

# Specification and Verification of N-Queen Problem

Ashish Bhangale, Kewal Karavinkoppa, Steven Johnson  
Indiana University  
{aabhanga, kkaravin, sjohnson}@cs.indiana.edu

## Abstract

This article details the formal specification and verification of the n-queen problem. The main purpose of this article is to specify and verify the abstract model of a 2-lines C program which computes all the possible solutions to the n-queen problem.

*Keywords:* Specification, Verification, PVS, N-Queen, termination, correctness, completeness

## Introduction<sup>4</sup>

In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has N rows and N columns. The N by N Queen's problem asks how to place N queens on an N x N chess board so that none of them can hit any other in one move. Thus, a solution requires that no two queens share the same row, column, or diagonal. Over the years this problem is seen as a classic problem in Artificial Intelligence, Constraint Satisfaction and Constraint Programming.

The article looks into the details of the formal specification and verification of the 2-line C program for n-queen problem which apparently is authored by Marcel van Kervinc and appeared on C signature programs

```
t(a,b,c){int d=0,e=a~b~c,f=1;if(a)for(f=0;d=(e==d)&-e;f+=t(a-d,(b+d)*2,(c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

This program takes n as input which is smaller than the machine word size and computes the number of possible ways n queens can be placed on a n x n chessboard. Thus it can be seen as a function f(n). More surprisingly this is a very efficient program to compute this number. This code does not only consider unique solutions but counts all the possible solutions.

## Algorithm<sup>3</sup>

The program finds solutions by starting with a queen in the top left corner of the chess board. It then places a queen in the second column and moves it until it finds a place where it cannot be hit by the queen in the first column. It then places a queen in the third column and moves it until it cannot be hit by either of the first two queens. Then it continues this process with the remaining columns. If there is no place for a queen in the current column the program goes back to the preceding column and moves the queen in that column. If the queen there is at the end of the column it removes that queen as well and goes to the preceding column. If the current column is the last column and a safe place has been found for the last queen, then a solution of the puzzle has been found. If the current column is the first column and its queen is being moved off the board then all possible configurations have been examined, all solutions have been found, and the algorithm terminates. When a solution has been found it adds to the list of solutions. Currently the program does not eliminate solutions that can be obtained from previous ones by rotations or reflections. A measure of the difficulty of the problem is given by the number of moves the above

algorithm takes to find the first solution. This number of course tends to go up as N increases, but it does not increase monotonically.

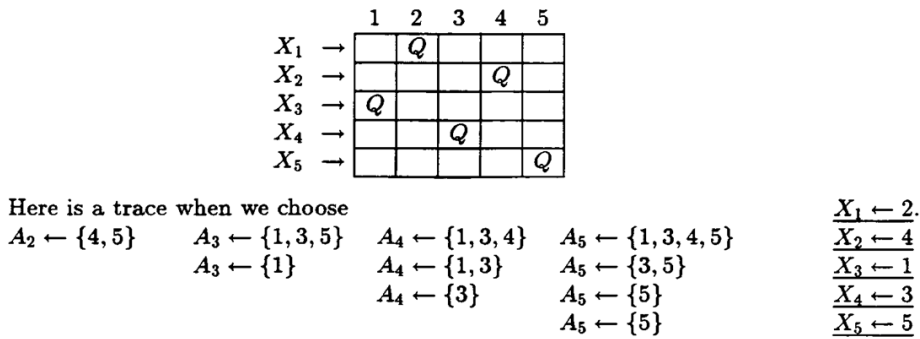
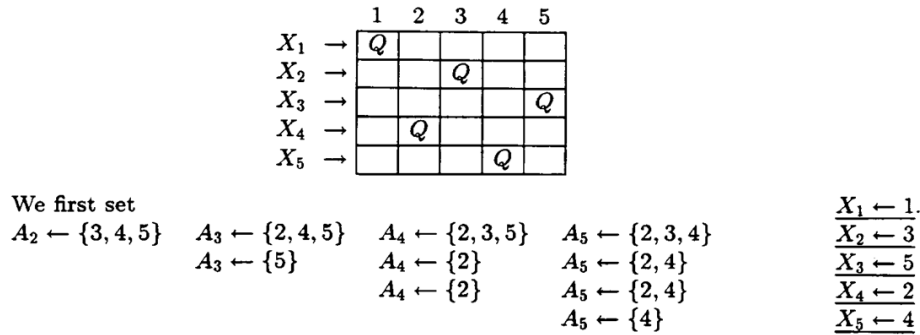


Figure 1: Solving 5 x 5 Queen Problem<sup>2</sup>

### Abstraction<sup>4</sup>

The 2-line C code can be abstracted as shown below

