Sudoku in Coq

Laurent Théry
thery@sophia.inria.fr
Marelle Project - INRIA Sophia Antipolis

February 2006

Abstract

This note presents a formalisation done in Coq of Sudoku. We formalize what is a Sudoku, a Sudoku checker and a Sudoku solver. The Sudoku solver uses a naive Davis-Putnam procedure.

1 The Grid

We are going to formalize the grid as a list of integers. In order to express what rows, columns, subrectangles are, we first need to have some basic list operations. The function `take` takes the `n` first element of a list `l`.

```
Fixpoint take (n: nat) (l: list A) {struct n}: list A :=
match l with
    nil => nil
| a::l1 =>
   match n with
     O => nil
   | S n1 => a:: take n1 l1
   end
end.
```

The function `jump` skips the first `n` elements of a list `l`.

```
Fixpoint jump (n: nat) (l: list A) {struct n}: list A :=
match l with
     nil => nil
```

```
|  a::l1 => match n with O => l | S n1 => jump n1 l1 end
end.
```

The function `take_and_jump` takes the `t` first elements then skips `j` elements, this `n` times.

```
Fixpoint take_and_jump (t j n: nat) (l: list A) {struct n}:
   list A :=
   match n with
        O => nil
   | S n1 =>  take t l ++ take_and_jump t j n1 (jump j l)
   end.
```

Now we are ready to consider an arbitrary grid. We take `h` the number of rows of a subrectangle and `w` the number of columns of a subrectangle. So a subrectangle contains `size` cells.

```
Variable h w: nat.
Definition size := h * w.
```

In a grid, there are `size` rows. The function `row` returns the row number `i`.

```
Definition row i (l: list nat) := take size (jump (i * size) l).
```

The function `column` returns the column number `i`.

```
Definition column i (l: list nat) :=
  take_and_jump 1 size size (jump i l).
```

The function `rect` returns the subrectangle number `i`.

```
Definition rect i (l: list nat) :=
  take_and_jump w size h
            (jump (w * (mod i h) +  h * (div i  h) * size) l).
```

In the list that represents the grid, we take the convention that the integer 0 indicates an empty cell. We define the list `ref_list` that contains all the possible value of a cell: 1, 2, . . . , `size`.

```
Fixpoint progression (n m: nat) {struct n}: list nat :=
  match n with O => nil | S n1 => m :: progression n1 (S m) end.

Definition ref_list := progression size 1.
```

2 The Sudoku

A way to define what is a Sudoku is to use the notion of `permutation`.

```
Inductive permutation : list A -> list A -> Prop :=
  | permutation_nil: permutation nil nil
  | permutation_skip: forall a l1 l2,
      permutation l2 l1 -> permutation (a :: l2) (a :: l1)
  | permutation_swap: forall a b l,
      permutation (a :: b :: l) (b :: a :: l)
  | permutation_trans: forall l1 l2 l3,
      permutation l1 l2 -> permutation l2 l3 -> permutation l1 l3.
```

A list of integers is a Sudoku if it has the right length, each of its row is a permutation of `ref_list`, each of its column is a permutation of `ref_list` and each of its subrectangle is a permutation of `ref_list`.

```
 Definition sudoku l :=
   length l = size * size                                 /\
  (forall i, i < size -> permutation (row i l) ref_list)    /\
  (forall i, i < size -> permutation (column i l) ref_list) /\
  (forall i, i < size -> permutation (rect i l) ref_list).
```

3 The Sudoku checker

We can decide if a list is a a permutation of the another one with the function `permutation_dec1`.

```
Definition permutation_dec1 :
  (forall a b : A, {a = b} + {a <> b}) ->
  forall l1 l2, {permutation l1 l2} + {~ permutation l1 l2}.
```

It follows that we can define if a grid is a Sudoku.

```
Definition check: forall l, {sudoku l} + {~sudoku l}.
```

4 The Sudoku solver

In order to solve Sudoku, we are going to translate the problem into a constraint problem and use a Davis-Putnam procedure [1] to solve it.

