Chapter 5

A simple Sudoku solver

HOW TO PLAY: Fill in the grid so that every row, every column and every 3×3 box contains the digits 1–9. There's no maths involved. You solve the puzzle with reasoning and logic.
Advice on how to play Sudoku, the Independent

This chapter is devoted to an extended exercise in the use of lists to solve problems, and in the use of equational reasoning to reason about them and to improve efficiency.

The game of Sudoku is played on a 9 by 9 grid, though other sizes are also possible. Given a matrix, such as that in Figure 5.1, the idea is to fill in the empty cells with the digits 1 to 9 so that each row, column and 3×3 box contains the numbers 1 to 9. In general there may be any number of solutions, though in a good Sudoku puzzle there should always be a unique solution. Our aim is to construct a program to solve Sudoku puzzles. Specifically, we will define a function `solve` for computing a list of all the ways a given grid may be completed. If only one solution is wanted, then we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

We begin with a specification, then use equational reasoning to calculate a more efficient version. There's no maths involved, just reasoning and logic!

5.1 Specification

Here are the basic data types of interest, starting with matrices:

```
type Matrix a = [Row a]
```


https://doi.org/10.1017/CBO9781316092415.006

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

Figure 5.1 A Sudoku grid

```
type Row a    = [a]
```

The two type synonyms say nothing more than that `Matrix a` is a synonym for `[[a]]`. But the way it is said emphasises that a matrix is a list of *rows*; more precisely, a $m \times n$ matrix is a list of m rows in which each row is a list with the same length n. Haskell type synonyms cannot enforce such constraints, though there are languages, called *dependently-typed* languages, that can.

A grid is a 9×9 matrix of digits:

```
type Grid  = Matrix Digit
type Digit = Char
```

The valid digits are 1 to 9 with 0 standing for a blank:

```
digits :: [Char]
digits = ['1' .. '9']

blank :: Digit -> Bool
blank = (== '0')
```

Recall that `Char` is also an instance of the type class `Enum`, so `['1' .. '9']` is a valid expression and does indeed return the list of nonzero digits.

We will suppose for simplicity that the input grid contains only digits and blanks, so we do not have to check for the input being well-formed. But should we also insist that no non-blank digit is repeated in any row, column or box? If there were such repetitions there would be no solution. We postpone this decision until after we see how the rest of the algorithm pans out.

Now for the specification. The aim is to write down the simplest and clearest specification without regard to how efficient the result might be. That's a key difference between functional programming and other forms of program construction: we can always begin with a clear and simple, though possibly extremely inefficient definition of `solve`, and then use the laws of functional programming to massage the computation into one that takes acceptable time and space.

One possibility is first to construct a list of all possible correctly filled grids, a vastly long but still finite list, and then to test the given grid against each of them to identify those whose entries match the given non-blank ones. Certainly that approach takes the idea of an inefficient specification to the extreme. Another reasonable alternative is to start with the given grid and to complete it by filling in every possible choice for the blank entries. The result will be a list of filled grids. Then we can filter this list for those that don't contain duplicates in any row, box or column. This specification is implemented by

```
solve :: Grid -> [Grid]
solve = filter valid . completions
```

where the subsidiary functions have types

```
completions :: Grid -> [Grid]
valid   :: Grid -> Bool
```

Let us work on `completions` first and consider `valid` afterwards. One way of defining `completions` is by a two-step process:

```
completions = expand . choices
```

where

```
choices :: Grid -> Matrix [Digit]
expand  :: Matrix [Digit] -> [Grid]
```

The function `choices` installs the available digits for each cell:

```
choices  = map (map choice)
choice d = if blank d then digits else [d]
```

If the cell is blank, then all digits are installed as possible choices; otherwise there is only one choice and a singleton is returned. If we want to apply `f` to every element of a matrix, then `map (map f)` is the function to use because, after all, a matrix is just a list of lists.