```

int t(a, b, c)
  f ← 1
  if a ≠ ∅
    e ← (a\b)\c
    f ← 0
    while e ≠ ∅
      d ← min_elt(e)
      f ← f + t(a\d, succ(b □ {d}), pred(c □ {d}))
      e ← e\d
  return f

```

### Understanding variables

The parameters a, b and c are to be seen as array which represent a row on the chessboard. And each element in the array represents the state of the column.

Parameter  $a$  depicts the notion of 0 imply that a queen cannot be placed in that column and 1 imply that a queen can be placed in that column. It maps the queens only vertically.

Parameter  $b$  depicts the notion of 1 imply that a queen cannot be placed in that column and 0 imply that a queen can be placed in that column. It maps the queens only diagonally top-left to bottom-right.

Parameter  $c$  depicts the notion of 1 imply that a queen cannot be placed in that column and 0 imply that a queen can be placed in that column. It maps the queens only diagonally top-right to bottom-left.

Parameter  $e$  depicts the notion of 1 imply the position where queen has to be placed in the current pass.

### Understanding functions

$a \setminus b$ :  $a \setminus b$  computes an array such that the output array contains elements in  $a$  and not in  $b$  ( $a$  AND NOT  $b$ ). The abstraction consists of  $e \leftarrow ((a \setminus b) \setminus c)$ . This implies that  $e$  not is computed such that elements in  $a$  and not in  $b$  and not in  $c$ . On a chessboard this actually computes the position where the queen can be placed in the next pass considering  $a$  (vertical),  $b$  (diagonal) and  $c$  (diagonal).

$\text{min\_elt}$ : This accepts  $e$  and returns an array which is stored back in  $d$  such that  $d$  has the smallest element of  $e$ . This computes the position of queen amongst available positions of  $d$ . Thus only one queen is placed in one pass and as the output array contains only 1, thus, only one queen can be placed on each row. Thus  $d \leftarrow \text{min\_elt}((a \setminus b) \setminus c)$  ensures that  $d$  has either all 0 or maximum one 1.

$\square$ : This implies union operation.  $a \square b$  and  $a \square c$  ensure that the vertical and diagonal positions are considered.

$\text{succ}$ : This calculates the diagonally top-left to bottom-right attacking position for the given chessboard.  $\text{succ}$  operation involves adding of 0 to the end and removing one start element and shifting one position to the right.

$\text{pred}$ : This calculates the diagonally top-right to bottom-left attacking position for the given chessboard.  $\text{pred}$  operation involves adding of 0 to the start and removing one end element and shifting one position to the left.

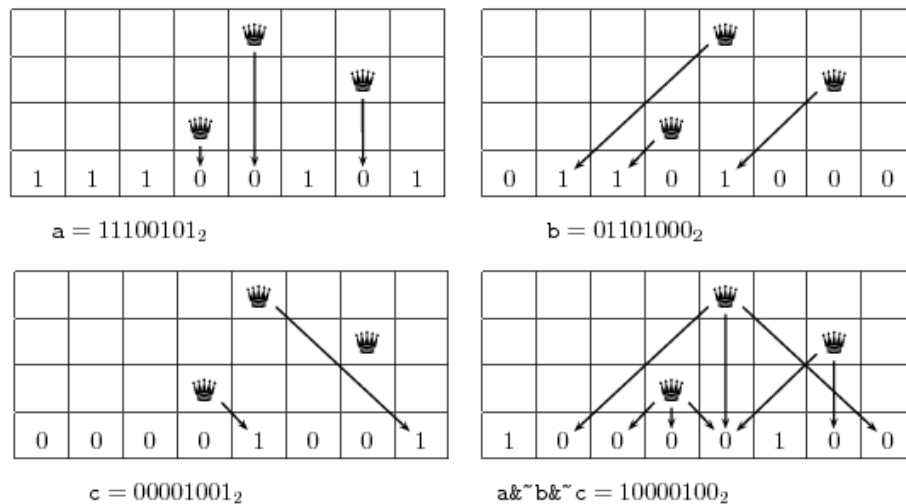


Figure 2: Understanding variables and functions<sup>4</sup>

The fourth board shows that  $e$  is computed using  $a$ ,  $b$  and  $c$  by the formula  $(a \setminus b) \setminus c$ . Now,  $\min\_elt$  of  $e$  is stored into  $d$ .  $\min\_elt$  of  $e$  gives the smallest element of  $e$ . Thus new value of  $1000000_2$  will be stored in  $d$ . This implies that the queen will be placed at first column in the current pass.

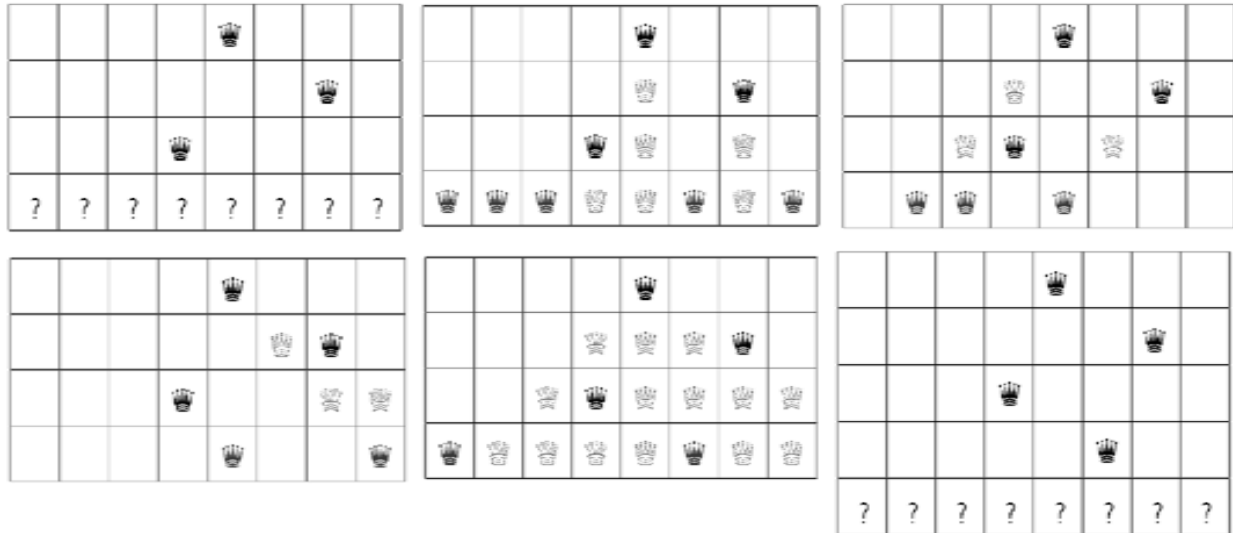


Figure 3: Positioning of a queen on row 4

This board 1 shows the start position. Board 2 shows the possible positions after considering the vertical positions. Board 3 shows the possible positions after considering the diagonal positions from right-top to left-bottom. Board 4 shows the possible positions after considering the diagonal positions from left-top to right-bottom. Using the board 2, board 3 and board 4 safe states for the next row is calculated.

### Understanding conditions

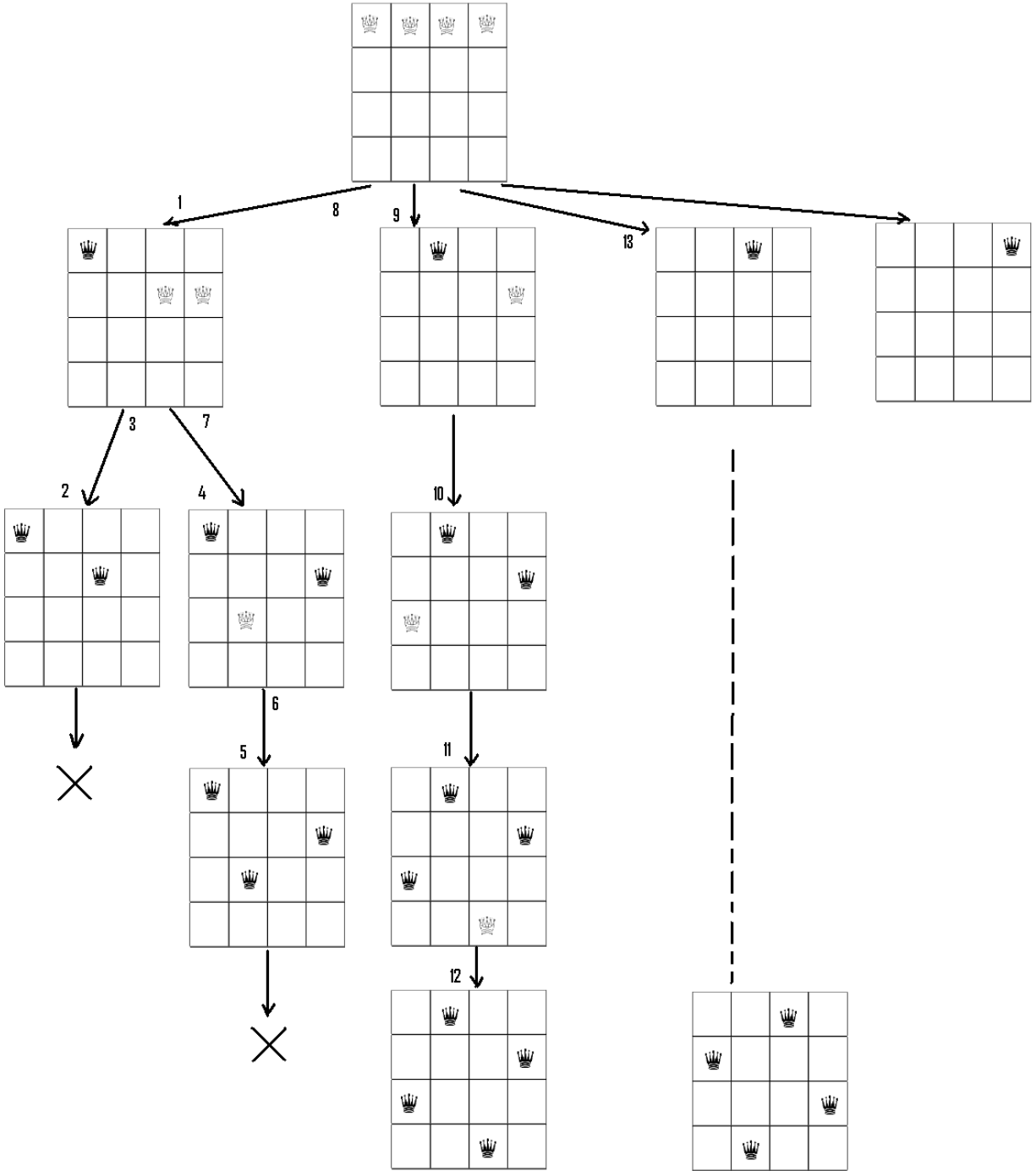
We have two conditions in the code which can be explained as follows

$a \neq \emptyset$  : This condition checks if there exists any column on the board where there is no queen. If such column exists then only it proceeds to the while loop which calculates the possible solutions starting from the gives  $a$ ,  $b$  and  $c$ .

$e \neq \emptyset$  : This condition is the termination condition for the while loop. This checks if there exists a location on the board where a queen can be placed. If such a location is exists then the control proceeds with the body of the while otherwise it returns from the loop or does not enter the loop.

### Understanding Recursion and Backtracking<sup>132</sup>

The above mentioned algorithm uses recursion and backtracking to get the all possible solutions. The conditions determines, whether the control returns from a function call or backtracks. The algorithm places a queen at the start row and recurs to find the position of the queen in the next row. The recursion looks at this as new problem of its own and tries to find a position where a queen can be placed. If a position is found then it again recurs to find the position of next queen. And if a position is not found then it backtracks. Then the control is passed back to the parent function from where a new position for the queen is searched using the original algorithm that is it again recurs. This process continues till the time the algorithm sees all the possible solutions for the given chessboard.



The above figure shows the exact stack trace how the program solves the 4 x 4 chessboard to get the 2 possible solutions. The numbers indicate the flow of control (important to map the values in the variables). It suggests that when a function returns and when the algorithm backtracks. The function returns from 12 to 13. The algorithm backtracks from 2 to 3 to 4 as no further expansion of 2 gives any valid solutions, the algorithm backtracks.

## Goal

We define the following terms:

1. Termination  
The program terminates and does not hang or crash.
2. Correctness  
All the solutions returned by the program are safe that is satisfies the constraints.
3. Completeness  
A brute force method is applied to generate solutions. Out of this all the safe solutions satisfying the constraints are chosen. Now this set of solutions must be included in the solution returned by the program.

Our goal is to prove the following:

1. The program terminates
2. The program produces correct output
3. The program produces complete output

## Specification

We formulate the algorithm in PVS system<sup>6</sup> using 2 theories:

### Theory 1: Board

We represent a chess board as array of array.

