JFP **16** (6): 671–679, 2006.
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL
A program to solve Sudoku

RICHARD BIRD
Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
email: bird@comlab.ox.ac.uk

There's no maths involved. You solve
the puzzle with reasoning and logic.
Advice on how to play Sudoku,
The Independent Newspaper

1 Introduction

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku \quad :: \quad Board \rightarrow [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive N, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test $blank :: Char \rightarrow Bool$ for determining whether a given entry

is blank. For concreteness, we can take

$$\begin{array}{lcl} boardsize & = & 9 \\ boxsize & = & 3 \\ cellvals & = & \text{“123456789”} \\ blank & = & (= \text{‘.’}) \end{array}$$

Changing cell values, e.g. to “TONYBLAIR” is easy.

2 Specification

The first aim is to write down the simplest and clearest specification of Sudoku without regard to how efficient the result might be. Such a specification will help us focus ideas on how a more efficient solution might be obtained, as well as being a starting point for program manipulation.

One possibility is first to construct a list of all correctly completed boards, and then to test the given board against these boards to identify those whose entries match the given ones. Another possibility, and the one we will take, is to start with the given board and to generate all possible completions. Each completed board is then tested to see if it is correct, that is, does not contain duplicate entries in each row, column or box.

A board is a matrix of characters:

$$\begin{array}{lcl} \textbf{type } Matrix\ a & = & [[a]] \\ \textbf{type } Board & = & Matrix\ Char \end{array}$$

Strictly speaking a given board should first be checked to see that every non-blank entry is an element of *cellvals*. Invalid boards should be rejected. However, for simplicity we will assume that the given board does satisfy the basic requirements.

The function *correct* tests whether a filled board, that is, one containing no blank characters, has different entries in each row, column and box:

$$\begin{array}{lcl} correct & :: & Board \to Bool \\ correct\ b & = & all\ nodups\ (rows\ b) \wedge \\ & & all\ nodups\ (cols\ b) \wedge \\ & & all\ nodups\ (boxs\ b) \end{array}$$

The function *nodups* can be defined by

$$\begin{array}{lcl} nodups & :: & Eq\ a \Rightarrow [a] \to Bool \\ nodups\ [\,] & = & True \\ nodups\ (x : xs) & = & notElem\ x\ xs \wedge nodups\ xs \end{array}$$

2.1 Rows, columns and boxes

If a matrix is given by a list of its rows, the function *rows* is just the identity function on matrices:

$$\begin{array}{lcl} rows & :: & Matrix\ a \to Matrix\ a \\ rows & = & id \end{array}$$

We have, trivially, that $rows \cdot rows = id$.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

The function *cols* computes the transpose of a matrix. One possible definition is:

$$\begin{array}{lcl} cols & :: & Matrix\ a \to Matrix\ a \\ cols\ [xs] & = & [[x] \mid x \leftarrow xs] \\ cols\ (xs\ :\ xss) & = & zipWith\ (:)\ xs\ (cols\ xss) \end{array}$$

We also have $cols \cdot cols = id$.

The boxes of a matrix can be computed by:

$$\begin{array}{lcl} boxs & :: & Matrix\ a \to Matrix\ a \\ boxs & = & map\ ungroup \cdot ungroup \cdot map\ cols \cdot group \cdot map\ group \end{array}$$

The function *group* groups a list into component lists of length *boxsize*, and *ungroup* takes a grouped list and ungroups it:

$$\begin{array}{lcl} group & :: & [a] \to [[a]] \\ group & = & groupBy\ boxsize \\ \\ ungroup & :: & [[a]] \to [a] \\ ungroup & = & concat \end{array}$$

We omit the definition of *groupBy*. Using $ungroup \cdot group = id$ and $group \cdot ungroup = id$, it is easy to show that $boxs \cdot boxs = id$ by simple equational reasoning.

2.2 Generating choices and matrix cartesian product

The function *choices* replaces blank entries in a board with all possible choices for that entry. Using *Choices* as a synonym for [*Char*], we have

$$\begin{array}{lcl} choices & :: & Board \to Matrix\ Choices \\ choices & = & map\ (map\ choose) \\ choose\ e & = & \textbf{if}\ blank\ e\ \textbf{then}\ cellvals\ \textbf{else}\ [e] \end{array}$$

Of course, not every possible choice is valid for each cell, and we will return to this point later on.