4.1 Positions, literals, clause and clauses

Positions

A literal represents the fact that a given cell contains a given value. We first define the type `pos` for positions.

```
Inductive pos: Set := Pos: nat -> nat -> pos.
```

Positions starts at 0,0 till `size`-1, `size`-1. A position is valid if it is inside the grid.

```
Definition valid_pos p :=
  match p with Pos x y => x < size /\ y < size end.
```

To enumerate all the positions in a left-right-up-down manner we use the function `next`.

```
Definition next p :=
match p with Pos x y =>
  if eq_nat (S y) size then Pos (S x) 0 else Pos x (S y)
end.
```

We can turn a position into its position in the list that represents the grid with the function `pos2n`

```
Definition pos2n p := match p with Pos x y => x * size + y end.
```

The function `get` retrieves from a list `s` the value inside the cell at position `p`

```
Definition get p l := nth 0 (jump (pos2n p) l) 0.
```

The function `update` changes the value of the position `p` with the new value `v`.

```
Fixpoint subst (n: nat) (v: A) (l: list A) {struct n} : list A :=
  match l with
    nil => nil
  | a :: l1 =>
     match n with O => v :: l1 | S n1 => a :: subst n1 v l1 end
  end.

Definition update p v (l: list nat) := subst (pos2n p) v l.
```

Literals

A literal is then composed of a position and a value.

```
Inductive lit: Set := v : pos -> nat -> lit.
```

Clause

A clause is a disjunction of literals represented as a list. A clause is satisfied if and only if one of its literal is satisfied,

```
Definition clause:= list lit.
```

We provide some operations to manipulate clauses. The function `lit_is_in` checks if a literal is in a clause.

```
Definition lit_is_in: lit -> clause -> bool.
```

The function `lit_insert` adds a literal inside a clause.

```
Definition lit_insert (v: lit) (l: clause): clause :=
```

The function `lit_rm` removes in `c2` all the literals that occur in `c1`.

```
Definition lit_rm (c1 c2: clause): clause := rm _ lit_test c1 c2.
```

The function `clause_merge` appends all the literals in `c1` and `c2`.

```
Definition clause_merge (c1 c2: clause): clause :=
```

Clauses

The problem to solve is a list of clauses. To solve it, we need to satisfy each clause, i.e. we need to satisfy at least one element of each clause.

```
Definition clauses:= list (nat * clause).
```

The integer associated to each clause represents the length of the clause, i.e. the number of literals in the clause. This integer is used to sort this list of clauses from clauses with few literals to clauses with lots of literals. The function `clause_insert` adds a clause in the list of clauses.

```
Definition clause_insert: clause -> clauses -> clauses.
```

The function `clause_merge` appends to list of clauses.

```
Definition clauses_merge: clauses -> clauses -> clauses.
```

The key function for manipulating clauses is the function `clauses_update`. It is used to update the list of clauses removing from the list all the clauses that contains the literal `l` and removing from each clauses all the literals that occurs in `c`.

```
Fixpoint clauses_update (l: lit) (c: list lit) (cs: clauses)
  {struct cs}: clauses :=
  match cs with
    nil => nil
  | (n , c1) :: cs1 =>
      if lit_is_in l c1 then clauses_update l c cs1 else
        let res := lit_rm c c1 in
        clause_insert res (clauses_update l c cs1)
  end.
```

The purpose of this function is to update the constraint when a new fact is known. The literal `c` is this new fact we know to hold (for example the cell 1, 2 contains 3). The list of literals `c` is the facts we know not to hold (for example the cell 1,2 does not contain 1, the cell 1,2 does not contain 2, the cell 1,2 does not contain 4, . . .).

4.2 Generating initial constraints

In order to generate constrains, we need some list. The list `indexes` contains all the possible index for a position from 0 to `size` -1.

```
Definition indexes := progression size 0.
```

The list `cross` contains all the position (x,y) with $0 \leq x <$ `h` and $0 \leq y <$ `w`.

```
Definition cross :=
  let p := progression h 0 in
  let q := progression w 0 in
  fold_right (fun x l => (map (fun y => (Pos x y)) q) ++ l) nil p.
```

The list `cross1` contains all the pairs (x,y) with $0 \leq x <$ `size` and $1 \leq y \leq$ `size`.

```
Definition cross1 :=
  let p := indexes in
  let q := ref_list in
  fold_right (fun x l => (map (fun y => (x, y)) q) ++ l) nil p.
```

The function `gen_row` generates the constrain that the row number `i` contains the value `z`.

```
Definition gen_row i z :=
  fold_right (fun y l => lit_insert (v (Pos i y) z) l) nil indexes.
```

The function `gen_column` generates the constrain that the column number `i` contains the value `z`.

```
Definition gen_column i z :=
  fold_right (fun x l => lit_insert (v (Pos x i) z) l) nil indexes.
```

The function `gen_rect` generates the constrain that the subrectangle number `i` contains the value `z`.

```
Definition gen_rect i z :=
  let x := h * div i h in
  let y := w * mod i h in
  fold_right (fun p l => lit_insert (v (shift p x y) z) l)
    nil cross.
```

where `div` and `mod` are the division and the modulo respectively. The function `gen_cell` generates the constrain that the cell at position `p` contains a value.

```
Definition gen_cell p :=
  fold_right (fun z l => lit_insert (v p z) l) nil ref_list.
```