```
Board[SIZE:posint]: THEORY
BEGIN
  range:      TYPE = { k: int | 1 <= k AND k <= SIZE }
  row_t:      TYPE = ARRAY [range -> bool]
  board_t:    TYPE = ARRAY [range -> row_t]
  solution_t: TYPE = list[board_t]
```

This theory accepts SIZE as a subtype. Thus this builds a chess board of SIZE x SIZE.

range: Iteration from first array index to array SIZE.

row\_t: is a boolean array representing a row of a chessboard. A FALSE represents an empty cell, while TRUE represents a queen is holding position in this cell.

board\_t: An array of rows representing the entire chessboard.

solution\_t: This is the set of all possible distinct solutions for the given chessboard size.

### Theory 2: Algorithm

This theory is the formulation of the 2-line C code. Following are PVS functions:

```
isMember(row:row_t):bool = EXISTS (c:range): row(c)
```

Parameters:

row: Chessboard row.

Output: True/False

Description: This function operates on a single row, checks if a queen is placed in the row. If yes, returns TRUE, FALSE otherwise.

```
appendRow_Board(brd, row, r): RECURSIVE board_t =
  IF r = SIZE
    THEN EmptyBoard
  ELSIF isMember(brd(r))
    THEN appendRow_Board(brd, row, r+1)
    ELSE setRow_Board(brd, r, row)
  ENDIF
  MEASURE SIZE - r

appendRow_Board(brd, row): board_t = appendRow_Board(brd, row, 1)
```

Parameters:

brd: Chessboard

row: Row to append (replace),

r: row iterator.

Output: Chessboard with new row appended (replaced).

Description: This function the first empty row with not queen and replaces it with the row provided a parameter.

```
slash(row1, row2: row_t):row_t = (lambda (r): row1(r) AND NOT row2(r))
```

Parameters:

row1: row1 represents a row in which a queen can be placed

row2: represents a row in which queens cannot be placed.

Output: A row where a queen can be placed.

Description: This function accepts 2 rows. First row represents a row in which a queen can be placed considering the vertical arrangements of the queens already placed on the chessboard while second row is the row in which the queens cannot be placed considering the diagonal attacking positions of the queens already placed on the chessboard. It computes the possible location where a queen can be placed safely in the current row and returns this new row.

```
union(row1, row2: row_t):row_t = (lambda (r): row1(r) OR row2(r))
```

Parameters:

row1 a row where queens cannot be placed and

row2 where a queen is placed in the current pass.

Output: A row where queen cannot be placed.

Description: This function makes a union of the previous impossible positions of queen with the current placing of queen and returns a row where a queen cannot be placed.

```
min_elt(row):row_t =
  (lambda (r:range): (row(r) AND NOT(EXISTS(i: int|i >= 1 AND i <
r):row(i)))
  )
```

Parameters:

row: Chessboard row.

Output: Row which contains the minimum information from the input row.

Description: This accepts 'row' and returns an row\_t such that it has the smallest element of 'row'. This computes the position of queen amongst available positions. Thus only one queen is placed in one pass and as the output 'row' contains only 1, thus, only one queen can be placed on each row.

```
succ(row, b): row_t =
  (LAMBDA (c): IF c = SIZE THEN b ELSE row(c+1) ENDIF)

pred(row, b): row_t =
  (LAMBDA (c): IF c = 1 THEN b ELSE row(c-1) ENDIF)
```

Parameters:

row : chessboard row.

b: The value to be appended.

Output: Row in which queen cannot be placed used for the next pass

Description: 'succ' function calculates the diagonally top-left to bottom-right attacking position for the given chessboard. 'succ' operation involves adding of 0 to the end and removing one start element and shifting one position to the right.

'pred' function calculates the diagonally top-right to bottom-left attacking position for the given chessboard. 'pred' operation involves adding of 0 to the start and removing last element and shifting one position to the left.

```
F(v: nat, a_row, b_row, c_row:row_t, brd:board_t, sol:solution_t, e_row:
row_t):
  RECURSIVE solution_t =
  IF v = 1
  THEN IF isMember(a_row)
        THEN F(2, a_row, b_row, c_row, brd, null, slash(slash(a_row,
b_row), c_row))
        ELSE cons(brd, null)
        ENDIF
  ELSIF isMember(e_row)
  THEN F( 2
```

```

    , a_row
    , b_row
    , c_row
    , brd
    , append(sol,
        F( 1
            , slash(a_row, min_elt(e_row))
            , succ(union(b_row, min_elt(e_row)), FALSE)
            , pred(union(c_row, min_elt(e_row)), FALSE)
            , appendRow_Board(brd, min_elt(e_row), 1)
            , sol
            , e_row)
        )
    , slash(e_row, min_elt(e_row))
)
ELSE sol
ENDIF
MEASURE exp2(SIZE + 1) - length(sol)

```

#### Parameters:

`a_row`: depicts the notion of 0 imply that a queen cannot be placed in that column and 1 imply that a queen can be placed in that column. It maps the queens only vertically.

`b_row` : depicts the notion of 1 imply that a queen cannot be placed in that column and 0 imply that a queen can be placed in that column. It maps the queens only diagonally top-left to bottom-right.

`c_row` : depicts the notion of 1 imply that a queen cannot be placed in that column and 0 imply that a queen can be placed in that column. It maps the queens only diagonally top-right to bottom-left.

`brd` : chessboard

`sol` : set of solution computed till now

`e_row` : depicts the notion of 1 imply the position where queen has to be placed in the current pass.

Output: `solution_t` a set of solutions

Description: This function is recursive and uses backtracking in order to compute the chessboard solutions. The conditions determines, whether the control returns from a function call or backtracks. The algorithm places a queen at the start row and recurs to find the position of the queen in the next row. The recursion looks at this as new problem of its own and tries to find a position where a queen can be placed. If a position is found then it again recurs to find the position of next queen. And if a position is not found then it backtracks. Then the control is passed back to the parent function from where a new position for the queen is searched using the original algorithm that is it again recurs. This process continues till the time the algorithm sees all the possible solutions for the given chessboard.

