Combinatorics

Functional Programming SS2007

Stefan Blom

Date: 2007/06/19 08:53:47
Revision: 1.3

1 Combining elements of a given list.

In this paragraph, we go through all the steps necessary to define the the list of all possible permutations perm l of a list l. The only argument of perm is a list, so it is natural to use recursion over a list:

```
let rec perm l = match l with
  | [] -> ...
  | x::xs -> ...
;;
```

The only possible permutation of an empty list is the empty list so the first case is easy. The second case is not obvious, so we just put the recursive call in a let and try to make sense of the result:

```
let rec perm l = match l with
  | [] -> [[]]
  | x::xs -> let ps = perm xs in ...
;;
```

Question: what are the possible permutations of x::xs, given the possible permutations of xs? Answer: given a permutation of xs, we can create a permutation of x::xs by inserting x anywhere in xs. So we must define a function perm1 that given an element and a list inserts the element anywhere in the list. This is obviously a recursive function,so we start with

```
let rec perm1 x l = match l with
  | [] -> ...
  | y::ys -> ...
;;
```

There is only one way of inserting x in the empty list. Inserting x in y::ys can be done before the whole list or somewhere in ys. This can be written as

```
let rec perm1 x l = match l with
  | [] -> [[x]]
  | y::ys -> (x::y::ys)::(map (fun zs -> y::zs) (perm1 x ys))
;;
```

Going back to our original problem, we can insert the x anywhere in a permutation of xs by writing map (perm1 x) ps the result is a list of lists of possible permutations, so we need to concatenate that list:

```
let rec perm l = match l with
 | [] -> [[]]
 | x::xs ->
     let ps = perm xs in
       let xps = map (perm1 x) ps in
         List.concat xps
;;
```

The structure of this definition is very common for function that return combinations of lists. Often there is precisely one possibility for the empty list. Given a non-empty list $x :: xs$, we can (i) recursively build a list of combinations for xs; (ii) for each combination combination c for xs we build the possible combinations for x and c; (iii) concatenate the list of lists of combinations built in step (ii) as the final result.

Exercise 1 Define the function sub_lists that takes a list and returns a list of all possible sub lists of the given list. (A sub list of a list is obtained by erasing zero or more elements of the list.) So for example

$$\text{sub\_lists}\quad [1;2;3]\ = [[]; [3]; [2]; [2;3]; [1]; [1;3]; [1;2]; [1;2;3]]$$

Note that the order of the returned list is irrelevant so

$$\text{sub\_lists}\quad [1;2;3]\ = [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]]$$

is just as good. Hint: start with

```
sub_lists l = match l with
 | [] -> ...
 | x::xs -> let ls = sub_lists xs in ...
```

and answer the questions

- What are the sub lists of the empty list?
- Given a sub list sl of xs, what are the possible sub lists of x::xs that we can construct from x and sl?
- How do we write that in OCaml?

Exercise 2 Define the Cartesian product. That is define a function cartesian n l, which given a list representation of a set S return the list representation of S^n, where each element of S^n is represented as a list. E.g.

$$\text{cartesian}\ \ 3\ \ [0;1]\ = [[0;0;0]; [0;0;1]; [0;1;0]; [0;1;1]; [1;0;0]; [1;0;1]; [1;1;0]; [1;1;1]]$$

2 The n-queens problem

The n-queens problem is the problem of placing n queens on a $n \times n$ chess board in such a way that no two queens can take each other. That is, each row, column and diagonal may contain at most one queen. For example in 4×4, we can have

	Q		
			Q
Q			
		Q	

Table 1: Solution to the n-queens problem (queens.ml).

```
let rec seq n k = if n<=k then n::(seq (n+1) k) else [];;

let rec pairs l1 l2 = match l1 with
 | [] -> []
 | x::xs -> (List.map (fun y -> (x,y)) l2)@(pairs xs l2)
;;

let rec choose n l = if n = 0 then [[]] else
  match l with
    | [] -> []
    | x::xs -> (List.map (fun ys->x::ys) (choose (n-1) xs))
               @
               (choose n xs)
;;

let good_pair [(i1,j1);(i2,j2)] =
  (i1<>i2) & (j1<>j2) & (i1-i2<>j1-j2) & (i1-i2<>j2-j1) ;;

let all l = List.fold_left ( & ) true l ;;

let good_list l = all (List.map good_pair (choose 2 l)) ;;

let queens n = let positions = pairs (seq 1 n) (seq 1 n) in
   List.filter good_list (choose n positions);;
```

We represent (possible) solutions by lists of positions on the board in matrix notation. Thus, the given solution corresponds to

$$[(1,2);(2,4);(3,1);(4,3)]$$

In Table 1, we have given a complete solution to the n-queens problem. The solution is based on

1. Generating the list of all positions on the board.
2. Generating the list of all lists of n positions.
3. Selecting the solutions from that list.