After applying `choices` we obtain a matrix each of whose entries is a list of digits. What we want to do next is to define `expand` to convert this matrix into a list of

grids by installing all the choices in all possible ways. That seems a little difficult to think about, so let's consider a simpler problem first, namely when instead of a 9×9 matrix we have a list of length 3. Suppose we want to convert

```
[[1,2,3],[2],[1,3]]
```

into the list

```
[[1,2,1],[1,2,3],[2,2,1],[2,2,3],[3,2,1],[3,2,3]]
```

The second list of lists arises by taking, in all possible ways, one element from the first list, one element from the second list and one element from the third list. Let us call the function that does this `cp` (short for 'cartesian product', which is exactly what a mathematician would call it). There doesn't seem to be any clever way of computing `cp` in terms of other functions, so we adopt the default strategy of defining this function by breaking up its argument list into two possibilities, the empty list `[]` and a nonempty list `xs:xss`. You might guess the definition of `cp []` but you would probably be wrong; the better alternative is to think about the second case first. Suppose we assume

```
cp [[2],[1,3]] = [[2,1],[2,3]]
```

How can we extend this definition to one for `cp ([1,2,3]:[[2],[1,3]])`? The answer is to prefix 1 to every element of `cp [[2],[1,3]]`, then to prefix 2 to every element of the same list, and finally to prefix 3 to every element. That process can be expressed neatly using a list comprehension:

```
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

In words, prefix every element of `xs` to every element of `cp xss` in all possible ways.

If your nose is good at sniffing out inefficiencies, you might suspect that this one-liner is not the best possible, and you would be right. We will return to this point in Section 7.3, but let's just say that a more efficient definition is

```
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

This version guarantees that `cp xss` is computed just once.

Now, what is `cp []`? The answer is not `[]` but `[[]]`. To see why the first is wrong, consider a little calculation:

```
cp [xs] = cp (xs:[])
        = [x:ys | x <- xs, ys <- cp []]
```

```
        = [x:ys | x <- xs, ys <- []]
        = []
```

In fact with `cp [] = []` we can show that `cp xss = []` for all lists `xss`. So that definition is clearly wrong. You can check that the second alternative, `[[]]`, does give what is wanted.

Summarising, we can define `cp` by

```
cp :: [[a]] -> [[a]]
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

For example,

```
ghci> cp [[1],[2],[3]]
[[1,2,3]]

ghci> cp [[1,2],[],[4,5]]
[]
```

In the second example there is no possible choice from the middle list, so the empty list is returned.

But what about matrices and `expand`, which does the same thing on matrices as `cp` does on lists? You will have to think a bit before seeing that what is wanted is

```
expand :: Matrix [Digit] -> [Grid]
expand  = cp . map cp
```

That looks a little cryptic, but `map cp` returns a list of all possible choices for each row, and so applying `cp` to the result installs each choice for the rows in all possible ways. The general type of the right-hand side is

```
cp . map cp :: [[[a]]] -> [[[a]]]
```

and the declared type of `expand` is just a restricted version of this type. Note that `expand` returns the empty list if any element in any row is the empty list.

Finally, a valid grid is one in which no row, column or box contains duplicates:

```
valid :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols g) &&
          all nodups (boxs g)
```

The prelude function `all` is defined by

```
all p = and . map p
```

Applied to a finite list `xs` the function `all p` returns `True` if all elements of `xs` satisfy `p`, and `False` otherwise. The function `nodups` can be defined by

```
nodups :: (Eq a) => [a] -> Bool
nodups []     = True
nodups (x:xs) = all (/=x) xs && nodups xs
```

Evaluation of `nodups` on a list of length n takes time proportional to n^2. As an alternative we could sort the list and check that it is strictly increasing. Sorting can be done in time proportional to $n \log n$ steps. That seems a big saving over n^2. However, with $n = 9$, it is not clear that using an efficient sorting algorithm is worthwhile. What would you prefer: $2n^2$ steps or $100n \log_2 n$ steps?

It remains to define `rows`, `cols` and `boxs`. If a matrix is given by a list of its rows, then `rows` is just the identity function on matrices:

```
rows :: Matrix a -> Matrix a
rows = id
```

The function `cols` computes the transpose of a matrix. Thus if a matrix consists of m rows, where each row has length n, the transpose is a list of n rows, where each row has length m. Assuming both m and n are not zero, we can define

```
cols :: Matrix a -> Matrix a
cols [xs]     = [[x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

It is usual in matrix algebra to suppose that the matrix is nonempty, and that certainly suffices here, but it is interesting to consider what happens if we allow m or n to be zero. This point is taken up in the exercises.