```

queen: solution_t =
  F( 1
    , FullRow
    , EmptyRow
    , EmptyRow
    , EmptyBoard
    , null
  )

```

```
, EmptyRow
)
```

Parameters: None.

Output: `solution_t` a set of solutions

Description: This is the wrapper function for the main F function.

## Verification

For verification we have written 2 theories:

### Theory 1: Verification

This theory implements the functions used in the unit test for output verification. We check the solutions generated by the *'queen'* function for completeness and correctness.

For correctness, we implement the `correct?` which checks if a given set of board configurations meets the requirements, i.e. none of the 2 queens are in attacking positions which means no 2 queens are in same column, row or diagonally aligned and the number of generated boards is what is expected for the given size of the board.

For completeness, we use the predicate *'complete?'* which compares the solutions generated by *'queen'* with its own generated solutions. For completeness, we generate all possible board configurations for the given size and pass it to the verifier. The verifier eliminates the boards that do not satisfy the condition, i.e. it eliminates if 2 queens are placed in attacking position which means either the queens are aligned diagonally or in same column or same row. Thus the boards that pass the verifier are the possible solutions since we prune through a huge set of all possible board configuration to the ones that satisfy the condition. This implies that we test the output for complete solution.

```
correct?(soln:solution_t, len:int):bool = solutions?(soln) AND length(soln) = len
```

Parameters:

`soln` : list of chessboards generated a solution.

`len` : number of solutions generated.

Output: True/False if the solutions passed is correct.

Description: This method checks if the chessboard configuration generated as solution for the N Queen problem is correct. It uses the *'solutions?'* to check if the solutions are correct and the number of generated boards is what is expected for the size of the board.

```
solutions?(soln:solution_t): bool = every(solution?) (soln)
```

Parameters:

`soln` : list of chessboards generated a solution.

Output: True/False if the solutions passed is correct.

Description: This method checks if the chessboard configuration generated as solution for the N Queen problem is correct. It uses the 'solution?' to check if the solutions are correct.

```
solution?(brd:board_t): bool = horz?(brd) AND vert?(brd) AND diag?(brd)
```

Parameters:

brd : chessboard

Output: True/False if the board is correct.

Description: This function checks if the given chessboard meets the required configurations of the queens on the board. If yes, returns True, False otherwise.

```
horz?(brd:board_t):bool =
  (FORALL (b:range): (count(brd(b), 1, 0) = 1))

vert?(brd:board_t):bool =
  horz?(transpose(brd))

diag?(brd:board_t):bool =
  (FORALL (i:range):
    (FORALL (j: int | 1 <= j AND j < i):
      NOT (abs(i - j) = abs(getPosition(brd(i), 1) - getPosition(brd(j),
1))))))
```

Description: These are the methods called by 'solution?' to check if the queens are places in came row, column and aligned diagonally.

```
complete?(sol:solution_t):bool =
  compare(sol, getComplete)
```

Parameters:

sol: list of chessboards generated a solution.

Output: True/False if the solutions passed is correct.

Description: This function checks for completes of the solutions. It takes the generated solution, generated it own set of complete solutions and calls 'compare' to compare the solutions if complete.

```
verifierComplete(sol:solution_t): RECURSIVE solution_t =
  CASES sol OF
    null : null,
    cons(a, tl) :
      IF solution?(a) THEN
        append(cons(a, null), verifierComplete(tl))
      ELSE
        verifierComplete(tl)
      ENDIF
  ENDCASES
  MEASURE sol BY <<
```

```
getComplete:solution_t =
    verifierComplete(genAllPerm)
```

Description: getComplete functions calls the verifierComplete function which returns the complete set of solutions. It passes all the possible board configurations generated by genAllPerm to verifierComplete.

verifierComplete: function prunes the unnecessary boards and returns only the correct solutions.

```
genPerm(i, j:int, orig: board_t): RECURSIVE solution_t =
    IF j <= SIZE THEN
        append(
            addRB(
                orig(j),
                genPerm(
                    1,
                    j + 1,
                    orig
                )
            ),
            genPerm(
                i + 1,
                j - 1,
                orig
            )
        )
    ELSE
        null
    ENDIF
    MEASURE SIZE - j
```

```
genCombo(i:int, brd:board_t): RECURSIVE board_t =
    IF i <= SIZE THEN
        genCombo(i + 1, brd WITH [(i) := setBool_Row(EmptyRow, i)])
    ELSE
        brd
    ENDIF
    MEASURE SIZE - i
```

```
genAllPerm:solution_t =
    genPerm(1, 1, genCombo(1, EmptyBoard))
```

Description: genAllPerm<sup>8</sup> generates all the possible board configurations. It uses the genCombo method which returns the possible ways in which a queen can be places in one row. This is then passed to genPerm which generates all possible permutations of these rows , resulting in a full set of possible boards in which a queen can be please in each row of a chess board.

### Invariants

```
int t(a, b, c)
a. f ← 1
b. if a ≠ ∅
    i. e ← (a\b)\c
    ii. f ← 0
    iii. while e ≠ ∅
        1. d ← min_elt(e)
```

```

2. f ← f + t(a\{d}, succ(b □ {d}), pred(c □ {d}))
3. e ← e\{d}
c. return f

```

- a) Here after line a. 'a' should have atleast one 1 implies the board should have column without a queen can be placed.
- b) After iii. 'e' should have atleast one possible position where a queen can be placed. So, count of 1 in e should be greater than or equal to 1.
- c) Before iii. 'e' may have any number of 1's.
- d) After 1.'d' should have exactly one 1 in the array. So, count of 1 in d should be equal to 1.
- e) The board under condition inside the while loop represents a partial board. Thus in array representation format the rows without queen will be represented with all false. Thus, if we ignore the atleast one queen on each row constraint and modify it to maximum 1 queen on each row then the board should be verifiable and correct. That is it must satisfy the constraints.
- f) In the PVS formulation or scheme representation the point where we add a board to the list of solutions the board should be verifiable and correct.
- g) When iii. fails and the board is not complete then we backtrack. In such condition the number of 1s in e before backtracking should be less than the number of 1s in e after backtracking.
- h) The number of 1s in a/d should be less than number of 1s in a when inside the while loop.

## Theory 2: Unit\_Test

The unit test theories contains claim regarding the correct and desired functionality of the PVS formulations of the scheme equivalent.

## Current Status

Translation of 2-line C code to scheme implementation is complete and working. We implement other crude method to generated possible solutions for the N Queens problem.

Our first attempt to formulate the solution for N Queen problem was using list representations of chessboards. List representation formulation is complete and was proved for board of size 1 and 2.

Later the formulation was changed to use 2D arrays to represent chessboard. Arrays are best suited for chessboard representation since any board cell can be accessed just using 2D array indices. Proof for some of the unit tests is complete. The full regressive unit testing of all PVS functions is not complete.

## Future Work

More unit tests need to be included for through testing especially for the methods involving for the correctness checking.

Generalization needs to be proved, and more ground work needs to be completed before proving that the algorithm solves the N Queens problems for any given 'n', size of the board.

It would be interesting to try and approach this problem without using the internal stack used in recursion. But in that case the state of the chess board has to be saved. Intermediate partial board state can be pushed to a stack and

popped at the time of function return and backtracking. The alternative approach which can be used is storing arrays `a`, `b` and `c`. These arrays can be used to reconstruct the `a`, `b` and `c` arrays from the previous state.

Different techniques like using heuristics or Hill Climbing algorithm, etc can be explored to get the other solution to the N-queen problem as an alternative to the naïve brute force method

## Related Work

Jean-Christophe Filliatre, LRI, University of Paris Sud, France proved the termination and correctness of the 2-line C code using Caduceus, a C program prover tool.

## Conclusion

We have presented the formal verification of an extremely complex 2-line C program using the PVS.

## References

1. ANSI C (recursive, congruence-free NxN-size queens problem solver with conflict heuristics), [http://www.0xe3.com/src/chess/chess\\_advanced.c](http://www.0xe3.com/src/chess/chess_advanced.c)
2. Constraint Satisfaction Problems, <http://www.ics.uci.edu/~dechter/courses/ics-175a/spring-2004/ics-270a-lect-05.ppt>
3. Foundations of Constraint Satisfaction by Edward Tsang
4. Jean-Christophe Filliatre, Queens on a Chessboard: an Exercise in Program Verification, Unpublished, January 2007, <http://why.lri.fr/queens/>
5. Rok Sasic, Jun Gu, Fast Search Algorithms for the N-Queens Problem (1991), IEEE Trans. on Systems, Man, and Cybernetics
6. S. Owre, J. M. Rushby, N. Shankar, The PVS System, <http://pvs.csl.sri.com/>
7. Wikipedia, "Eight queens puzzle", [http://en.wikipedia.org/wiki/Eight\\_queque\\_puzzle](http://en.wikipedia.org/wiki/Eight_queque_puzzle)
8. <http://paste.lisp.org/display/11447>, Computing permutations in scheme

## Appendix

The scheme implementation along with sample output, PVS formulation and the status of proofs for the PVS formulation is added for reference.

## Queen.ss

