A formal system for automated reasoning about retrograde chess problems using Coq

Marko Maliković
Faculty of Arts and Sciences
University of Rijeka
Omladinska 14, 51000 Rijeka, Croatia
marko.malikovic@ffri.hr

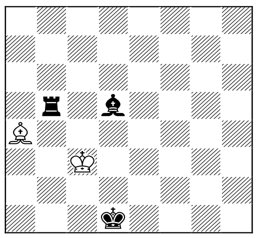
Abstract*. This paper presents a formal system for automated reasoning about retrograde chess problems using Coq – a formal proof management system. The system is divided into two parts. The first part describes the environment that includes the axioms, definitions and hypotheses of chess objects, and also the functions for computing changes in states. The second part is developed for generating possible retrograde chess moves and includes Coq's tactics combined with the use of tacticals (elements of Ltac - the Coq's language for combining tactics). All of these tactics are defined as one Ltac function. This approach enables reasoning about retrograde chess problems with respect to reasoning about sequences of retrograde moves. In the aforementioned Ltac function, a number of heuristic solutions are implemented with the aim of solving the problems within a big search space such as retrograde chess analysis. These heuristics, as well as tactics and tacticals, are not the subject of this article.*

Keywords. retrograde chess analysis, formal system, Coq, automated reasoning

1 Introduction

Retrograde chess analysis is a method employed by chess problem solvers to determine which moves were played leading up to a given position. These moves are called the *history of the position*. Also, using retrograde chess analysis sometimes it is possible to determine if castling is disallowed, whether an *en passant* capture is possible or if a particular position is legal. Retrograde analysis is essentially a matter of logical reasoning as we can see in the example shown on diagram 1. The solver must deduce what were the last three moves.

Diagram 1. What were the last 3 moves?

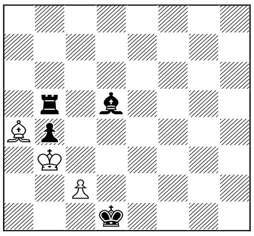


Black king is in check but the bishop making the check cannot have made the last checking move. Therefore the white king must have moved off *b3* to discover the check. On *b3* the white king was in check by both the rook and the bishop which is, at first thought, impossible. But, if white had a pawn on *c2* and black had a pawn on *b4*, then white blocked the bishop's check by *c2c4* and after black's *b4xc3 en passant*, white is in double check. Since the black pawn is no longer on the board, white must have captured it in the last move. So, the solution to this problem is *1. c2c4, b4c3ep 2. Kb3c3x* and the position before three moves is shown in diagram 1a.

We define a retrograde chess move as follows:

Definition (*retrograde chess move*). If in accordance with the laws of chess [4] position P_{n+1} arises from position P_n due to the move p of piece f then the retrograde chess move p' of move p is the *movement* of piece f due to the position P_n arising from position P_{n+1}.

Diagram 1a. The position three moves before the position in diagram 1



In accordance with the above mentioned definition we can set up the rules of retrograde chess moves into two main groups: those rules which describe the retrograde moves of each chess piece and common rules for all moves.

In this article we will not present all of the rules for each piece because they clearly arise from the definition of retrograde chess moves. We give just one example: The retrograde move of the king is the movement from the starting square to one of the eight closest squares in the same row, column or diagonal with the following conditions: the end square is not near the opponent's king and any retrograde castling has not been yet played with the moved king.

Common rules for all retrograde moves are:

I. To make a move it must be the player's turn
II. After retrograde *en passant* capture, two squares backwards by the retrograde captured pawn must be played
III. The end square of the retrograde move must be empty
IV. Some of the opponent's pieces can appear[1] on the starting square or the square can remain empty, but the following conditions must be satisfied:
   a) The pawn can't appear on the first or last row of the chessboard
   b) In cases of retrograde capture by the pawn, the starting square can't remain empty
   c) In those situations of retrograde moves by the pawn without retrograde capturing (one or two squares backwards), any retrograde castling and any retrograde *en passant* capturing, the starting square must remain empty
   d) In case of retrograde promotion with retrograde capturing, the starting square can't remain empty
V. After some retrograde moves, the opponent's king may not be in check
VI. After the move no players can have more than eight pawns or more than sixteen pieces

The *Coq* system[2] is a computer tool for verifying theorem proofs in higher-order logic. These theorems may concern usual mathematics, proof theory, or program verification. The underlying theory of the *Coq* system is *Calculus of Inductive Constructions*, a formalism that combines logic from the point of view of λ-calculus and typing. *Objective Caml*[3] is the implementation language for the *Coq*.