The function `boxs` is a little more interesting. We give the definition first and explain it afterwards:

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup .
       map cols .
       group . map group
```

The function `group` splits a list into groups of three:

```
group :: [a] -> [[a]]
```

```
group [] = []
group xs = take 3 xs:group (drop 3 xs)
```

The function `ungroup` takes a grouped list and ungroups it:

```
ungroup :: [[a]] -> [a]
ungroup = concat
```

The action of `boxs` in the 4×4 case, when `group` splits a list into groups of two rather than three, is illustrated by the picture

$$\begin{array}{ccc}
\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} & \longrightarrow & \begin{pmatrix} \begin{pmatrix} ab & cd \\ ef & gh \end{pmatrix} \\ \begin{pmatrix} ij & kl \\ mn & op \end{pmatrix} \end{pmatrix} \\
 & & \downarrow \\
\begin{pmatrix} a & b & e & f \\ c & d & g & h \\ i & j & m & n \\ k & l & o & p \end{pmatrix} & \longleftarrow & \begin{pmatrix} \begin{pmatrix} ab & ef \\ cd & gh \end{pmatrix} \\ \begin{pmatrix} ij & mn \\ kl & op \end{pmatrix} \end{pmatrix}
\end{array}$$

Grouping produces a list of matrices; transposing each matrix and ungrouping yields the boxes, as a matrix whose rows are the boxes of the original matrix.

5.2 Lawful program construction

Observe that instead of thinking about matrices in terms of indices, and doing arithmetic on indices to identify the rows, columns and boxes, we have gone for definitions of these functions that treat the matrix as a complete entity in itself. This style has aptly been called *wholemeal programming*. Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction.

For example, here are three laws that are valid on Sudoku grids:

```
rows . rows = id
cols . cols = id
boxs . boxs = id
```

In other words, all three functions are involutions. The first two are valid on all matrices, and the third is valid on arbitrary $n^2 \times n^2$ matrices (provided we change

the definition of `group` to group by n). Two are easy to prove, but one is more difficult. The difficult law is not the one about `boxs`, as you might expect, but the involution property of `cols`. Though it is intuitively obvious that transposing a matrix twice gets you back to the original matrix, proving it from the definition of `cols` is a little tricky and we won't go into details, basically because we haven't yet discussed the tools available to do the job.

By contrast, here is the proof of the involution property of `boxs`. The proof is by simple equational reasoning. It makes use of various laws, including the functor laws of `map`, the fact that `id` is the identity element of composition, and the facts that

```
ungroup . group = id
group . ungroup = id
```

The second equation is valid only on grouped lists, but that will be the case in the calculation to come.

We will talk through the proof rather than lay everything out in a long chain. The starting point is to use the definition of `boxs` to rewrite `boxs . boxs`:

```
map ungroup . ungroup . map cols . group . map group .
map ungroup . ungroup . map cols . group . map group
```

The middle expression `map group . map ungroup` simplifies to `id` using the functor law of `map` and the property that `group` and `ungroup` are inverses. That gives

```
map ungroup . ungroup . map cols . group .
ungroup . map cols . group . map group
```

An appeal to `group . ungroup = id` gets us to

```
map ungroup . ungroup . map cols .
map cols . group . map group
```

The functor law of `map` and the involution property of `cols` now gets us to

```
map ungroup . ungroup . group . map group
```

And the proof is finished off using `ungroup . group = id` twice more. As you can see, it's a very simple calculation.

Here are three more laws, valid on $N^2 \times N^2$ matrices of choices:

```
map rows . expand = expand . rows
map cols . expand = expand . cols
```

```
map boxs . expand = expand . boxs
```

We will make use of these laws in a short while.

Finally, here are two laws about cp:

```
map (map f) . cp    = cp . map (map f)
filter (all p) . cp = cp . map (filter p)
```

The first law, a naturality law, is suggested solely by the type of cp; we saw similar laws in the previous chapter. The second law says that as an alternative to taking the cartesian product of a list of lists, and then retaining only those lists all of whose elements satisfy p, we can first filter the original lists to retain only those elements that satisfy p and then take the cartesian product. As the previous sentence illustrates, one equation can be worth a thousand words.

5.3 Pruning the matrix of choices

Summarising what we have at the moment,