```
;;
;;
;; Project: Specification and Verification of N Queen problem.
;;
;; Course: P515
;;
;; Description: This file is the scheme implementation of the 2-line C program
;;              for n-queen problem which apparently is authored by Marcel van Kervinc
;;              and appeared on C signature programs.
;;
;;              C Program -
;;              t(a,b,c){int d=0,e=a~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,
(c+d)/2));return
;;              f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
;;
;; Implementation tested for Chez Scheme Version 7.4, Copyright (c) 1985-2007
Cadence Research Systems
;;
;; Authors:
;; Ashish Bhangale <aabhanga@indiana.edu>
;; Kewal Karavinkoppa <kkaravin@indiana.edu>
;;
;;

;;
;; removes the last cell from the row
;;
(define (remove-last row res)
  (if(<= (length row) 1)
      res
      (remove-last (cdr row) (append res (cons (car row) '()))))
  ))

;;
;; returns a row with appends the provided list to a row, adjusting it to the board size
;; padding a's in order to make it the list of board size.
;;
(define (append-with row ca size)
  (cond
    [(equal? (length row) size) row]
    [(< (length row) size) (append-with (cons ca row) ca size)]
  ))

;;
;; min-elt-helper helper routine for min-elt
;; This accepts 'row' and returns an row such that it has the smallest element of 'row'. This
;; computes the position of queen amongst available positions. Thus only one queen is placed in
one
;; pass and as the output 'row' contains only 1, thus, only one queen can be placed on each row.
;;
(define (min-elt-helper row)
  (if(null? row)
      '()
      (if(equal? (car row) #f)
          (cons #f (min-elt-helper (cdr row)))
          (cons #t '()))))
  )))

(define (min-elt row)
  (reverse (min-elt-helper (reverse row))))

;;
;; slash-helper helper routine for slash operation.
;;
(define (slash-helper a b ca cb size)
  (if(null? a)
      '()

```

```

                (cons (and (car a) (not (car b))) (slash-helper (cdr a) (cdr b) ca cb
size))
        ))

;;
;; This function accepts 2 rows. First row represents a row in which a queen can be placed
;; considering the vertical arrangements of the queens already placed on the chessboard while
second
;; row is the row in which the queens cannot be placed considering the diagonal attacking
positions
;; of the queens already placed on the chessboard. It computes the possible location where a
queen
;; can be placed safely in the current row and returns this new row.
;;
(define (slash a b ca cb size)
  (if(equal? (length a) (length b))
      (slash-helper a b ca cb size)
      (slash-helper (append-with a ca size) (append-with b cb size) ca cb size)
  ))

;;
;; 'succ' function calculates the diagonally top-left to bottom-right attacking position for the
;; given chessboard. 'succ' operation involves adding of 0 to the end and removing one start
element
;; and shifting one position to the right.
;;
(define (my-succ row ca)
  (if(null? row)
      (cons ca '())
      (append (cdr row) (cons ca '())))
  ))

;;
;; 'pred' function calculates the diagonally top-right to bottom-left attacking position for the
;; given chessboard. 'pred' operation involves adding of 0 to the start and removing last element
and
;; shifting one position to the left.
;;
(define (my-pred row ca)
  (if(null? row)
      (cons ca '())
      (remove-last (append (cons ca '()) row) '()))
  ))

;;
;; union-helper helper routine for my-union operation
;;
(define (my-union-helper a b ca cb size)
  (if(null? a)
      '()
      (cons (or (car a) (car b)) (my-union-helper (cdr a) (cdr b) ca cb size)))
  ))

;;
;; This function makes a union of the previous impossible positions of queen with the current
;; placing of queen and returns a row where a queen cannot be placed.
;;
(define (my-union a b ca cb size)
  (if(equal? (length a) (length b))
      (my-union-helper a b ca cb size)
      (my-union-helper (append-with a ca size) (append-with b cb size) ca cb size)
  ))

;;
;; This is the main recursive translation of the C program. This function is recursive and uses
;; backtracking in order to compute the chessboard solutions. The conditions determines, whether
the
;; control returns from a function call or backtracks. The algorithm places a queen at the start
row

```

```

;; and recurs to find the position of the queen in the next row. The recursion looks at this as
new
;; problem of its own and tries to find a position where a queen can be placed. If a position is
;; found then it again recurs to find the position of next queen. And if a position is not found
then
;; it backtracks. Then the control is passed back to the parent function from where a new
position
;; for the queen is searched using the original algorithm that is it again recurs. This process
;; continues till the time the algorithm sees all the possible solutions for the given
chessboard.
;;
(define (F v a b c ls size lls e)
  (cond
    [(eq? v 't)
     (if (memq #t a)
         (F 'w a b c ls size '()) (slash (slash a b #f #f
size) c #f #f size))
         (list ls) )])
    [(eq? v 'w)
     (if (memq #t e)
         (F 'w a b c ls size
           (append lls (F
size)
                    (slash a (min-elt e) #f #f
(my-succ (my-union b (min-elt e) #f
#f size) #f)
(my-pred (my-union c (min-elt e) #f
#f size) #f)
(append (list (append-with (min-elt
e) #f size)) ls)
                    size
                    lls
                    e
                    ))
           (slash e (min-elt e) #f #f size))
         lls)]
  ))

;;
;; wrapper for the F method, this returns the solutions of a N Queen problem
;;
(define (queen size)
  (F
   't
   (append-with '() #t size)
   (append-with '() #f size)
   (append-with '() #f size)
   '()
   size
   '()
   '()
  ))

;;
;; counts the number of queens placed on the current row
;;
(define (count row ca val)
  (if(null? row)
      val
      (if(eq? (car row) ca)
          (count (cdr row) ca (+ val 1))
          (count (cdr row) ca val))))

;;
;; finds a value in a row
;;
(define (find-list ls a val)
  (if(null? ls)
      -1

```

```

    (if(eq? (car ls) a)
      val
      (find-list (cdr ls) a (+ val 1))))

;;
;; verifier-helper helper routine for verifier. It accepts a board checks if 2 queens are places
;; horizontally, vertically and diagonally. If yes returns true, false otherwise.
;;
(define (verifier-helper board i j k)
  (if(< i k)
    (if(eq? 1 (count (list-ref board i) #t 0))
      (if(< j i)
        (if(not (eq? (list-ref board i) (list-ref board j)))
          (if(not (eq? (abs (- i j)) (abs (- (find-list (list-ref board i) #t 0) (find-
list (list-ref board j) #t 0))))
            (verifier-helper board i (+ j 1) k)
            #f)
          #f)
        (verifier-helper board (+ i 1) 0 k))
      #f)
    #t))

;;
;; verifier routine, checks if each board configuration in the solution list is a correct
solution
;; for the N Queens problem. I uses the verifier-helper routine to test each board configuration.
;;
(define (verifier sol size)
  (if(null? sol)
    #t
    (if(eq? #t (verifier-helper (car sol) 0 0 size))
      (verifier (cdr sol) size)
      #f)))

