The Monad.Reader/Issue5/Generating Polyominoes

From HaskellWiki

< The Monad.Reader | Issue5

Contents

1 Generating Polyominoes

by Dominic Fox

1.1 A simple algorithm for enumerating the polyominoes of a given rank, implemented in Haskell

Readers of Arthur C. Clarke's novel *Imperial Earth* may remember the "Pentomino" puzzle that beguiles the book's protagonist, Duncan Mackenzie. The object of the puzzle is to arrange twelve puzzle pieces, each made of five congruent squares, in a rectangular box. For a box with the dimensions six squares by ten, there are many possible solutions; for a box with the dimensions three squares by twenty, there are only two. The young Duncan Mackenzie's initial encounter with the six-by-ten version of the puzzle teaches him a lesson about the uses of intuition in mathematical problem solving. Solving the harder three-by-twenty version requires that intuition be supplemented by technique: it is a task for a more developed formal imagination.

In this article, we will take a first step towards the construction of a solver for the pentomino puzzle. We will begin with the task of *enumerating* the pentominoes, and other similar shapes. Duncan's puzzle has twelve pieces, and these pieces are the twelve distinct "free" pentominoes; that is, they are all the shapes that can be made by gluing five squares together orthogonally along their edges, that are different from each other even when rotated, reflected and/or translated. For example, under this definition, the following shapes are all the "same" free pentomino (the "F" pentomino):

```
**       *       *       *      **     *       *       *
 **     ***     **      ***     **     ***     **     ***
 *      *        **      *       *      *      **       *
```

If we wanted to find out what the other eleven free pentominoes were, we would need both a way of generating new candidate pentominoes, and a way of recognising when one of these new candidates was the same - under rotation, reflection and/or translation - as a pentomino we had already found.

The set of twelve pentominoes belongs to a larger set of "polyforms" known as polyominoes. The name "polyominoes" was first given to these shapes by the mathematician Solomon W. Golomb, whose book *Polyominoes* is a fascinating introduction to the subject of polyomino puzzles and techniques for solving them. Probably the best-known polyominoes are the five "tetrominoes" that appear in the game "Tetris":

```
****     **       ***      **      ***
         **        *      **      *
```

As their name suggests, each of the tetrominoes is made of four squares orthogonally connected along their edges. Every pentomino is a tetromino with an additional square attached to one of its constituent squares along an edge not already taken by another square. In the same manner, every tetromino is a *tromino* - a polyomino made of three squares, or a polyomino of rank three - with an additional square attached to it. Every tromino is a domino with an extra square, and a domino is a monomino with an extra square.

Given a definition of the polyominoes of rank zero as an empty set, and the polyominoes of rank one as a set containing a single monomino, we can define the polyominoes of rank *n* as all of the polyominoes that can be created by adding an extra square to some polyomino of rank *n-1*, that are different from one another under translation, rotation and reflection. This definition gives us the outline of a simple algorithm for enumerating all of the polyominoes of rank *n*.

1.2 Implementing the algorithm in Haskell

Implementing such an algorithm in Haskell involves generating a list of candidate polyominoes of rank *n*, based on a list of known polyominoes of rank *n-1*, and removing from that list all polyominoes that are found to be the same as another included polyomino after they have been translated, rotated and/or reflected. The module described below does just that, providing a function,

```
rank
```

, that generates all the polyominoes of a given rank. The approach taken by this module

is the least efficient of the three discussed in the wikipedia article on polyominoes (http://en.wikipedia.org/wiki/Polyomino) , but it is easy to follow and provides a nice illustration of the usefulness of function composition in expressing algorithms in Haskell.

We begin with the module declaration, some imports, and a couple of types:

```
> module Generator (rank) where

> import List (sort)
> import Data.Set (setToList, mkSet)

> type Point = (Int, Int)
> type Polyomino = [Point]
```

In order to compare two candidate polyominoes and determine whether they are the same, we introduce some functions that will convert any candidate polyomino into a normalised, "canonical" form. The first kind of normalisation we perform is to translate the candidate polyomino such that its bottom and left edges are aligned with the x and y axes:

```
> minima :: Polyomino -> Point
> minima (p:ps) = foldr (\(x, y) (mx, my) -> (min x mx, min y my)) p ps

> translateToOrigin :: Polyomino -> Polyomino
> translateToOrigin p =
>     let (minx, miny) = minima p in
>         map (\(x, y) -> (x - minx, y - miny)) p
```

The second kind of normalisation we perform is to take all of the rotated and reflected forms of the translated polyomino, and sort them in order to find the "bottommost and leftmost" form.

```
> rotate90 :: Point -> Point
> rotate90 (x, y) = (y, -x)

> rotate180 :: Point -> Point
> rotate180 (x, y) = (-x, -y)

> rotate270 :: Point -> Point
> rotate270 (x, y) = (-y, x)

> reflect :: Point -> Point
> reflect (x, y) = (-x, y)

> rotationsAndReflections :: Polyomino -> [Polyomino]
> rotationsAndReflections p =
>     [p,
>      map rotate90 p,
>      map rotate180 p,
>      map rotate270 p,
>      map reflect p,
>      map (rotate90 . reflect) p,
>      map (rotate180 . reflect) p,
>      map (rotate270 . reflect) p]

> canonical :: Polyomino -> Polyomino
> canonical = minimum . map (sort . translateToOrigin) . rotationsAndReflections
```