```
solve :: Grid -> [Grid]
solve = filter valid . expand . choices
```

Though executable in theory, this definition of solve is hopeless in practice. Assuming about 20 of the 81 entries are fixed initially, there are about 9^{61}, or

```
ghci> 9^61
1617309269922988089371861846558644535758328064784065995760 9
```

grids to check! We therefore need a better approach.

To make a more efficient solver, an obvious idea is to remove any choices from a cell c that already occur as singleton entries in the row, column and box containing c. A singleton entry corresponds to a fixed choice. We therefore seek a function

```
prune :: Matrix [Digit] -> Matrix [Digit]
```

so that

```
filter valid . expand = filter valid . expand . prune
```

How can we define prune? Well, since a matrix is a list of rows, a good place to start is by pruning a single row. The function pruneRow is defined by

```
pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
```

```
        where fixed = [d | [d] <- row]
```

The fixed choices are the singleton entries in each row. The definition of `fixed` uses a list comprehension involving a pattern: all elements of `row` that are not singletons are discarded.

The function `remove` removes the fixed choices from any choice that is not fixed:

```
remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]
remove ds xs  = filter (`notElem` ds) xs
```

The standard prelude function `notElem` is defined by

```
notElem :: (Eq a) => a -> [a] -> Bool
notElem x xs = all (/= x) xs
```

Here are a couple of examples of the use of `pruneRow`:

```
ghci> pruneRow [[6],[1,2],[3],[1,3,4],[5,6]]
[[6],[1,2],[3],[1,4],[5]]

ghci> pruneRow [[6],[3,6],[3],[1,3,4],[4]]
[[6],[],[3],[1],[4]]
```

In the first example, `[6]` and `[3]` are the fixed choices; removing these choices from the other entries reduces the last entry to a fixed choice. In the second example, removing the fixed choices reduces the second entry to the empty list of choices.

The function `pruneRow` satisfies the equation

```
filter nodups . cp = filter nodups . cp . pruneRow
```

In words, this equation says that pruning a row will not throw away any list that contains no duplicates. We will also make use of this law in a short while.

We are now nearly ready for a calculation that will determine the function `prune`. Nearly, but not quite because we are going to need two more laws: If `f . f = id`, then

```
filter (p . f) = map f . filter p . map f
filter (p . f) . map f = map f . filter p
```

The second law follows from the first (Why?). Here is the proof of the first law:

```
  map f . filter p . map f
=   {we proved in the previous chapter that
          filter p . map f = map f . filter (p . f)}
  map f . map f . filter (p . f)
=   {functor law of map and f . f = id}
  filter (p . f)
```

Now for the main calculation. The starting point is to use the definition of `valid` to rewrite the expression `filter valid . expand` in the form

```
filter valid . expand
 = filter (all nodups . boxs) .
   filter (all nodups . cols) .
   filter (all nodups . rows) . expand
```

The order in which the filters appear on the right is not important. The plan of attack is to send each of these filters into battle with `expand`. For example, in the `boxs` case we can calculate:

```
  filter (all nodups . boxs) . expand
=   {above law of filter, since boxs . boxs = id}
  map boxs . filter (all nodups) . map boxs . expand
=   {since map boxs . expand = expand . boxs}
  map boxs . filter (all nodups) . expand . boxs
=   {definition of expand}
  map boxs . filter (all nodups) . cp . map cp . boxs
=   {since filter (all p) . cp = cp . map (filter p)}
  map boxs . cp . map (filter nodups) . map cp . boxs
=   {functor law of map}
  map boxs . cp . map (filter nodups . cp) . boxs
```

Now we use the property

```
filter nodups . cp = filter nodups . cp . pruneRow
```

to rewrite the final expression in the form

```
map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
```

The remaining steps essentially repeat the calculation above, but in the reverse direction:

```
  map boxs . cp . map (filter nodups . cp . pruneRow) .
  boxs
=   {functor law of map}
  map boxs . cp . map (filter nodups) .
  map (cp . pruneRow) . boxs
=   {since cp . map (filter p) = filter (all p) . cp}
  map boxs . filter (all nodups) . cp .
  map (cp . pruneRow) . boxs
=   {functor law of map}
  map boxs . filter (all nodups) .
  cp . map cp . map pruneRow . boxs
=   {definition of expand}
  map boxs . filter (all nodups) .
  expand . map pruneRow . boxs
=   {law of filter since boxs . boxs = id}
  filter (all nodups . boxs) . map boxs .
  expand . map pruneRow . boxs
=   {since map boxs . expand = expand . boxs}
  filter (all nodups . boxs) . expand .
  boxs . map pruneRow . boxs
=   {introducing pruneBy f = f . pruneRow . f}
  filter (all nodups . boxs) . expand . pruneBy boxs
```

We have shown that