;;
;; Following is an implementation of Permutation generation algorithm that we referred
;; we use this algorithm in order to generate possible permutations of rows which have
;; one queen placed in each column, thus when we consider all permutations of these rows
;; we get the most possible board configurations which we use for testing completeness of
;; the solutions.
;;
;; The original implementation has been modified to meet our requirements and for Chez Scheme.
;; URL : http://paste.lisp.org/display/11447
;;

(define (add-el-ls elem lists)
  (cond
    [(null? lists) '()]
    [else
     (cons
      (cons elem (car lists))
      (add-el-ls elem (cdr lists)))]))

;;
;; gen-perm accepts a list of objects and returns a list
;; of all permutations of these objects
;; e.g. (gen-perm 'g '() '(1 2 3))
;;      ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
;;
(define (gen-perm v done alist)
  (if (eq? v 'g)
    (cond
      [(null? alist) (list '())]
      [(null? (cdr alist)) (list alist)]
      [else
       (gen-perm 'u '() alist)])
    (if(eq? v 'u)
      (cond

```

```

        [(null? alist) '()]
      [else
       (append
        (add-el-ls
         (car alist)
         (gen-perm 'g '())
         (append done (cdr alist))))
       (gen-perm 'u
        (append done (list (car alist))
        (cdr alist)))]
    )))

;;
;; returns a row placing queen in position pos
;;
(define (get-row i pos size ls)
  (if(< i size)
    (if(eq? i pos)
      (get-row (+ i 1) pos size (cons #t ls))
      (get-row (+ i 1) pos size (cons #f ls)))
    ls))

;;
;; returns a list of rows with one queen placed in each possible column
;;
(define (gen-combo i size ls)
  (if(< i size)
    (gen-combo (+ i 1) size (append (list (get-row 0 i size '())) ls))
    ls
  ))

;;
;; accepts a list of chessboard configurations and verifies each.
;; if a board is correct solution, it adds to the output list.
;;
(define (comp-ver size ls lls)
  (if(null? ls)
    lls
    (if (eq? #t (verifier-helper (car ls) 0 0 size))
      (comp-ver size (cdr ls) (cons (car ls) lls))
      (comp-ver size (cdr ls) lls)
    )))

;;
;; returns a complete list of verified chessboard configuration that
;; are solutions selected/verified out of a huge number of most possible configurations
;; generated using gen-perm.
;;
(define (gen-complete size)
  (gen-perm 'g '() (gen-combo 0 size '())))

;;
;; returns a complete list of possible chessboard configurations which
;; are solutions for the N Queen problem
;;
(define (complete size)
  (comp-ver size (gen-complete size) '()))

;;
;; Test & Debug :
;;
(define print (lambda (b a) (printf "\n~a: ~a" b a)))

(define BOARDSIZE 5)
(define res (queen BOARDSIZE))
(print "Solution" res)
(print "Length" (length res))
(print "Verification" (verifier res BOARDSIZE))
(print "Complete" (complete BOARDSIZE))

```



## Queen.pvs

```
%%%
%%% Theories are parameterized by SIZE > 0.
%%% A board is a SIZE x SIZE array of truth values, represented as:
%%%   range -> [range -> bool] where range = [1..SIZE]
%%%
Board[SIZE:posint]: THEORY
BEGIN
  range:      TYPE = { k: int | 1 <= k AND k <= SIZE }
  row_t:      TYPE = ARRAY [range -> bool]
  board_t:    TYPE = ARRAY [range -> row_t]
  solution_t: TYPE = list[board_t]

  %%%
  %%% constructors from Theory Algorithm
  %%%
  %%%

  getEmptyRow(b:bool):row_t = (LAMBDA (c:range): b)
  EmptyRow:row_t = (LAMBDA (c:range): FALSE)
  FullRow:row_t = (LAMBDA (c:range): TRUE)

  getEmptyBoard(b:bool):board_t = (lambda (r:range): getEmptyRow(b))
  EmptyBoard:board_t = (lambda (r:range): EmptyRow)

  %%%
  %%% Elementary propositions
  %%%

  Board_prop_1: PROPOSITION EmptyRow = getEmptyRow(FALSE)
  Board_prop_2: PROPOSITION EmptyBoard = getEmptyBoard(FALSE)

  %%%
  %%% test from Theory Algorithm
  %%%
  %%% NOTES:
  %%% [1] All applications of isMember start a column 0 (now 1), so
  %%%     so I eliminate argument "count"
  %%% [2] Since positions are booleans, I eliminate the "val" argument
  %%%     in favor of just returning the truth value. So, for example
  %%%
  %%%     isMember(0, true, row) --> isMember(row)
  %%%     and
  %%%     isMember(0, false, row) --> NOT isMember(row)
  %%%

  isMember(row:row_t):bool = EXISTS (c:range): row(c)

  %%%
  %%% More primitives like setBool, etc., might be moved into this theory
  %%% so that more representation independence is achieved.
  %%%

END Board

Algorithm[SIZE:posint]: THEORY
BEGIN

IMPORTING Board[SIZE]

%%%
%%% fix some variable types to reduce clutter
%%%

b, b1, b2: VAR bool
r, r1, r2: VAR range
c, c1, c2: VAR range
row:      VAR row_t
```

```

brd:      VAR board_t

%%%
%%% These operations might be added to Theory Board, above.
%%%

setBool_Row(row, r):row_t = row WITH [(r) := TRUE]
clrBool_Row(row, r):row_t = row WITH [(r) := FALSE]
setRow_Board(brd, r, row):board_t = brd WITH [(r) := row]

%%%
%%% These methods replace a given row the empty row with smallest index.
%%%
appendRow_Board(brd, row, r): RECURSIVE board_t =
  IF r = SIZE
  THEN EmptyBoard
  ELSIF isMember(brd(r))
  THEN appendRow_Board(brd, row, r+1)
  ELSE setRow_Board(brd, r, row)
  ENDIF
  MEASURE SIZE - r

%%%
%%% This appendRow_Board wrapper
%%%
appendRow_Board(brd, row): board_t = appendRow_Board(brd, row, 1)

%%%
%%% This function accepts 2 rows. First row represents a row in which a queen can be placed
%%% considering the vertical arrangements of the queens already placed on the chessboard while
%%% second row is the row in which the queens cannot be placed considering the diagonal attacking
%%% positions of the queens already placed on the chessboard. It computes the possible location
%%% where a queen can be placed safely in the current row and returns this new row.
%%%
slash(row1, row2: row_t):row_t = (lambda (r): row1(r) AND NOT row2(r))

%%%
%%% Returns the union of 2 rows. row-1 a row where queens cannot be placed and row-2 where a
queen is placed in the
%%% current pass.
union(row1, row2: row_t):row_t = (lambda (r): row1(r) OR row2(r))

%%%
%%% This accepts 'row' and returns an row_t such that it has the smallest element of 'row'.
%%% This computes the position of queen amongst available positions. Thus only one queen is
%%% placed in one pass and as the output 'row' contains only 1, thus, only one queen can be
%%% placed on each row.
%%%
min_elt(row):row_t =
  (lambda (r:range): (row(r) AND NOT(EXISTS(i: int|i >= 1 AND i < r):row(i))))

%%%
%%% succ is called just once, in F. In that instance "val" is FALSE
%%% suggesting that argument "b" could be subsumed in the definition.
%%%

succ(row, b): row_t =
  (LAMBDA (c): IF c = SIZE THEN b ELSE row(c+1) ENDIF)

pred(row, b): row_t =
  (LAMBDA (c): IF c = 1 THEN b ELSE row(c-1) ENDIF)

%%%
%%% ADDED functions
%%%
%%% It seems like it would be useful to know the index
%%% of the first empty row in a board.
%%%
%%% NOTE: alternatively, these could be defined using FORALL/EXISTS

```