The function `all_cell` generates the constraint that all the cells of the grid contain a value.

```
Definition all_cell :=
  let c0 := cross2 in
  (fold_right (fun p l =>  let res := gen_cell p in
                              clause_insert res l) nil c0).
```

To generate the initial constrains, we just accumulate the clauses that each number appears in each row, each number appears in each column, each number appears in each subrectangle and finally each cell contains a value.

```
Definition init_c :=
  let c1 := cross1 in
  fold_right
     (fun iz l => let res := gen_row (fst iz) (snd iz) in
                             clause_insert res l)
```

```
 (fold_right
     (fun iz l => let res := gen_column (fst iz) (snd iz) in
                             clause_insert res l)
 (fold_right
     (fun iz l => let res := gen_rect (fst iz) (snd iz) in
                             clause_insert res l)
  all_cell c1) c1) c1.
```

These constrains are redundant but most importantly a list that satisfies these constraints is a Sudoku. Note that the initial constrain only ensures that the list `ref_list` is included in each row, each column and subrectangle. It is sufficient because the list `ref_list` is composed of unique elements.

```
Inductive ulist : list A ->  Prop :=
  ulist_nil: ulist nil
 | ulist_cons: forall a l, ~ In a l -> ulist l ->  ulist (a :: l) .


Theorem ref_list_ulist : ulist ref_list.
```

So we can apply a derived version of the pigeon-hole principle.

```
Theorem ulist_eq_permutation: forall (l1 l2 : list A), ulist l1 ->
  incl l1 l2 -> length l1 = length l2 ->  permutation l1 l2.
```

where `incl` is the inclusion predicate.

The initial constrains are the ones that need to be satisfied from an empty grid. For a solving a given Sudoku, we have to incorporate the information of that are already present in order to simplify the list of clauses. This is done by the function `gen_init_clauses_aux` that walks through the initial grid `s` and update the list of clauses `c`. The first element of this grid corresponds to the position `p`

```
Fixpoint gen_init_clauses_aux (s: list nat) (p: pos) (c: clauses)
  {struct s}: clauses :=
  match s with
   nil => c
  | a :: s1 =>
    let p1 := next p in
    let l1 := v p a in
      if (In_dec eq_nat a ref_list) then
         let c1 := clauses_update l1 (anti_literals l1) c in
           gen_init_clauses_aux s1 p1 c1
      else gen_init_clauses_aux s1 p1 c
  end.
```

The main function is gen_init_clauses.

```
Definition gen_init_clauses s :=
  gen_init_clauses_aux s (Pos 0 0) init_c.
```

5 Finding one solution

To organize the search for a solution, we have to deal with Coq necessity to write structural recursive function in Coq. The first function we write is the function try_one that given the current state of the Sudoku and the list of constrains cs tries to satisfies the clause c. For this, it picks the literal of c one after another, supposes it holds and call the continuation f to check if the result is satisfiable. If so, it returns the solution, otherwise it tries the next literal.

```
Fixpoint try_one (s: list nat) (c: clause)
                 (cs: clauses)
                 (f: list nat -> clauses -> option (list nat))
                 {struct c}:
   option (list nat) :=
   match c with
      nil => None
    | (v p z) as k:: c1 =>
         let s1 := update p z s in
         let cs1 := clauses_update k (anti_literals k) cs in
         match f s1 cs1 with
           None => try_one s c1 cs f
         | Some c1 => Some c1
         end
   end.
```

The function find_one_aux is the one that organizes the recursion. It takes the current state of the Sudoku s and the corresponding list of clauses cs. It looks at the top of the list of clauses. Remember that the list is ordered, so clauses with few literals comes first. If the list contains the empty clause, this clause is unsatisfiable so there is no solution. Otherwise we take the first clause and tries to satisfy it using try_one. The argument n is a dummy argument put there to ensures structural recursion. As at each recursion we remove at least one clause from the list of clauses, taking the initial list of clauses as the dummy structural argument is sufficient.

```
Fixpoint try_one (s: list nat) (c: clause) (cs: clauses)
                 (f: list nat -> clauses -> option (list nat))
                 {struct c}:
   option (list nat) :=
   match c with
      nil => None
    | (v p z) as k:: c1 =>
         let s1 := update p z s in
         let cs1 := clauses_update k (anti_literals k) cs in
         match f s1 cs1 with
           None => try_one s c1 cs f
         | Some c1 => Some c1
         end
   end.
```