```
filter (all nodups . boxs) . expand
 = filter (all nodups . boxs) . expand . pruneBy boxs
```

where `pruneBy f = f . map pruneRow . f`. Repeating the same calculation for rows and columns, we obtain

```
filter valid . expand = filter valid . expand . prune
```

where

```
prune = pruneBy boxs . pruneBy cols . pruneBy rows
```

In conclusion, the previous definition of `solve` can now be replaced with a new one:

```
solve = filter valid . expand . prune . choices
```

In fact, rather than have just one `prune` we can have as many prunes as we like. This is sensible because after one round of pruning some choices may be resolved into singleton choices and another round of pruning may remove still more impossible choices.

So, let us define

```
many :: (Eq a) => (a -> a) -> a -> a
many f x = if x == y then x else many f y
           where y = f x
```

and redefine `solve` once again to read

```
solve = filter valid . expand . many prune . choices
```

The simplest Sudoku problems are solved just by repeatedly pruning the matrix of choices until only singleton choices are left.

5.4 Expanding a single cell

The result of `many prune . choices` is a matrix of choices that can be put into one of three classes:

1. A *complete* matrix in which every entry is a singleton choice. In this case `expand` will extract a single grid that can be checked for validity.
2. A matrix that contains the empty choice somewhere. In this case `expand` will produce the empty list.
3. A matrix that does not contain the empty choice but does contain some entry with two or more choices.

The problem is what to do in the third case. Rather than carry out full expansion, a more sensible idea is to make use of a partial expansion that installs the choices for just one of the entries, and to start the pruning process again on each result. The hope is that mixing pruning with single-cell expansions can lead to a solution more quickly. Our aim therefore is to construct a partial function

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
```

that expands the choices for one cell only. This function will return well-defined results only for incomplete matrices, and on such matrices is required to satisfy

```
expand = concat . map expand .  expand1
```

Actually this equality between two lists is too strong. We want to ensure that no possible choice is lost by partial expansion, but do not really care about the precise order in which the two sides deliver their results. So we will interpret the equation as asserting the equality of the two sides up to some permutation of the answers.

Which cell should we perform expansion on? The simplest answer is to find the first cell in the matrix with a non-singleton entry. Think of a matrix `rows` broken up as follows:

```
rows = rows1 ++ [row] ++ rows2
row = row1 ++ [cs] ++ row2
```

The cell `cs` is a non-singleton list of choices in the middle of `row`, which in turn is in the middle of the matrix `rows`.

Then we can define

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
```

To break up the matrix in this way, we use the prelude function `break`:

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break p = span (not . p)
```

The function `span` was defined in Section 4.8. For example,

```
ghci> break even [1,3,7,6,2,3,5]
([1,3,7],[6,2,3,5])
```

We also need the standard prelude function `any`, defined by

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

where `or` takes a list of booleans and returns `True` if any element is `True`, and `False` otherwise:

```
or :: [Bool] -> Bool
or []     = False
or (x:xs) = x || or xs
```

Finally, the `single` test is defined (using don't care patterns) by

```
single :: [a] -> Bool
single [_] = True
single _   = False
```

Now we can define

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
   where
   (rows1,row:rows2) = break (any (not . single)) rows
   (row1,cs:row2)    = break (not . single) row
```

The first `where` clause breaks a matrix into two lists of rows with the row at the head of the second list being one that contains a non-singleton choice. A second appeal to `break` then breaks this row into two lists, with the head of the second list being the first non-singleton element. If the matrix contains only singleton entries, then