Concerning a proposition that one wants to prove, the *Coq* system proposes tools, called *tactics*, to construct a proof, using elements taken from a context, namely, declarations, definitions, axioms, hypotheses, lemmas, and already proven theorems. In addition, the *Coq* system provides operators, called *tacticals*, that make it possible to combine tactics and, in such manner, to build more complex tactics. This paper presents a formal system for automated reasoning about retrograde chess problems using Coq.

2 Definitions

2.1 Definitions of the chess pieces, the colors of pieces, types of retrograde chess moves and the chessboard

We define the chess pieces, the colors of pieces and types of retrograde chess moves as enumerated inductive type [2, 137]:

```
Inductive pieces:Set:=P | B | R | Q | N | K | p | b | r | q | n | k | O | v.
```

Constructors of above mentioned enumerated inductive type *pieces* and its meanings are shown in table 1.

The definition of the colors of pieces has got only two constructors:

```
Inductive colors : Set := white | black.
```

[1] We call this *retrograde capturing*.

[2] http://coq.inria.fr

[3] http://caml.inria.fr

Table 1: Constructors of enumerated inductive type *pieces* and its meanings

Cons.	Piece	Cons.	Piece	Cons.	Piece
P	♙	p	♟	O	empty square
B	♗	b	♝		
R	♖	r	♜	v	area outside the board
Q	♕	q	♛		
N	♘	n	♞		
K	♔	k	♚		

We group the types of moves in accordance with their properties:

```
Inductive type_of_move : Set :=
standard_move
| promotion
| promotion_cap_3
| promotion_cap_4
| castling_kingside_white
| castling_queenside_white
| castling_kingside_black
| castling_queenside_black
| p_1
| p_2
| p_cap_3
| p_cap_4
| p_ep_cap_5
| p_ep_cap_6.
```

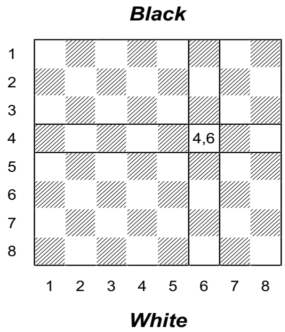
The meanings of the above constructors are:

- *promotion* - retrograde promotion
- *promotion_cap_3*, *promotion_cap_4* - different retrograde promotions with capturing of an opponent's piece (according to the direction of the pawn's move)
- *castling_kingside_white* - retrograde white's kingside castling
- *castling_queenside_white* - retrograde white's queenside castling
- *castling_kingside_black* - retrograde black's kingside castling
- *castling_queenside_black* - retrograde black's queenside castling
- *p_1* - one square backwards by the pawn
- *p_2* - two squares backwards by the pawn
- *p_cap_3*, *p_cap_4* - two different retrograde captures by the pawn (according to the direction of the pawn's move)
- *p_ep_cap_5*, *p_ep_cap_6* - two different retrograde *en passant* captures (according to the direction of the pawn's move)
- *standard_move* - all other retrograde moves

On the other hand, we introduce both coordinates of the squares (rows and columns) as just one annotated inductive type [8, 39]:

```
Inductive coordinates : nat -> Prop :=
| coord1 : coordinates 1
| coord2 : coordinates 2
| coord3 : coordinates 3
| coord4 : coordinates 4
| coord5 : coordinates 5
| coord6 : coordinates 6
| coord7 : coordinates 7
| coord8 : coordinates 8.
```

Diagram 2. The coordinates of the squares due to the orientation of chessboard



2.2 Definition of chess position

The player's color, whose turn it is, we declare in a retrograde sense (who moved the last piece) as the following declaration of global parameter *on_turn*:

```
Parameter on_turn : nat -> colors.
```

