for finding an appropriate set of folding steps, with a previously calculated final solution as the goal, and the flexibility to adjust the angle between any pair of adjacent sections at each step in the assembly.

In this paper, we have described two distinct versions of the 3×3×3 snake cube puzzle: *standard* and *meanGreen*. Several others have been manufactured as physical puzzles. But, as a final challenge, particularly for those with an interest in combinatorics, how many distinct variants are possible, and how many of those have only one unique solution once we account for symmetry? These questions could potentially be tackled by a brute force method, generating all of the possible integer lists containing 2s and 3s that satisfy the sanity check described in Section 2, and then feeding those candidate designs as inputs to our solver. But is there a more efficient solution? And can such an approach be scaled up to larger cube sizes, or even to more general $n \times m \times p$ puzzles?

Acknowledgments

The author wishes to thank the Campbell family for introducing him to the snake cube puzzle. Thanks also to Jeremy Gibbons and to the anonymous referees for their comments that have helped to improve the presentation. Finally, we note that the original snake cube design—under the name "Block Puzzle," and credited to Allen F. Dreyer of Richmond, California—was submitted to the US Patent Office on June 11, 1962, and was eventually awarded as patent number 3,222,072 on December 7, 1965.

Supplementary Material

For supplementary materials for this article please visit http://dx.doi.org/10.1017/S0956796813000014 (also available at http://web.cecs.pdx.edu/~mpj/snakecube at the time of writing).

References

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

Ressler, G. (2012) *Sketch: Simple 3D Sketching, Version 0.3*. Available at: http://www.frontiernet.net/∼eugene.ressler/. Accessed 1 March 2013.

Tantau, T. (2010) *The TikZ and* PGF *Packages Manual for version 2.10*. Institut für Theoretische Informatik, Universität zu Lübeck. Available at: http://sourceforge.net/projects/pgf. Accessed 1 March 2013.

Wadler, P. (1985) How to replace failure by a list of successes. In *The Second International Conference on Functional Programming Languages and Computer Architecture (FPCA 1985)*, Lecture Notes in Computer Science, vol. 201. Nancy, France: Springer-Verlag.