The function *mcp* (matrix cartesian product) generates a list of all possible boards from a given matrix of choices:

$$\begin{array}{lcl} mcp & :: & Matrix\ [a] \to [Matrix\ a] \\ mcp & = & cp \cdot map\ cp \end{array}$$

The function *cp* computes the cartesian product of a list of lists:

$$\begin{array}{lcl} cp & :: & [[a]] \to [[a]] \\ cp\ [\,] & = & [[\,]] \\ cp\ (xs\ :\ xss) & = & [x\ :\ ys \mid x \leftarrow xs,\ ys \leftarrow cp\ xss] \end{array}$$

Note that *cp xss* returns an empty list if *xss* contains an empty list. Thus *mcp cm* returns an empty list if any entry of *cm* is the empty list.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

2.3 Specification

The function *sudoku* can now be defined by

$$\begin{array}{lcl} sudoku & :: & Board \to [Board] \\ sudoku & = & filter\ correct \cdot mcp \cdot choices \end{array}$$

However, this specification is executable in principle only. Assuming about a half of the 81 entries are fixed initially, there are about 9^{40}, or

$$147808829414345923316083210206383297601$$

boards to check! We therefore need a better approach.

3 Pruning the choices

Obviously, not every possible choice is valid for each cell. A better choice for a blank entry in row r, column c and box b is any cell value that does not appear among the fixed entries in row r, column c or box b. An entry in a matrix of choices is fixed if it is a singleton list. The fixed entries in a given row, column or box, are given by

$$\begin{array}{lcl} fixed & :: & [Choices] \to Choices \\ fixed & = & concat \cdot filter\ single \end{array}$$

where $single :: [a] \to Bool$ tests whether the argument is a singleton list. The fixed entries can be removed from a list of choices by

$$\begin{array}{lcl} reduce & :: & [Choices] \to [Choices] \\ reduce\ css & = & map\ (remove\ (fixed\ css))\ css \\ remove\ fs\ cs & = & \textbf{if}\ single\ cs\ \textbf{then}\ cs\ \textbf{else}\ delete\ fs\ cs \end{array}$$

We leave the definition of *delete* to the reader.

Now, how shall we prune the matrix of choices? The aim is to define a function

$$prune :: Matrix\ Choices \to Matrix\ Choices$$

satisfying the equation

$$filter\ correct \cdot mcp = filter\ correct \cdot mcp \cdot prune$$

The function *prune* removes the fixed choices from each row, column or box. The question is a good test of one's programming ability for it seems easy to get into a mess. So, it is worthwhile adding a small pause at this point, to see if the reader can come up with a short definition that meets the requirement.

(Pause)

The calculational programmer would calculate a definition, and that is precisely what we are going to do.

The first step is to rewrite *filter correct* in the form

$$\begin{array}{lcl} filter\ correct & = & filter\ (all\ nodups \cdot boxs) \cdot \\ & & filter\ (all\ nodups \cdot cols) \cdot \\ & & filter\ (all\ nodups \cdot rows) \end{array}$$

The order of the component filters is unimportant.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

Now we send these filters one by one into battle with *mcp*. We will need some weapons, the first of which is the law

$$filter\ (p \cdot f) = map\ f \cdot filter\ p \cdot map\ f$$

which is valid provided $f \cdot f = id$. In particular, the law is valid if f is one of *rows*, *cols*, or *boxs*.

Another useful law is the following one:

$$filter\ (all\ p) \cdot cp \quad = \quad cp \cdot map\ (filter\ p)$$

In words, if we want only those lists all of whose elements satisfy p from a cartesian product, then we can obtain them by taking the cartesian product of the elements satisfying p of the component lists.

We will also need the following facts:

$$\begin{array}{lcl} map\ rows \cdot mcp & = & mcp \cdot rows \\ map\ cols \cdot mcp & = & mcp \cdot cols \\ map\ boxs \cdot mcp & = & mcp \cdot boxs \end{array}$$

These laws are intuitively clear and we will not verify them formally.