This parameter has type *nat -> colors* where *nat* is a type of current ordinal number of the move. Also, we introduce hypothesis *Hon* about the value of this number (at the beginning of reasoning about a retrograde chess problem, the value of this number is zero):

```
Parameter on : nat.
Hypothesis Hon : on=0.
```

The position of pieces on the board in the moment *on* is a list of lists and we introduce it as hypothesis (first we need to declare parameter *position* with type *nat -> list (list pieces)*):[4]

```
Parameter position : nat -> list (list pieces).
```

[4] All hypotheses at the end of these sections correspond to problem 5 which will be considered in Section 4.3.

```
Variable H_position : position on =
(v :: nil) ::
(v :: k :: O :: K :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: Q :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: B :: O :: P :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: P :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: P :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: O :: nil) ::
(v :: O :: O :: O :: O :: O :: O :: O :: B :: nil) ::
nil.
```

In the given position the black king is in check. This is because we can conclude that white made the last move and we introduce this fact as a hypothesis:

```
Hypothesis H_on_turn : on_turn on=white.
```

From the hypothesis *H_position* the coordinates of the white and black kings arise, as well as the number of white and black pawns and the total number of white and black pieces. This data is very important for the speed of our system and we store them in separate hypotheses in the following way:

```
Parameter xKw yKw xkb ykb : nat -> nat.

Hypothesis Hxkb : xkb on=1.
Hypothesis Hykb : ykb on=1.
Hypothesis HxKw : xKw on=1.
Hypothesis HyKw : yKw on=3.

Parameter
white_pawns_number
black_pawns_number
white_pieces_number
black_pieces_number : nat -> nat.

Hypothesis H_white_pawns_number:white_pawns_number on=3.
Hypothesis H_black_pawns_number:black_pawns_number on=0.
Hypothesis H_white_pieces_number:white_pieces_number on=7.
Hypothesis H_black_pieces_number:black_pieces_number on=1.
```

2.3 Definition of retrograde move

Each retrograde move is uniquely defined by several attributes. For example, every standard move is defined by the coordinates of the starting square, the coordinates of the end square, the opponent's retrograde captured piece and the type of move. On the other hand, the retrograde move *p_2* is only defined by the column of the moved pawn and the type of move.

Anyway, our definition of the retrograde move will contain the same arguments for all retrograde moves: the ordinal number of the move, the piece that is moved, the coordinates of the starting and the end squares, the captured piece and the type of move:

```
Inductive move : Set := moved : nat -> pieces -> nat -> nat -> nat -
> nat -> pieces -> type_of_move -> move.
```

The reason for such a wide definition lies in our need to have all the relevant information about the moves in a single hypothesis. As a result, our system becomes faster.

When solving problems we will thread the sequences of retrograde moves in the special hypothesis *H_list_moves*. For example, the fact that a given position arises after *1. Nb6a8, Ka7a8x* will be stored on the list as follows:

```
H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 b standard_move :: nil
```

2.4 Possible captured pieces

In rules IV and VI in Section 1 we outlined the conditions which must be satisfied concerning the content of the starting square after the retrograde move. These conditions will be later checked by tactics. In general, we can introduce the following annotated inductive types for possible white and black captured pieces:

```
Inductive possible_captured_pieces_white : pieces -> Prop :=
| cap_piece_w_b : possible_captured_pieces_white b
| cap_piece_w_r : possible_captured_pieces_white r
| cap_piece_w_q : possible_captured_pieces_white q
| cap_piece_w_n : possible_captured_pieces_white n
| cap_piece_w_p : possible_captured_pieces_white p
| cap_piece_w_O : possible_captured_pieces_white O.

Inductive possible_captured_pieces_black : pieces -> Prop :=
| cap_piece_b_B : possible_captured_pieces_black B
| cap_piece_b_R : possible_captured_pieces_black R
| cap_piece_b_Q : possible_captured_pieces_black Q
| cap_piece_b_N : possible_captured_pieces_black N
| cap_piece_b_P : possible_captured_pieces_black P
| cap_piece_b_O : possible_captured_pieces_black O.
```

3 Functions

3.1 End squares

In this section we introduce those functions that will determine whether a square can be the end square of a piece in a given position on the basis of the content of the relevant squares for that piece. We call such squares *possible end squares* because each retrograde move must also satisfy a number of other conditions (for example after the retrograde move the opponent's king may not be in check - see rule V in Section 1).