A solution is a list of position, where each pair of positions on the list is good. Meaning that each pair is neither in the same column nor in the same row nor on the same diagonal.

The given solution suffers from two performance problems. First, building the initial list of solution by selecting n positions randomly generates too many candidates. It would be better to start by choosing a random position on each row. That is, to generate only those candidates where we have precisely one queen in each row. As a further improvement it is also possible to start from those solutions with precisely one queen in each row and column. (Hint: consider permutations of $[1;\cdots;n]$.) Second, the solution uses the default list operations. It would be better to use either lazy or tail recursive list operations.

Exercise 3 Rewrite the solution in Table 1 until you obtain a version, for which the interpreter is able to generate all solutions to the 8-queens problem.

3 Sudoku

An easy way to represent a Sudoku puzzle is to represent the empty fields with the list $[1;\cdots;9]$ and a given field i with $[i]$ as follows: (see `sudoku.ml`)

```
let any = [1;2;3;4;5;6;7;8;9];;

let sample1 = [
[any;any;[3];   [9];any;any;   [7];[6];any];
[any;[4];any;   any;any;[6];   any;any;[9]];
[[6];any;[7];   any;[1];any;   any;any;[4]];

[[2];any;any;   [6];[7];any;   any;[9];any];
[any;any;[4];   [3];any;[5];   [6];any;any];
[any;[1];any;   any;[4];[9];   any;any;[7]];

[[7];any;any;   any;[9];any;   [2];any;[1]];
[[3];any;any;   [2];any;any;   any;[4];any];
[any;[2];[9];   any;any;[8];   [5];any;any]]
;;
```

This representation was chosen because it can be used as the data structure in the solving process as well: every field of the matrix contains the list of possible values.

We will now consider a few components from which a full Sudoku solver can be built.

a) Given a row (a list of lists of possible values), we can:

1. Generate the list of all possible permuations of $[1;\cdots;9]$.
2. Select those permutations that are possible given the possible values for each position.
3. From this list of possible permutation go back to a list of possibilities.

For example, if we consider the smaller example $[[1;2];[2;3];[2;3]]$ then we get

$$[[1;2;3];[1;3;2];[2;1;3];[2;3;1];[3;1;2];[3;2;1]]$$

as the list of permutations. If we check against the first field ($[1;2]$) we are left with

$$[[1;2;3];[1;3;2];[2;1;3];[2;3;1]]$$

Checking the second field reduces it to

$$[[1;2;3];[1;3;2];[2;3;1]]$$

And checking against the last field yields

$$[[1;2;3];[1;3;2]]$$

If we generate possible values from this we get

$$[[1];[2;3];[2;3]]$$

Note that you should treat the lists of possible values as sets: at most one occurrence of each value.

b) We do not only need to solve for rows, we also must solve for columns and sub-matrices. To be able to use the row-sovler for that, we introduce the following operations on matrices:

transpose

$$\text{trans}\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{pmatrix} = \begin{matrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{matrix}$$

block transpose

$$\text{block}\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{pmatrix} = \begin{matrix} a_1 & a_2 & b_1 & b_2 \\ a_3 & a_4 & b_3 & b_4 \\ c_1 & c_2 & d_1 & d_2 \\ c_3 & c_4 & d_3 & d_4 \end{matrix}$$

Note that

$$\text{trans}(\text{trans}(M)) = M \text{ and } \text{block}(\text{block}(M)) = M$$

That means that you can (block) transpose, solve rows and (block) transpose again to do column (block) solving.

c) This solving process has to be repeated until there is no more change. For this the function fix is useful:

```
let rec fix f x =
  let y = f x in
  if x = y then x else fix f y
;;
```

For example, if we have a function that decreases until we hit a minimum then we alwas get the minimum:

```
# fix (fun x -> max 2 (x-1)) 6;;
- : int = 2
```

d) The last element is that when we get stuck solving, we find a field for which more than one element is possible and simply try each possibility. Note that if you choose wrong then you might end up with an unsolvable puzzle which results in one or more of the possible lists being empty.

Exercise 4 Implement a Sudoku solver.