The function

```
canonical
```

is constructed by composing together other functions, so as to create a kind of pipeline of transformations which are applied to an initial value in right-to-left order. Thus,

```
rotationsAndReflections
```

takes a polyomino and returns a list of all the rotated and reflected forms of that polyomino. The next stage in the pipeline uses

```
map
```

to apply the composed function

```
sort . translateToOrigin
```

to each of these forms in turn, translating them so that their bottommost and leftmost square is in the position (0, 0) and sorting the list of points that makes up each polyomino so that the bottommost and leftmost square appears first in the list, and the rightmost and topmost appears last. We then use

```
minimum
```

take the lowest-ordered polyomino in the resulting list of translated and internally-sorted polyominoes. Given a polyomino of rank *n*, we would like to know what polyominoes of rank *n+1* can be generated by attaching another point to it. We therefore need to find all the unique places where another point can be attached. This definition of

```
unique
```

is efficient enough for short lists:

```
> unique :: (Eq a) => [a] -> [a]
> unique [] = []
> unique (x:xs) = foldr (\y ys -> if y `elem` ys then ys else y:ys) [x] xs
```

We also define an alternative

```
unique'
```

function for removing duplicates from a long list of polyominoes. This alternative function simply converts a list to a set, and then back into a list:

```
> unique' :: (Ord a) => [a] -> [a]
> unique' = setToList . mkSet
```

The function

```
contiguous
```

returns all the orthogonally adjacent points of a given point.

```
> contiguous :: Point -> [Point]
> contiguous (x, y) =
>     [(x - 1, y),
>      (x + 1, y),
>      (x, y - 1),
>      (x, y + 1)]
```

Given these two functions, we can find the contiguous points for each point in a polyomino. We're only interested in points that fall outside of the original polyomino, so we filter out any that are already taken.

```
> newPoints :: Polyomino -> [Point]
> newPoints p =
>     let notInP = filter (`notElem` p) in
>         unique . notInP . concatMap contiguous $ p
```

Now we can generate a list of new polyominoes using newPoints. We'll put them all into canonical form, and only take the unique ones.

```
> newPolys :: Polyomino -> [Polyomino]
> newPolys p = unique . map (canonical . (:p)) $ newPoints p
```

Again, this function is composed out of smaller functions:

```
newPoints
```

feeds its results to a function that adds each new point to the initial polyomino and returns the resulting new polyomino in canonical form, and

```
unique
```

then removes all duplicates from the resulting list of new polyominoes.

We now define the first two ranks of polyominoes. The zeroth rank of polyominoes is the empty list. The first rank of polyominoes is the monominoes, which is a list containing a single element:

```
> monomino = [(0, 0)]
> monominoes = [monomino]

> rank :: Int -> [Polyomino]
> rank 0 = []
> rank 1 = monominoes
```

The next rank of monominoes can be generated from the rank before it. We find all the new polyominoes in rank *n* that can be generated from each polyomino in rank *n -1*, concatenate them together into a single list, and throw out any duplicates.

```
> rank n = unique' . concatMap newPolys $ rank (n - 1)
```

That is the entire algorithm!

1.3 Conclusions

The above program uses function composition to express an algorithm as a pipeline made up of three kinds of function: functions of type

```
a -> [a]
```

that generate new values from some source value, functions of type

```
[a] -> [a]
```

that filter a list of values to remove duplicate or unwanted values, and functions of type

```
[a] -> a
```

that extract a single result from a list of values. Many algorithms based on the blind generation of a list of new candidate results, which is then filtered to extract only the valid answers, can easily be expressed in this style. It also lends itself to recursion, as each new generation of results can be fed back into the pipeline to produce another generation. Haskell's lazy evaluation and garbage collection can help to minimize the creation and retention of large data structures during the course of such pipelined processing. However, there are limitations to how much work lazy evaluation can help us to avoid. In the function

```
canonical
```

in the program above, for example, the entire list consumed by

```
minimum
```

must be constructed before the lowest-ordered value can be found.

As I mentioned earlier, this is not by any means the best algorithm for enumerating polyominoes. It is better if possible to avoid generating values that will only be thrown away, as these still have to be checked before being discarded. It is worth looking for a heuristic that can limit the number of invalid or duplicate candidate results generated, as this will often have a greater impact on performance than any optimisation that can be applied to the validation stage.

1.3.1 For Further Investigation

Can an algorithm for enumerating polyforms made of cubes be constructed as easily as one for enumerating polyforms made of squares? How about polyforms of arbitrary

dimension? And what about two-dimensional polyforms made out of other shapes, such as hexagons?

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue5/Generating_Polyominoes"

Category: Article