3.1.1 Standard moves by the bishop, rook and queen

The bishop, rook or queen functions are recursive. With these functions we will determine how many squares are empty, beginnings from the starting square in a number of possible directions.

First we introduce the possible directions of a move for the bishop, rook and queen, since in different directions, the coordinates of the possible end squares change in different ways (see diagram 3).

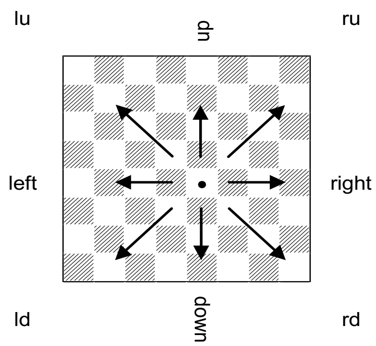
Thus, we introduce the following enumerated inductive types:

```
Inductive directions_bishop : Set := lu | ru | rd | ld.

Inductive directions_rook : Set := left | right | up | down.
```

The set of directions for the queen is actually a union of the directions for the bishop and rook. But, the constructors in two different inductive definitions must be different. This is because we need an extra definition for the queen:

```
Inductive directions_queen : Set := lu_queen | ru_queen |
rd_queen | ld_queen | left_queen | right_queen | up_queen |
down_queen.
```

Diagram 3. The possible directions of moves by the bishop, rook and queen



In this article we only provide the function that will determine whether a square can be the end square of the bishop:

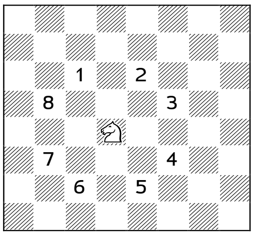
```
Parameter xp_var yp_var xz yz u : nat.

Fixpoint end_square_bishop_temp (xp_var yp_var : nat)
(s:directions_bishop) (l:list (list pieces)) (i:nat) {struct i} : Prop :=
match i with
S i' =>
match s with
lu =>
match xp_var with
S xp_var' =>
match yp_var with
S yp_var' => match nth yp_var' (nth xp_var' l nil) v with
            O => end_square_bishop_temp xp_var' yp_var' lu l i'
           | _ => u<8-i /\ xz=xp-u-1 /\ yz=yp-u-1
            end
| _ => False
end
| _ => False
end
| ru =>
match xp_var with
S xp_var' =>
match S (S yp_var) with
S yp_var' => match nth yp_var' (nth xp_var' l nil) v with
            O => end_square_bishop_temp xp_var' yp_var' ru l i'
           | _ => u<8-i /\ xz=xp-u-1 /\ yz=yp+u+1
            end
| _ => False
end
| _ => False
end
| rd =>
match S (S xp_var) with
S xp_var' =>
match S (S yp_var) with
S yp_var' => match nth yp_var' (nth xp_var' l nil) v with
            O => end_square_bishop_temp xp_var' yp_var' rd l i'
           | _ => u<8-i /\ xz=xp+u+1 /\ yz=yp+u+1
            end
| _ => False
end
| _ => False
end
| ld =>
match S (S xp_var) with
S xp_var' =>
match yp_var with
S yp_var' => match nth yp_var' (nth xp_var' l nil) v with
            O => end_square_bishop_temp xp_var' yp_var' ld l i'
           | _ => u<8-i /\ xz=xp+u+1 /\ yz=yp-u-1
            end
| _ => False
end
| _ => False
end
end
| _ => False
end.

Definition end_square_bishop xp_var yp_var s l :=
end_square_bishop_temp xp_var yp_var s l 8.
```

The functions for the rook and queen are analogous.

3.1.2 Standard moves by the knight and king, moves by the pawn and other non-standard moves

The functions which determine possible end squares for knight and king are not recursive. These functions simply check whether a square is empty or not. For these pieces we introduce numerical notations of the possible end squares (see the example for the knight in diagram 4).