```
break (any (not . single)) rows = [rows,[]]
```

and execution of `expand1` returns an error message.

The problem with this definition of `expand1` is that it can lead to wasted work. If the first non-singleton entry found in this way happens to be the empty list, then `expand1` will return the empty list, but if such a list is buried deep in the matrix, then `expand1` will do a lot of useless calculation trying to find a solution that isn't there. It is arguable that a better choice of cell on which to perform expansion is one with the *smallest* number of choices (not equal to 1 of course). A cell with no choices means that the puzzle is unsolvable, so identifying such a cell quickly is a good idea.

The change to `expand1` to implement this idea is as follows:

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
expand1 rows
 = [rows1 ++ [row1 ++ [c]:row2] ++ rows2 | c <- cs]
   where
   (rows1,row:rows2) = break (any smallest) rows
   (row1,cs:row2)    = break smallest row
   smallest cs       = length cs == n
   n                 = minimum (counts rows)
```

The function `counts` is defined by

```
counts = filter (/= 1) . map length . concat
```

The value `n` is the smallest number of choices, not equal to 1, in any cell of the matrix of choices. We will leave the definition of `minimum` as an exercise. The value of `n` will be 0 if the matrix has an empty choice entry anywhere, and in this case `expand1` will return the empty list. On the other hand, if the matrix of choices contains only singleton choices, then `n` is the minimum of the empty list, which is the undefined value $\perp$. In this case `expand1` will also return $\perp$, so we had better ensure that `expand1` is applied only to incomplete matrices. A matrix is incomplete if it does not satisfy `complete`:

```
complete :: Matrix [Digit] -> Bool
complete = all (all single)
```

We can also usefully generalise `valid` to a test on matrices of choices. Suppose we define `safe` by

```
safe :: Matrix [Digit] -> Bool
safe m = all ok (rows cm) &&
         all ok (cols cm) &&
         all ok (boxs cm)
ok row = nodups [x | [x] <- row]
```

A matrix is safe if none of the singleton choices in any row, column or box contain duplicates. But a safe matrix may contain non-singleton choices. Pruning can turn a safe matrix into an unsafe one, but if a matrix is safe after pruning it has to be safe beforehand. In symbols, `safe . prune = safe`. A complete and safe matrix yields a solution to the Sudoku problem, and this solution can be extracted by a simplified version of `expand`:

```
extract :: Matrix [Digit] -> Grid
extract = map (map head)
```

Hence on a safe and complete matrix `m` we have

```
filter valid (expand m) = [extract m]
```

On a safe but incomplete matrix we have

```
  filter valid . expand
= filter valid . concat . map expand . expand1
```

up to permutation of each side. Since

```
filter p . concat = concat . map (filter p)
```

we obtain that `filter valid . expand` simplifies to

```
concat . map (filter p . expand) . expand1
```

And now we can insert a single `prune` to obtain

```
concat . map (filter p . expand . prune) . expand1
```

Hence, introducing

```
search = filter valid . expand . prune
```

we have, on safe but incomplete matrices, that

```
search = concat . map search . expand1 . prune
```

And now we can replace `solve` by a third version:

```
solve = search . choices
search cm
  | not (safe pm) = []
  | complete pm   = [extract pm]
  | otherwise     = concat (map search (expand1 pm))
  where pm = prune cm
```

This is our final simple Sudoku solver. We could replace `prune` in the last line by `many prune`. Sometimes many prunes work faster than one prune; sometimes not. Note that the very first safety test occurs immediately after one round of pruning on the installed choices; consequently flawed input is detected quickly.

5.5 Exercises

Exercise A

How would you add 1 to every element in a given matrix of integers? How would you sum the elements of a matrix? The function `zipWith (+)` adds two rows, but what function would add two matrices? How would you define matrix multiplication?

Exercise B

What are the dimensions of the matrix `[[],[]]`? Of the matrix `[]`?

The function `cols` (here renamed as `transpose`) was defined by

```
transpose :: [[a]] -> [[a]]
transpose [xs]     = [[x] | x <- xs]
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

Fill in the dots that would enable you to replace the first clause by

```
transpose []       = ...
```

The above definition of `transpose` proceeds row by row. Here is part of a definition that proceeds column by column:

```
transpose xss = map head xss:transpose (map tail xss)
```

Complete this definition.

Exercise C

Which of the following equations are true (no justification is necessary):

```
any p = not . all (not p)
any null = null . cp
```

Exercise D

Given a function `sort :: (Ord a) => [a] -> [a]` that sorts a list, construct a definition of