```

%%%

first_empty_row(brd, r): RECURSIVE range =
  IF r = SIZE
    THEN SIZE
  ELSIF isMember(brd(r))
    THEN first_empty_row(brd, r+1)
  ELSE r
  ENDIF
  MEASURE SIZE - r

first_empty_row(brd):range = first_empty_row(brd, 1)

%%%
%%% COMMENT: If I understand appendRow_Board correctly, it could be
%%% written this way in terms of first_empty_row:
%%%
alt_appendRow_Board(brd, row): board_t = brd WITH [(first_empty_row(brd)):= row]
%%%

%%%
%%% Function F with argument "size" removed.
%%%

F(v: nat, a_row, b_row, c_row:row_t, brd:board_t, sol:solution_t, e_row: row_t):
  RECURSIVE solution_t =
  IF v = 1
    THEN IF isMember(a_row)
      THEN F(2, a_row, b_row, c_row, brd, null, slash(slash(a_row, b_row), c_row))
      ELSE cons(brd, null)
    ENDIF
  ELSIF isMember(e_row)
    THEN F( 2
      , a_row
      , b_row
      , c_row
      , brd
      , append(sol,
        F( 1
          , slash(a_row, min_elt(e_row))
          , succ(union(b_row, min_elt(e_row))), FALSE)
          , pred(union(c_row, min_elt(e_row))), FALSE)
          , appendRow_Board(brd, min_elt(e_row), 1)
          , sol
          , e_row
        )
      , slash(e_row, min_elt(e_row))
    )
  ELSE sol
  ENDIF
  MEASURE exp2(SIZE + 1) - length(sol)

%%%
%%% queen function that produces solution to the N Queens problem
%%%

queen: solution_t =
  F( 1
    , FullRow
    , EmptyRow
    , EmptyRow
    , EmptyBoard
    , null
    , EmptyRow
  )

END Algorithm

Verification[SIZE:posint]: THEORY
BEGIN

```

```

IMPORTING Board[5]
IMPORTING Algorithm[5]

%%%
%%% Count returns the number of queen's places on the given row
%%%

count(row:row_t, i, k:int): RECURSIVE int =
    IF i <= SIZE THEN
        IF row(i) THEN count(row, (i + 1), (k + 1)) ELSE count(row, (i + 1), k) ENDIF
    ELSE k ENDIF
    MEASURE SIZE - i

%%%
%%% getPositions returns the index where the queen is placed in give row
%%%

getPosition(row:row_t, k:int): RECURSIVE int =
    IF k <= SIZE THEN
        IF row(k) THEN k ELSE getPosition(row, k + 1) ENDIF
    ELSE
        0
    ENDIF
    MEASURE SIZE - k

%%%
%%% horz? returns true if there is only one queen placed on each row, false otherwise
%%%

horz?(brd:board_t):bool =
    (FORALL (b:range): (count(brd(b), 1, 0) = 1))

%%%
%%% vert? returns true if there is only one queen in each column, false otherwise
%%%

vert?(brd:board_t):bool =
    horz?(transpose(brd))

%%%
%%% diag returns true if there are no 2 queens diagonally aligned, false otherwise
%%%

diag?(brd:board_t):bool =
    (FORALL (i:range):
        (FORALL (j: int | 1 <= j AND j < i):
            NOT (abs(i - j) = abs(getPosition(brd(i), 1) - getPosition(brd(j), 1))))))

%%%
%%% solution? returns true if the given board configuration meets the requirements
%%% i.e. no 2 qeeuns are in attacking position and is an solution to N Qeeun problem.
%%%

solution?(brd:board_t): bool = horz?(brd) AND vert?(brd) AND diag?(brd)

%%%
%%% solutions? returns true if the given set of board configurations meet the requirements
%%% i.e. no 2 qeeuns are in attacking position and is an solution to N Qeeun problem.
%%%

solutions?(soln:solution_t): bool = every(solution?) (soln)

%%%
%%% correct? checks for correctness of the solutions. It returns true if the given set of board
%%% configurations meet the requirements i.e. no 2 queens are in attacking position and is an
%%% solution to N Queen problem and the number of solutions produces are same as the possible
%%% solutions for the given 'N'
%%%
%%%
correct?(soln:solution_t, len:int):bool = solutions?(soln) AND length(soln) = len

```

```

%%%
%%% factorial of number n
%%%

factorial(i,k:nonneg_int): RECURSIVE int =
  IF i > 1 THEN
    factorial(i - 1, k * i)
  ELSE
    k
  ENDIF
  MEASURE i

%%%
%%% addRB appends a board configuration to solutions list if it meets the requirements and
%%% is a solution for N Queen problem
%%%

addRB(row:row_t, ls:solution_t): RECURSIVE solution_t=
  CASES ls OF
    null : null,
    cons(a, tl) :
      append(
        cons(appendRow_Board(a, row), null),
        addRB(row, tl))
  ENDCASES
  MEASURE ls BY <<

%%%
%%% genPerm generates list of possible (partial) board configurations of how N queens can be
%%% placed on a chessboard.
%%%

genPerm(i, j:int, orig: board_t): RECURSIVE solution_t =
  IF j <= SIZE THEN
    append(
      addRB(
        orig(j),
        genPerm(
          1,
          j + 1,
          orig
        )
      ),
      genPerm(
        i + 1,
        j - 1,
        orig
      )
    )
  ELSE
    null
  ENDIF
  MEASURE SIZE - j

%%%
%%% genCombo returns a list or rows with one Queen placed in each column.
%%%

genCombo(i:int, brd:board_t): RECURSIVE board_t =
  IF i <= SIZE THEN
    genCombo(i + 1, brd WITH [(i) := setBool_Row(EmptyRow, i)])
  ELSE
    brd
  ENDIF
  MEASURE SIZE - i

%%%
%%% genAllPerm returns a list of possible chessboard configurations
%%%

```

```

genAllPerm:solution_t =
    genPerm(1, 1, genCombo(1, EmptyBoard))

%%%
%%% verifierComplete verifies each provided chessboard configuration, it is a solution it adds
%%% it to the output solution list.
%%%

verifierComplete(sol:solution_t): RECURSIVE solution_t =
    CASES sol OF
        null : null,
        cons(a, t1) :
            IF solution?(a) THEN
                append(cons(a, null), verifierComplete(t1))
            ELSE
                verifierComplete(t1)
            ENDIF
    ENDCASES
    MEASURE sol BY <<

%%%
%%% getComplete returns a verified solution list of chessboard
%%%

getComplete:solution_t =
    verifierComplete(genAllPerm)

%%%
%%% contains checks if a board configuration is present in the solution list provided.
%%%

contains(brd:board_t, sol:solution_t): RECURSIVE bool =
    CASES sol OF
        null : FALSE,
        cons(a, t1):
            brd = a AND contains(brd, t1)
    ENDCASES
    MEASURE sol BY <<

%%%
%%% compare_h helper compares if 2 solution lists are same by checking if both lists
%%% contain same list of board configurations.
%%%

compare_h(sol1, sol2:solution_t):RECURSIVE bool=
    CASES sol1 OF
        null : TRUE,
        cons(a, t1) :
            contains(a, sol2) AND compare_h(t1, sol2)
    ENDCASES
    MEASURE sol1 BY <<

%%%
%%% compare compares if 2 solution lists are same by checking if both lists
%%% contain same list of board configurations, have same number of board configurations
%%% and have symmetric solutions.
%%%

compare(sol1, sol2:solution_t):bool =
    length(sol1) = length(sol2) &
    compare_h(sol1, sol2) &
    compare_h(sol2, sol1)

%%%
%%% complete tests if ths solution is complete
%%%

complete?(sol:solution_t):bool =
    compare(sol, getComplete)

```

```

%%%
%%% prover checks if the solution is complete and correct
%%%

prover: THEOREM correct?(queen, 2) AND complete?(queen)