Diagram 4. Possible end squares for the knight



So, the function for the knight is:

```
Definition end_square_knight (xp yp i:nat) (l : list (list figure)) :
Prop :=
 match i with
    1 => match nth (yp-1) (nth (xp-2) l nil) v with
           O => xz=xp-2 /\ yz=yp-1
         | _ => False
        end
  | 2 => match nth (yp+1) (nth (xp-2) l nil) v with
           O => xz=xp-2 /\ yz=yp+1
         | _ => False
        end
  | 3 => match nth (yp+2) (nth (xp-1) l nil) v with
           O => xz=xp-1 /\ yz=yp+2
         | _ => False
        end
  | 4 => match nth (yp+2) (nth (xp+1) l nil) v with
           O => xz=xp+1 /\ yz=yp+2
         | _ => False
        end
  | 5 => match nth (yp+1) (nth (xp+2) l nil) v with
           O => xz=xp+2 /\ yz=yp+1
         | _ => False
        end
  | 6 => match nth (yp-1) (nth (xp+2) l nil) v with
           O => xz=xp+2 /\ yz=yp-1
         | _ => False
        end
  | 7 => match nth (yp-2) (nth (xp+1) l nil) v with
           O => xz=xp+1 /\ yz=yp-2
         | _ => False
        end
  | 8 => match nth (yp-2) (nth (xp-1) l nil) v with
           O => xz=xp-1 /\ yz=yp-2
         | _ => False
        end
  |_ => False
 end.
```

For the moves by the pawn and other non-standard moves, the functions and methods for determining possible end squares are very diverse and will not be shown in this article.

3.2 Axioms about possible combinations of moves attributes

With a view to generating valid retrograde moves more easily, as well as the easier elimination of invalid retrograde moves, we introduce axioms about possible combinations of moves attributes according to piece and type of move. Here we give the axiom about standard move by the white bishop:

```
Axiom A_standard_move_information_B :
move_information B=
(tm=standard_move /\
direction_bishop=db /\
xp_var=xp /\
yp_var=yp /\
end_square_bishop xp_var yp_var db (position on) /\
possible_captured_pieces_white p_cap).
```

The above axiom contains data about the type of move, the direction, the end square and the captured opponent's piece. The axiom about, e.g. move *p_1* does not contain anything about the type of move, direction or the opponent's captured piece because none of this data exists for move *p_1*.

With the aim of applying the described axioms during reasoning about chess positions, we introduce the following hypothesis in which the axioms will be applied:

```
Parameter move_information : pieces -> Prop.
Hypothesis H_move_information : move_information (fp on).
```

3.3 Functions for computing new positions after a retrograde move

After every retrograde move a new position arises. Before we define the main function in order to compute the new position of pieces on the chessboard (list in variable *H_position*), we need to define several auxiliary functions.

The function *beginning_of_list* takes the *n* first rows of a list:

```
Fixpoint beginning_of_list (n : nat) (l : list (list pieces)) {struct n} :
list (list pieces) :=
 match l with
    nil => nil
  | l' :: l1 => match n with
              0 => nil
            | S n1 => l' :: beginning_of_list n1 l1
           end
 end.
```

The function *rest_of_list* takes the rows from *m*-th to the end of list (including *m*-th row):

```
Fixpoint rest_of_list (m : nat) (l : list (list pieces)) {struct m}: list (list
pieces) :=
 match l with
    nil => nil
  | l' :: l1 => match m with
              0 => l
            | S n1 => rest_of_list n1 l1 end
      end.
```

The function *change_of_piece* for the given linear list of pieces returns the list with changed *n*-th element by piece *f*:

```
Fixpoint change_of_piece (n : nat) (f : pieces) (l : list pieces) {struct
n} : list pieces :=
      match l with
         nil => nil
      | l' :: l1 => match n with
                     0 => f :: l1
                   | S n1 => l' :: change_of_piece n1 f l1
                  end
    end.
```

And finally we need a function which for the given position returns a new position with the changed piece in place *(xp,yp)* of the new piece *fo* (*fo* is the captured piece):

```
Definition position_xp_yp (xp yp : nat) (l : list (list pieces)) (fo :
pieces) := app (app (beginning_of_list xp l) ((change_of_piece yp
fo (nth xp l nil)) :: nil)) (rest_of_list (xp+1) l).
```