```
nodups :: (Ord a) => [a] -> Bool
```

Exercise E

The function `nub :: (Eq a) => [a] -> [a]` removes duplicates from a list (a version of this function is available in the library `Data.List`). Define `nub`. Assuming the order of the elements in the result is not important, define

```
nub :: (Ord a) => [a] -> [a]
```

so that the result is a more efficient function.

Exercise F

The functions `takeWhile` and `dropWhile` satisfy

```
span p xs = (takeWhile p xs,dropWhile p xs)
```

Give direct recursive definitions of `takeWhile` and `dropWhile`.

Assuming `whiteSpace :: Char -> Bool` is a test for whether a character is

white space (such as a space, a tab or a newline) or a visible character, construct a definition of

```
words :: String -> [Word]
```

that breaks a string up into a list of words.

Exercise G

Define `minimum :: Ord a => [a] -> a`.

Exercise H

Why didn't we define `solve` by the following?

```
solve = search . choices
search m
 | not (safe m) = []
 | complete m   = [extract m]
 | otherwise    = process m
 where process = concat . map search . expand1 . prune
```

5.6 Answers

Answer to Exercise A

Adding 1 to every matrix element is defined by `map (map (+1))`.

Summing a matrix is defined by `sum . map sum`, where `sum` sums a list of numbers. Alternatively, we could use `sum . concat`.

Matrix addition is defined by `zipWith (zipWith (+))`.

For matrix multiplication we first define

```
scalarMult :: Num a => [a] -> [a] -> a
scalarMult xs ys = sum (zipwith (*) xs ys)
```

Then we have

```
matMult :: Num a => Matrix a -> Matrix a -> Matrix a
matMult ma mb = [map (scalarMult row) mbt | row <- ma]
                where mbt = transpose mb
```

Answer to Exercise B

The matrix `[[],[]]` has dimensions 2×0. The matrix `[]` has dimensions $0 \times n$ for every n. The transpose of such a matrix therefore has to have dimensions $n \times 0$ for every n. The only reasonable possibility is to let n be infinite:

```
transpose :: [[a]] -> [[a]]
transpose []       = repeat []
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

where `repeat x` gives an infinite list of repetitions of `x`. Note that

```
transpose [xs] = zipWith (:) xs (repeat [])
               = [[x] | x <- xs]
```

The alternative definition is

```
transpose ([]:xss) = []
transpose xss = map head xss:transpose (map tail xss)
```

The assumption in the first line is that if the first row is empty, then all the rows are empty and the transpose is the empty matrix.

Answer to Exercise C

Both the equations are true.

Answer to Exercise D

```
nodups :: (Ord a) => [a] -> Bool
nodups xs = and (zipWith (/=) ys (tail ys))
            where ys = sort xs
```

Answer to Exercise E

```
nub :: (Eq a) => [a] -> [a]
nub []     = []
nub (x:xs) = x:nub (filter (/= x) xs)

nub :: (Ord a) => [a] -> [a]
nub = remdups . sort

remdups []     = []
remdups (x:xs) = x:remdups (dropWhile (==x) xs)
```

The function `dropWhile` is defined in the next exercise.

Answer to Exercise F

```
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
          = if p x then x:takeWhile p xs else []
dropWhile p [] = []
dropWhile p (x:xs)
          = if p x then dropWhile p xs else x:xs
```

The definition of `words` is

```
words :: String -> [Word]
words xs | null ys   = []
         | otherwise = w:words zs
         where ys = dropWhile whiteSpace xs
               (w,zs) = break whiteSpace ys
```

Answer to Exercise G

```
minimum :: Ord a => [a] -> a
minimum [x]    = x
minimum (x:xs) = x `min` minimum xs
```

Note that the minimum of the empty list is undefined.

Answer to Exercise H

The suggested definition of `solve` would return the undefined value if the matrix becomes complete after one round of pruning.

5.7 Chapter notes

The *Independent* newspaper no longer uses the rubric for Sudoku quoted at the start of the chapter. The presentation follows that in my book *Pearls of Functional Algorithm Design* (Cambridge, 2010). The site

```
haskell.org/haskellwiki/Sudoku
```

contains about 20 Haskell implementations of Sudoku, many of which use arrays and/or monads. We will meet arrays and monads in Chapter 10.