END Verification

Unit_Test: THEORY
BEGIN

IMPORTING Board[5]
IMPORTING Algorithm[5]
IMPORTING Verification[5]

a1: LEMMA isMember(FullRow)

a2: LEMMA isMember(setBool_Row(EmptyRow, 3))

a3: LEMMA min_elt(setBool_Row(EmptyRow, 2)) = setBool_Row(EmptyRow, 2)

a4: LEMMA min_elt(setBool_Row(setBool_Row(EmptyRow, 2), 3))
    = setBool_Row(EmptyRow, 2)

a5: LEMMA appendRow_Board( EmptyBoard
                          , setBool_Row(EmptyRow, 3)
                          )
    = setRow_Board( EmptyBoard
                  , 1
                  , setBool_Row(EmptyRow, 3)
                  )

a6: LEMMA appendRow_Board( setRow_Board(EmptyBoard, 1, setBool_Row(EmptyRow, 2))
                          , setBool_Row(EmptyRow, 3)
                          )
    = setRow_Board( setRow_Board(EmptyBoard, 1, setBool_Row(EmptyRow, 2))
                  , 2
                  , setBool_Row(EmptyRow, 3)
                  )

%a7: LEMMA
v_diag1: LEMMA diag?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 3),
(3) := setBool_Row(EmptyRow, 5),
(4) := setBool_Row(EmptyRow, 2),
(5) := setBool_Row(EmptyRow, 4)])

v_diag2: LEMMA NOT diag?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 3),
(3) := setBool_Row(EmptyRow, 5),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 2)])

v_diag3: LEMMA NOT diag?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 3),
(3) := setBool_Row(EmptyRow, 4),
(4) := setBool_Row(EmptyRow, 2),
(5) := setBool_Row(EmptyRow, 5)])

v_trans: LEMMA transpose(transpose((EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5)]))) =
(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),

```

```

(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5])

v_horz1: LEMMA vert?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5)])

v_horz2: LEMMA NOT vert?(EmptyBoard WITH
[(1) := setBool_Row(setBool_Row(EmptyRow, 1), 4),
(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5)])

v_vert1: LEMMA NOT horz?(EmptyBoard)

v_vert2: LEMMA horz?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5)])

v_vert3: LEMMA NOT horz?(EmptyBoard WITH
[(1) := setBool_Row(setBool_Row(EmptyRow, 1), 2),
(2) := setBool_Row(EmptyRow, 2),
(3) := setBool_Row(EmptyRow, 3),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 5)])

a_slash1: LEMMA slash(setBool_Row(setBool_Row(EmptyRow, 1), 4), setBool_Row(setBool_Row(EmptyRow,
1), 3)) = setBool_Row(EmptyRow, 4)

a_union1: LEMMA union(setBool_Row(setBool_Row(EmptyRow, 1), 4), setBool_Row(setBool_Row(EmptyRow,
1), 3)) = setBool_Row(setBool_Row(setBool_Row(EmptyRow, 4), 3), 1)

a_succ1: LEMMA succ(setBool_Row(setBool_Row(EmptyRow, 2), 4), FALSE) =
setBool_Row(setBool_Row(EmptyRow, 1), 3)

a_pred1: LEMMA pred(setBool_Row(setBool_Row(EmptyRow, 2), 4), FALSE) =
setBool_Row(setBool_Row(EmptyRow, 5), 3)

v_count: LEMMA count(setBool_Row(setBool_Row(EmptyRow, 2), 4), 1, 0) = 2

v_getPosition: Lemma getPosition(setBool_Row(EmptyRow, 2), 0) = 2

v_solution?1: LEMMA solution?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 3),
(3) := setBool_Row(EmptyRow, 5),
(4) := setBool_Row(EmptyRow, 2),
(5) := setBool_Row(EmptyRow, 4)])

v_solution?2: LEMMA NOT solution?(EmptyBoard WITH
[(1) := setBool_Row(EmptyRow, 1),
(2) := setBool_Row(EmptyRow, 3),
(3) := setBool_Row(EmptyRow, 5),
(4) := setBool_Row(EmptyRow, 4),
(5) := setBool_Row(EmptyRow, 2)])

END Unit_Test

```

## Queen.status

### Proof summary for theory Board

```
Board_prop_1.....proved - complete [shostak] (0.03 s)
Board_prop_2.....proved - complete [shostak] (0.01 s)
Theory totals: 2 formulas, 2 attempted, 2 succeeded (0.04 s)
```

### Proof summary for theory Algorithm

```
appendRow_Board_TCC1.....proved - complete [shostak] (0.01 s)
appendRow_Board_TCC2.....proved - complete [shostak] (0.02 s)
appendRow_Board_TCC3.....proved - complete [shostak] (0.04 s)
appendRow_Board_TCC4.....proved - complete [shostak] (0.00 s)
min_elt_TCC1.....proved - complete [shostak] (0.01 s)
succ_TCC1.....proved - complete [shostak] (0.02 s)
pred_TCC1.....proved - complete [shostak] (0.02 s)
first_empty_row_TCC1.....proved - complete [shostak] (0.00 s)
F_TCC1.....unfinished [shostak] (0.01 s)
F_TCC2.....unfinished [shostak] (0.12 s)
F_TCC3.....unfinished [shostak] (0.05 s)
F_TCC4.....unfinished [shostak] (0.03 s)
Theory totals: 12 formulas, 12 attempted, 8 succeeded (0.33 s)
```

### Proof summary for theory Verification

```
count_TCC1.....unfinished [shostak] ( 0.02 s)
count_TCC2.....unfinished [shostak] ( 0.11 s)
count_TCC3.....proved - complete [shostak] ( 0.00 s)
count_TCC4.....proved - complete [shostak] ( 0.02 s)
diag?_TCC1.....proved - complete [shostak] ( 0.00 s)
factorial_TCC1.....proved - complete [shostak] ( 0.02 s)
factorial_TCC2.....proved - complete [shostak] ( 0.00 s)
addRB_TCC1.....proved - complete [shostak] ( 0.02 s)
genPerm_TCC1.....proved - complete [shostak] ( 0.01 s)
genPerm_TCC2.....unfinished [shostak] ( 0.03 s)
prover.....unfinished [shostak] (12.54 s)
Theory totals: 11 formulas, 11 attempted, 7 succeeded (12.77 s)
```

### Proof summary for theory Unit\_Test

```
a1.....proved - complete [shostak] (0.01 s)
a2.....proved - complete [shostak] (0.01 s)
a3.....proved - complete [shostak] (0.06 s)
a4.....proved - complete [shostak] (0.06 s)
a5.....proved - complete [shostak] (0.03 s)
a6.....proved - complete [shostak] (0.09 s)
v_diag1.....proved - incomplete [shostak] (2.34 s)
v_diag2.....proved - incomplete [shostak] (1.36 s)
v_diag3.....proved - incomplete [shostak] (1.29 s)
v_trans.....proved - complete [shostak] (0.10 s)
v_horz1.....proved - incomplete [shostak] (0.92 s)
v_horz2.....proved - incomplete [shostak] (0.11 s)
v_vert1.....proved - incomplete [shostak] (0.05 s)
v_vert2.....proved - incomplete [shostak] (1.76 s)
v_vert3.....proved - incomplete [shostak] (1.53 s)
a_slash1.....proved - complete [shostak] (0.06 s)
a_union1.....proved - complete [shostak] (0.06 s)
a_succ1.....proved - complete [shostak] (0.17 s)
a_pred1.....proved - complete [shostak] (0.13 s)
v_count.....proved - incomplete [shostak] (0.02 s)
v_getPosition.....proved - incomplete [shostak] (0.01 s)
v_solution?1.....proved - incomplete [shostak] (5.12 s)
v_solution?2.....proved - incomplete [shostak] (1.60 s)
Theory totals: 23 formulas, 23 attempted, 23 succeeded (16.89 s)
```

Grand Totals: 48 proofs, 48 attempted, 40 succeeded (30.03 s)