The main function *position_new* will compute the new position of pieces on the board depending on the type of move and is based on a pattern matching over hypotheses about types of move. For example, if the *standard_move* is matched in the context then the function will be the result of the following term:[5]

```
app (app (beginning_of_list xk (position_xp_yp xp yp l fo))
((change_of_piece yk fp (nth xk (position_xp_yp xp yp l fo) nil)) ::
nil)) (rest_of_list (xk+1) (position_xp_yp xp yp l fo))
```
[6]

We define those parts of the function that correspond to others types of moves in different ways. For example, retrograde promotion with capturing (*promotion_cap_3*) is a double move like we see in figure 1.

[5] The function *position_new* is very long and we only can show here some of its elements.

[6] Parameters *xp*, *yp* and *xk*, *yk* are the coordinates of the starting and end squares, *fp* is a piece on the starting square, *fo* is a captured piece and *l* is a list of the lists with the given position of pieces on the board.

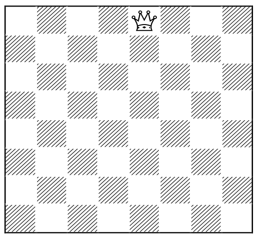
The part of the function *position_new* that corresponds to the move shown on figure 1 looks like this:

```
match xp with
1 => position_xp_yp 1 yp (app (app (beginning_of_list 1
(position_xp_yp 2 (yp-1) l P)) ((change_of_piece yp O (nth 1
(position_xp_yp 2 (yp-1) l P) nil)) :: nil)) (rest_of_list 2
(position_xp_yp 2 (yp-1) l P))) fo
| 8 => position_xp_yp 8 yp (app (app (beginning_of_list 8
(position_xp_yp 7 (yp-1) l p)) ((change_of_piece yp O (nth 8
(position_xp_yp 7 (yp-1) l p) nil)) :: nil)) (rest_of_list 9
(position_xp_yp 7 (yp-1) l p))) fo
| _ => nil
end
```

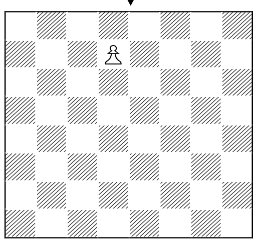
We get a new position (position in the moment of time *on+1*) if we apply the function *position_new* in the position in the moment of time *on*. We store this new position in a special hypothesis *H_new_position*:

```
Variable H_new_position : position (on+1) =
position_new xp yp xk yk (fp rb) fo vp (position on).
```

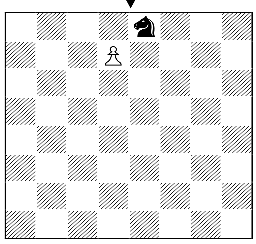
Figure 1. Retrograde promotion of the white queen with a captured black knight as a double move



Move of empty square (piece *O*) from *(2,4)* to *(1,5)* and the captured white pawn



The empty square is replaced with the black knight



3.4 Functions for computing check positions

3.4.1 Recursive functions for the bishop, rook and queen

In this section, we outline the recursive functions that check the content of the squares, starting from the closest square of (white or black) the king in all eight directions. If the first non-empty square in a diagonal direction engages with the opponent's bishop or queen, or if the first non-empty square in a vertical or horizontal direction engages with the opponent's rook or queen then the observed king is in check. If another piece is on the first non-empty square, then the king is not in check from the observed direction.

For example, the function for direction *left-up* (*lu*) looks like this:

```
Fixpoint check_lu_k (xkb ykb : nat) (pos : list (list pieces)) {struct
xkb} : Prop :=
 match xkb with
   S xkb' => match ykb with
             S ykb' => match nth ykb' (nth xkb' pos nil) v with
                    O => check_lu_k xkb' ykb' pos
                   | Q => check_lu_queen_k /\
                         x_check_lu_queen_k=xkb' /\
                         y_check_lu_queen_k=ykb'
                   | B => check_lu_bishop_k /\
                         x_check_lu_bishop_k=xkb' /\
                         y_check_lu_bishop_k=ykb'
                   | _ => True
                  end
           | _ => True
          end
  | _ => True
 end.
```

3.4.2 Functions for the knight and pawn

The knight and pawn functions do not need to be recursive, since they only need to check either on a concrete square with regard to king is opponent's knight or pawn.

