Variations around the Sudoku Puzzle

The goal of this project is to specify, implement, and prove algorithms for solving the famous Sudoku puzzle (`http://en.wikipedia.org/wiki/Sudoku`).

You are expected to achieve this project using the Why3 tool version 0.80, using automated provers (typically Alt-Ergo, CVC3, Z3) and, if you like, the Coq proof assistant. The project must be done individually (no team work allowed). You must send by e-mail to `Claude.Marche@inria.fr`, `Guillaume.Melquiond@inria.fr`, no later than **Monday March 4, 2013 at 12:00 UTC**, an archive (`zip` or `tar.gz`) containing:

- A *`name`*`.mlw` source file containing your specifications and implementations, where you replace *name* with your family name.
- The directory *name* generated by Why3, containing the proof session file `why3session.mlw` and Coq proof scripts `.v` if any.
- A document *`name`*`.pdf` in PDF format reporting on your work, where you should detail your approach to the problem, the choices you made to design the specifications, the implementations, etc.

A note about the structure and the evaluation of the project: Section 2 asks to implement and prove a naive version of a Sudoku solver, which is not efficient but supposed to be simple to prove correct. In the other sections, several improvements over the naive version are then proposed. The evaluation will consider the quality of the work in priority, rather than quantity. In other words, we expect you to send us a solution that is fully proved correct, even if it does not implement all the extensions but only some of them.

Notice also that the quality of the PDF report counts at least for half of the final evaluation. You can add in the conclusion some discussions about the extensions you were not able to implement or prove, if any, and what were the problems.

1 Description of the Sudoku Puzzle

The puzzle itself is fully described on the page `http://en.wikipedia.org/wiki/Sudoku`. In short, a Sudoku problem is given by a 9×9 grid, where a few cells are filled with numbers between 1 and 9, such as:

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

The goal is to fill the remaining cells with numbers between 1 and 9, such that in each column, each row, and each of the nine 3×3 sub-squares, there is exactly one occurrence of each number between 1 and 9. For example, the following is a solution of the problem above.

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

To help you to start, we give an initial file `grid.mlw` containing a few definitions. The type `grid` is defined as a map of integers indexed by integers. We propose a representation with a one-dimensional array only, with indexes of cells ranging from 0 to 80. If a cell does not contain a value between 1 and 9, it is assumed to be empty.

Constant arrays formalize the notion of columns, rows, and squares: for every cell, array `column_start` gives the numbers of the first cell in the same column, and array `column_offsets` gives the set of offsets one needs to obtain the other cells of the same column. In other words, the set of cells of the column containing cell i is

$$\{\texttt{column\_start}[i] + \texttt{column\_offsets}[0], \ldots, \texttt{column\_start}[i] + \texttt{column\_offsets}[8]\}$$

Others arrays play a similar role for rows and squares.

2 Naive Solver

A very naive algorithm for solving a grid is the brute-force search: filling the empty cells with all the possibilities from 1 to 9, and checking if the resulting grid is a solution. Such an algorithm performs 9^n verifications where n is the number of empty cells.

To turn this idea into a still naive but more efficient algorithm, one can try to fill the empty cells one at a time, and check if the constraints are satisfied before continuing to fill the rest of the grid. Let us make this idea more precise.

First, let us call *chunk* a part of a grid that is either a column, a row, or a 3×3 square. We say that a chunk in a partially filled grid is *valid* if it contains at most one occurrence of each number from 1 to 9. Empty cells do not count. By extension, a partially filled grid is valid if all of the chunks are valid. Thus, a grid is a solution if it is both full and valid.

The naive algorithm is given on Figure 1.

2.1 Checking validity of grids

The first step is to design a Why3 function that given a grid g, an index i between 0 and 80, and two arrays $start$ and $offsets$, checks whether the chunk defined by $(start, offsets)$ that contains i is valid in grid g. The naive, quadratic algorithm is given on Figure 2.

There are two ways to implement this checker: either by returning a Boolean, or returning just `()` and raising an exception when it is invalid.

```
solve(g, i) =
  if i = 81 then g is a solution else
  if cell i of g is already filled with a number then solve(g, i + 1)
  else for x from 1 to 9
          let g' be g where cell i is filled with x
          if g' is valid then solve(g', i + 1)
       done

solveGrid(g) = solve(g, 0)
```

Figure 1: Naive solver

```
check_chunk_valid(g, i, start, offsets) =
  let s be start[i]
  for o1 from 0 to 8
    for o2 from 0 to 8
      if o1 <> o2 then
        if g[s + offsets[o1]] = g[s + offsets[o2]] then return false
    done
  done
  return true
```

Figure 2: Naive chunk validity checker

1. *Specify, implement, and prove a Why3 function that checks whether a chunk is valid, either returning a Boolean or raising an exception.*

The next step is to design a function that checks the validity of a grid, as given on Figure 3. We propose to check all the chunk of all cells. Note that it is not efficient because it checks each chunk 9 times, but it does not matter for a naive algorithm. One of the extensions will deal with this issue.

As above, you can either return a Boolean or raise an exception when it is invalid.

2. *Specify, implement, and prove a Why3 function that checks whether a grid is valid.*

Finally, we want to implement functions corresponding to the pseudo-code for `solve` and `solveGrid` of Figure 1. For the moment, we are interested only in the soundness of the method, not its completeness. We want to specify that whenever it returns a solution, then it is indeed a correct solution.

3. *Specify, implement, and prove Why3 functions that solve a Sudoku grid.*

3 Extensions

Note that the following extensions are not sorted by increasing difficulty.

3.1 Efficient check of chunk validity

Change the chunk validity checker so that it uses a linear rather than quadratic algorithm, using an extra boolean array, as shown in Figure 4

```
check_valid(g) =
  for i from 0 to 80
    if column chunk for cell i is invalid then return false
    if row chunk for cell i is invalid then return false
    if square chunk for cell i is invalid then return false
  done
  return true
```

Figure 3: Naive grid validity checker

```
check_chunk_valid(g, i, start, offsets) =
  let s be start[i]
  let b be a fresh mutable array indexed from 1 to 9, initialised to False
  for o from 0 to 8
     if cell s + offsets[o] of g contains a number v then
        if b[v] then return false
        set b[v] to True
  done
  return true
```

Figure 4: Pseudo-code for efficient check of chunk validity

3.2 Efficient check of grid validity

Change the validity checker so that, instead of checking the whole grid, it checks only the 3 chunks that are impacted by the modified cell. In other words, change the code

```
for x from 1 to 9
  let g' be g where cell i is filled with x
  if g' is valid then solve(g', i + 1)
done
```

into

```
for x from 1 to 9
  let g' be g where cell i is filled with x
  if column chunk g' i is valid and row chunk g' i is valid and
     square chunk g' i is valid then solve(g', i + 1)
done
```

3.3 Mutable arrays

Change the solver so that it uses in-place modifications inside the original grid instead of creating a new grid for each recursive call (`let g' be g where...`).

3.4 Completeness

Specify and prove that, if the solver returns no solution, there are indeed no solutions. Note that automated provers might have difficulty with such proofs; in that case, either use Coq, or introduce intermediate lemmas which correctness you can justify by hand.