We need one final law: the crucial property of the function *reduce* defined above is that

$$filter\ nodups \cdot cp \quad = \quad filter\ nodups \cdot cp \cdot reduce$$

Here is the calculation. Let f be one of *rows*, *cols* or *boxs*:

$$\begin{array}{ll} & filter\ (all\ nodups \cdot f) \cdot mcp \\ = & \quad \{\text{since } filter\ (p \cdot f) = map\ f \cdot filter\ p \cdot map\ f \text{ if } f \cdot f = id\} \\ & map\ f \cdot filter\ (all\ nodups) \cdot map\ f \cdot mcp \\ = & \quad \{\text{since } map\ f \cdot mcp = mcp \cdot f \text{ if } f \in \{boxs, cols, rows\}\} \\ & map\ f \cdot filter\ (all\ nodups) \cdot mcp \cdot f \\ = & \quad \{\text{definition of } mcp\} \\ & map\ f \cdot filter\ (all\ nodups) \cdot cp \cdot map\ cp \cdot f \\ = & \quad \{\text{since } filter\ (all\ p) \cdot cp = cp \cdot map\ (filter\ p)\} \\ & map\ f \cdot cp \cdot map\ (filter\ nodups \cdot cp) \cdot f \\ = & \quad \{\text{property of } reduce\} \\ & map\ f \cdot cp \cdot map\ (filter\ nodups \cdot cp \cdot reduce) \cdot f \\ = & \quad \{\text{since } filter\ (all\ p) \cdot cp = cp \cdot map\ (filter\ p)\ \} \\ & map\ f \cdot filter\ (all\ nodups) \cdot cp \cdot map\ (cp \cdot reduce) \cdot f \\ = & \quad \{\text{since } map\ f \cdot filter\ p = filter\ (p \cdot f) \cdot map\ f \text{ if } f \cdot f = id\} \\ & filter\ (all\ nodups \cdot f) \cdot map\ f \cdot mcp \cdot map\ reduce \cdot f \\ = & \quad \{\text{since } map\ f \cdot mcp = mcp \cdot f \text{ if } f \in \{boxs, cols, rows\}\} \\ & filter\ (all\ nodups \cdot f) \cdot mcp \cdot f \cdot map\ reduce \cdot f \\ = & \quad \{\text{definition of } pruneBy\ f\text{; see below}\} \\ & filter\ (all\ nodups \cdot f) \cdot mcp \cdot pruneBy\ f \end{array}$$

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

The definition of *pruneBy* is

$$\begin{array}{lcl} pruneBy & :: & (MatrixChoices \to MatrixChoices) \to \\ & & (MatrixChoices \to MatrixChoices) \\ pruneBy\, f & = & f \cdot map\, reduce \cdot f \end{array}$$

We have shown that, provided f is one of *rows*, *cols* or *boxs*,

$$filter\ (all\ nodups \cdot f) \cdot mcp \quad = \quad filter\ (all\ nodups \cdot f) \cdot mcp \cdot pruneBy\, f$$

For the final step we need one more law, the fact that we can interchange the order of two *filter* operations:

$$filter\ p \cdot filter\ q \quad = \quad filter\ q \cdot filter\ p$$

This law is not generally valid in Haskell without qualification on the boolean functions p and q, but provided p and q are total functions, as is the case here, the law is OK. Indeed we implicitly made use of it when claiming that the order of the component filters in the expansion of *filter correct* was unimportant.

Now we can calculate, abbreviating *nodups* to *nd* to keep the expressions short:

$$\begin{array}{ll} & filter\ correct \cdot mcp \\ = & \quad \{\text{rewriting } filter\ correct \text{ as three filters}\} \\ & filter\ (all\ nd \cdot boxs) \cdot filter\ (all\ nd \cdot cols) \cdot filter\ (all\ nd \cdot rows) \cdot mcp \\ = & \quad \{\text{calculation above}\} \\ & filter\ (all\ nd \cdot boxs) \cdot filter\ (all\ nd \cdot cols) \cdot filter\ (all\ nd \cdot rows) \cdot mcp \cdot \\ & pruneBy\ rows \\ = & \quad \{\text{interchanging the order of the filters}\} \\ & filter\ (all\ nd \cdot rows) \cdot filter\ (all\ nd \cdot boxs) \cdot filter\ (all\ nd \cdot cols) \cdot mcp \cdot \\ & pruneBy\ rows \\ = & \quad \{\text{using the calculation above again}\} \\ & filter\ (all\ nd \cdot rows) \cdot filter\ (all\ nd \cdot boxs) \cdot filter\ (all\ nd \cdot cols) \cdot mcp \cdot \\ & pruneBy\ cols \cdot pruneBy\ rows \\ = & \quad \{\text{repeating the last two steps one more time}\} \\ & filter\ (all\ nd \cdot rows) \cdot filter\ (all\ nd \cdot boxs) \cdot filter\ (all\ nd \cdot cols) \cdot mcp \cdot \\ & pruneBy\ boxs \cdot pruneBy\ cols \cdot pruneBy\ rows \\ = & \quad \{\text{definition of } filter\ correct\} \\ & filter\ correct \cdot mcp \cdot pruneBy\ boxs \cdot pruneBy\ cols \cdot pruneBy\ rows \end{array}$$

Hence, we can define *prune* by

$$\begin{array}{lcl} prune & :: & MatrixChoices \to MatrixChoices \\ prune & = & pruneBy\ boxs \cdot pruneBy\ cols \cdot pruneBy\ rows \end{array}$$

Readers who gave this solution (or a similar one in which the three components appear in any other order) can award themselves full marks.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

The revised definition of *sudoku* now reads

$$\begin{array}{lcl} sudoku & :: & Board \to [Board] \\ sudoku & = & filter\ correct \cdot mcp \cdot prune \cdot choices \end{array}$$

However, this version remains non-executable in practice. Again, assuming about a half of the 81 entries are fixed and an average of 3 choices/cell is generated by refining choices, there are still 3^{40}, or 12157665459056928801 boards to check. We still need something better.

4 One choice at a time

Humans employ a number of devices for filling in entries when solving Sudoku problems. For example, after pruning a matrix of choices, one or more entries that were previously blank may become fixed. In such a case, we can always prune again to see if more entries are filled in. The calculation above shows that we can have the composition of as many *prune* functions as we like. This is the way the simplest puzzles are solved.

There are also other strategies. For example, again after pruning the choice matrix it may turn out that a single row (or column or box) contains, for example, three entries such as 12, 12 and 123. It is clear that the third entry has to receive 3; if it receives 1 or 2, the first two entries cannot be filled in.

Repeatedly pruning the choice matrix is sensible, but we can combine it with another basic strategy. Rather than applying *mcp* when pruning fails to produce anything new, we can focus on one cell that has at least two choices, and generate a list of matrices in which this cell alone is expanded to each of its possible fixed choices.

Suppose we define a function

$$expand :: Matrix\ Choices \to [Matrix\ Choices]$$

that installs the fixed choices for one cell. This function satisfies the property that

$$mcp \quad \approx \quad concat \cdot map\ mcp \cdot expand$$

where $\approx$ means equality up to a permutation of the answer. After applying *prune*, we can apply *expand* and then apply *prune* again to each of the results. Provided we discard any matrix that becomes blocked (see below), this process can be continued until we are left with a list of matrices, all of whose choices are fixed choices.

4.1 Blocked matrices

A matrix of choices can be *blocked* in that:

- One or more cells may contain zero choices. In such a case *mcp* will return an empty list;
- The same fixed choice may occur in two or more positions in the same row, column or box. In such a case *mcp* will still compute all the completed boards, but the correctness test will throw all of them away.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

Blocked matrices can never lead to a solution. In following the strategy of repeatedly pruning and expanding the matrix of choices, we can identify and discard any blocked matrix. Provided we do this, any remaining matrix that consists solely of fixed choices will be a solution to the puzzle.