For example, the following function will check whether the black king is in check with a white knight:

```
Definition check_knight_k (i':nat) (l : list (list pieces)) (xkb' ykb':nat)
: Prop :=
     match i' with
       1 => match nth (ykb'-1) (nth (xkb'-2) l nil) v with
            N => check_knight_k_1
           | _ => True
          end
      | 2 => match nth (ykb'+1) (nth (xkb'-2) l nil) v with
            N => check_knight_k_2
           | _ => True
          end
      | 3 => match nth (ykb'+2) (nth (xkb'-1) l nil) v with
            N => check_knight_k_3
           | _ => True
          end
      | 4 => match nth (ykb'+2) (nth (xkb'+1) l nil) v with
            N => check_knight_k_4
           | _ => True
          end
      | 5 => match nth (ykb'+1) (nth (xkb'+2) l nil) v with
            N => check_knight_k_5
           | _ => True
          end
      | 6 => match nth (ykb'-1) (nth (xkb'+2) l nil) v with
            N => check_knight_k_6
           | _ => True
          end
      | 7 => match nth (ykb'-2) (nth (xkb'+1) l nil) v with
            N => check_knight_k_7
           | _ => True
          end
      | 8 => match nth (ykb'-2) (nth (xkb'-1) l nil) v with
            N => check_knight_k_8
           | _ => True
          end
      |_ => True
     end.
```

3.5 Computing the new positions of kings, new player's turns, the numbers of white and black pawns, the total number of white and black pieces and the ordinal number of the move

If one of the kings has been moved, either by a standard move or by castling, their coordinates will also change. We need functions to compute these new coordinates. For the white king, the functions are:

```
Definition change_xKw on : nat :=
 match (fp on) with
   K => xk
  | R => match tm with
        castling_kingside_white => 8
       | castling_queenside_white => 8
       | _ => xKw on
      end
  | _ => xKw on
 end.

Definition change_yKw on : nat :=
 match (fp on) with
   K => yk
  | R => match tm with
        castling_kingside_white => 5
       | castling_queenside_white => 5
       | _ => yKw on
      end
  | _ => yKw on
 end.
```

The functions for the black king are analogous. We store the new coordinates in the following hypotheses:

```
Variable H_new_xKw : xKw (on+1) = change_xKw on.
Variable H_new_yKw : yKw (on+1) = change_yKw on.
Variable H_new_xkb : xkb (on+1) = change_xkb on.
Variable H_new_ykb : ykb (on+1) = change_ykb on.
```

In a similar way, we define the functions and hypotheses for computing and storing other information mentioned in this section: which player's move it is, the numbers of white and black pawns, the total number of white and black pieces and the ordinal number of move.

3.6 Hypotheses about check positions

In the hypotheses about check positions we will store the results of the functions for computing check positions (see chapter 3.4). We can split this kind of information into four groups:

1. Is the player whose turn it is in check?
2. Is the player whose turn it isn't in check?
3. Will the player whose turn it is be in check after his move?
4. Will the player whose turn it isn't be in check after their opponent's move?

If the answers to the first and fourth cases are positive, then the position is not valid and must be eliminated.[7] If the answers in the second and third cases are positive, then the position is valid. However, in the second case the player whose turn it is must in their move eliminate the check position. The third case will be in the next move the same as the second.

So, we introduce into the context hypotheses about the check positions in two adjoining moments of time:[8]

Hypothesis H_check_knight_k_*M* : check_knight_k M (position on) (xkb on) (ykb on).

Hypothesis H_check_pawn_k_*N* : check_pawn_k N (position on) (xkb on) (ykb on).

Hypothesis H_check_*DIRECTION*_k : check_*DIRECTION*_k (xkb on) (ykb on) (position on).

Hypothesis H_check_knight_k_*M*_new : check_knight_k M (position (on+1)) (xkb (on+1)) (ykb (on+1)).

Hypothesis H_check_pawn_k_*N*_new : check_pawn_k N (position (on+1)) (xkb (on+1)) (ykb (on+1)).