The main function is then

```
Definition find_one s :=
  let cs := gen_init_clauses s in find_one_aux cs s cs.
```

To state the correctness of this algorithm, we need to define the predicate that indicates if a grid is a refinement of another grid

```
Definition refine s1 s2 :=
  length s1 = size * size /\
  length s2 = size * size /\
  forall p, valid_pos p ->
    In (get p s1) ref_list -> get p s1 = get p s2.
```

With this predicate, the correctness is expressed as

```
Theorem find_one_correct:
  forall s, length s = size * size ->
    match find_one s with
      None => forall s1, refine s s1 -> ~ sudoku s1
    | Some s1 => refine s s1 /\ sudoku s1
    end.
```

6 Finding all solutions

It is trivial to modify the previous algorithm so that it returns not the first solution but all the solutions. The function `try_all` accumulates all the solution instead of stopping at the first one.

```
Fixpoint try_all (s: list nat) (c: clause) (cs: clauses)
                 (f: list nat -> clauses -> list (list nat))
                 {struct c}:
   list (list nat) :=
   match c with
      nil => nil
    | (v p z) as k:: l1 =>
         let s1 := update p z s in
         let cs1 := clauses_update k (anti_literals k) cs in
          merges (f s1 cs1) (try_all s l1 cs f)
   end.
```

where the function

```
     merges:list nat -> list (list nat) -> list (list nat)
```

inserts a solution in a list of solutions.

```
Fixpoint find_all_aux (n: clauses) (s: list nat) (cs: clauses)
  {struct n}: list (list nat) :=
match cs with
   nil => s :: nil
| (_, nil) :: _  => nil
| (_, p) :: cs1 =>
    match n with
      nil => nil
    | _ :: n1 =>
       try_all s p cs1 (find_all_aux n1)
    end
end.
```

The main function is then

```
Definition find_all s :=
  let cs := gen_init_clauses s in
      find_all_aux cs s cs.
```

The corresponding correctness statement is the following.

```
Theorem find_all_correct:
  forall s s1, refine s s1 -> (sudoku s1 <-> In s1 (find_all s)).
```

7 Running example

To use the Sudoku contribution, we first need to load it.

```
Require Import Sudoku.
```

Suppose we want to solve 3x3 Sudoku, we first create the initial position taking the convention that zeros correspond to empty cells.

```
Definition test :=
  5 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 0 ::
  0 :: 4 :: 0 :: 8 :: 1 :: 0 :: 0 :: 0 :: 0 ::
  0 :: 9 :: 3 :: 0 :: 0 :: 0 :: 0 :: 0 :: 2 ::
  0 :: 0 :: 0 :: 0 :: 0 :: 0 :: 2 :: 0 :: 3 ::
  9 :: 0 :: 0 :: 7 :: 0 :: 0 :: 0 :: 0 :: 0 ::
  2 :: 3 :: 0 :: 0 :: 0 :: 6 :: 0 :: 7 :: 0 ::
  3 :: 6 :: 5 :: 1 :: 0 :: 0 :: 0 :: 0 :: 0 ::
  0 :: 0 :: 0 :: 0 :: 5 :: 0 :: 8 :: 0 :: 0 ::
  0 :: 0 :: 1 :: 0 :: 7 :: 0 :: 6 :: 0 :: 0 :: nil.
```

Doing

```
Eval compute in find_one 3 3 test.
```

returns

```
  5 :: 8 :: 6 :: 2 :: 3 :: 7 :: 9 :: 1 :: 4 ::
  7 :: 4 :: 2 :: 8 :: 1 :: 9 :: 3 :: 5 :: 6 ::
  1 :: 9 :: 3 :: 4 :: 6 :: 5 :: 7 :: 8 :: 2 ::
  6 :: 5 :: 7 :: 9 :: 8 :: 1 :: 2 :: 4 :: 3 ::
  9 :: 1 :: 4 :: 7 :: 2 :: 3 :: 5 :: 6 :: 8 ::
  2 :: 3 :: 8 :: 5 :: 4 :: 6 :: 1 :: 7 :: 9 ::
  3 :: 6 :: 5 :: 1 :: 9 :: 8 :: 4 :: 2 :: 7 ::
  4 :: 7 :: 9 :: 6 :: 5 :: 2 :: 8 :: 3 :: 1 ::
  8 :: 2 :: 1 :: 3 :: 7 :: 4 :: 6 :: 9 :: 5 :: nil
```

Doing

```
Eval compute in length (find_all 3 3 test).
```

returns 1.
The source code is available at

ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Sudoku.zip

References

[1] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.