Formally, we define *blocked* by

$$\begin{array}{lcl} blocked & :: & Matrix\ Choices \to Bool \\ blocked\ cm & = & void\ cm \vee not\ (safe\ cm) \\ \\ void & :: & Matrix\ Choices \to Bool \\ void & = & any\ (any\ null) \\ \\ safe & :: & Matrix\ Choices \to Bool \\ safe\ cm & = & all\ (nodups \cdot fixed)\ (rows\ cm) \wedge \\ & & all\ (nodups \cdot fixed)\ (cols\ cm) \wedge \\ & & all\ (nodups \cdot fixed)\ (boxs\ cm) \end{array}$$

4.2 Smallest number of choices

A good choice of cell on which to perform expansion is one with the *smallest* number of choices (greater than one of course). We will need a function that breaks up a matrix on the first entry with the smallest number of choices. A matrix that is not blocked is broken into five pieces:

$$cm \quad = \quad rows1 +\!\!+ [row1 +\!\!+ cs\ : row2] +\!\!+ rows2$$

The smallest-choice entry is *cs*. The definition of *expand* is

$$\begin{array}{lcl} expand\ cm & = & [rows1 +\!\!+ [row1 +\!\!+ [c]\ : row2] +\!\!+ rows3 \mid c \leftarrow cs] \\ & & \textbf{where}\ (rows1, row\ : rows2) = break\ (any\ best)\ cm \\ & & \qquad (row1, cs\ : row2) = break\ best\ row \\ & & \qquad best\ cs = (length\ cs = n) \\ & & \qquad n = minchoice\ cm \end{array}$$

The definition of *minchoice* is

$$minchoice \quad = \quad minimum \cdot filter\ (>1) \cdot concat \cdot map\ (map\ length)$$

The number of choices in each cell is computed twice in the above definition, and it may be more efficient to avoid this duplication of effort. Also, we could probably make *minchoice* more efficient since once we have found an entry with two choices there is no point in looking further. Observe that *expand* returns $\perp$ if there is no entry with at least two choices, for then *n* is undefined.

With this definition of *expand* we have

$$mcp \quad \approx \quad concat \cdot map\ mcp \cdot expand$$

Hence

$$\begin{array}{ll} & filter\ correct \cdot mcp \\ \approx & \quad \{\text{above law of } expand\} \\ & filter\ correct \cdot concat \cdot map\ mcp \cdot expand \\ = & \quad \{\text{since } filter\ p \cdot concat = concat \cdot map\ (filter\ p)\} \end{array}$$

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press

$$\begin{array}{ll} & concat \cdot map\,(filter\ correct \cdot mcp) \cdot expand \\ = & \quad \{\text{property of } prune\} \\ & concat \cdot map\,(filter\ correct \cdot mcp \cdot prune) \cdot expand \end{array}$$

Writing $search = filter\ correct \cdot mcp$ we therefore have

$$search \quad = \quad concat \cdot map\,(search \cdot prune) \cdot expand$$

This equation can be used as the basis of a definition of *search* provided we trap the terminal cases:

$$\begin{array}{l} search :: Matrix\ Choices \to [Matrix\ Choices] \\ search\ cm \\ \quad \mid \ blocked\ cm \quad = \quad [\,] \\ \quad \mid \ all\,(all\ single)\,cm \quad = \quad [cm] \\ \quad \mid \ otherwise \quad = \quad (concat \cdot map(search \cdot prune) \cdot expand)\,cm \end{array}$$

4.3 Final version

Now we can write down our final definition of *sudoku*:

$$\begin{array}{lll} sudoku & :: & Board \to [Board] \\ sudoku & = & map\,(map\ head) \cdot search \cdot prune \cdot choices \end{array}$$

The function $map\,(map\ head)$ converts a matrix of singleton choices into a board. The program is quite fast, rarely taking more than a second or two to solve a puzzle.

5 Conclusions

The Sudoku problem provides an ideal classroom example with which to illustrate manipulations of arrays as well as manipulation of programs. Indeed, the pearl is more or less a straightforward transcription of two lectures I gave to first-year undergraduates, omitting most of the calculations. There is a temptation to identify array elements by their cartesian coordinates, and to think of the rows, columns and boxes of an array in terms of operations on these coordinates. Instead, and this is the pedagogic value of the exercise, we have gone for *wholemeal* programming, identifying these structures as complete entities in themselves. There are other Sudoku solvers out there, but the present one certainly seems one of the clearest and simplest.

https://doi.org/10.1017/S0956796806006058 Published online by Cambridge University Press