Hypothesis H_check_*DIRECTION*_k_new : check_*DIRECTION*_k (xkb (on+1)) (ykb (on+1)) (position (on+1)).

$M \in \{1, ..., 8\}$

$N \in \{1, 2\}$

DIRECTION $\in$ {lu, ru, rd, ld, left, right, up, down}

4 Reasoning about retrograde chess problems

4.1 Goal

In our system we present the goal as the proposition *not_valid_move*:

```
Parameter not_valid_move : Prop.

Goal not_valid_move.
```

During the reasoning about chess positions the logical value of the proposition *not_valid_move* will be unknown except in those cases when we conclude that a move or position is not valid. To eliminate invalid moves we introducing the following meta-axiom:

```
Axiom Invalidity_of_move : not_valid_move=True.
```

4.2 Tactics, tacticals and *Ltac* function

In this article we do not present tactics for generating retrograde chess moves or reasoning about the validity of moves and their related positions.[9] We simply use the *Ltac* function *One_Move* which is made up of all these tactics. Whilst generating the valid retrograde moves in a given position, this function inductively builds up starting and end squares, captured pieces and the types of moves. In this way, this function builds up a certain number of subgoals. Each subgoal belongs to one retrograde move. With the developed heuristics invalid moves are eliminated as soon as possible. After the first application of the function *One_Move*, only the valid moves remain in the form of unproven subgoals. The context of each of these subgoals is the same as the starting context. In the second iteration, the function *One_Move* will be applied to all remaining subgoals and so on. In such a way, we use *Coq*'s proof tree as our tree of

[7] See Axiom V in Section 1.
[8] Here we show just the hypotheses for the black king. For the white king the hypotheses are analogous.

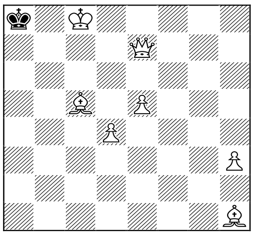
[9] The code of our system has more than 5500 lines and more than 200,000 characters.

moves and positions. In the last three sections we will show how the function *One_Move* can be used for a higher level of reasoning - reasoning about sequences of retrograde moves.

4.3 Invalidity of a given position

Diagram 5: Which side is white? R. M. Smullyan [10, 13]

North



South

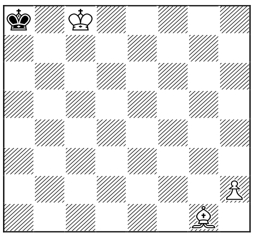
Let us assume that the white side is on the south. We can check this assumption by repeatedly applying the function *One_Move* in a given position:

```
repeat One_Move.
```

Our system proves that the position is not valid.[10] It means that the assumption is wrong and that white is on the north.

4.4 Last move

Diagram 6. What was black's last move? R. M. Smullyan [10, 23]



The results of applying the *Ltac* function *One_Move* are three unproven subgoals with the following hypotheses:

```
H_list_moves : list_moves 1 =
moved 0 k 1 1 2 1 B standard_move :: nil

H_list_moves : list_moves 1 =
moved 0 k 1 1 2 1 N standard_move :: nil

H_list_moves : list_moves 1 =
moved 0 k 1 1 2 1 O standard_move :: nil
```

So, we must to use the *Ltac* function *One_Move* at least twice:

```
One_Move;
One_Move.
```

In this way we get five unproven subgoals with the following hypotheses:

```
H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 b standard_move :: nil

H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 r standard_move :: nil

H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 q standard_move :: nil

H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 n standard_move :: nil

H_list_moves : list_moves 2 =
moved 0 k 1 1 2 1 N standard_move ::
moved 1 A 1 1 3 2 O standard_move :: nil
```

We can see that every list of moves contains the same first retrograde move: the standard move by the black king from *a8* to *a7* with retrograde captured white knight. So, the problem is solved and solution is: the last move of the black was the move *Ka7a8x* with the capture of the white knight.[11]

4.5 Last *n* moves

Now we can solve problem 1 from the Section 1 using our system. We have to apply the following tactical on goal (we need to find at three last moves):

```
One_Move; One_Move; One_Move.
```

We get the following solution which is in accordance with the solution we already gave in Section 1:

```
H_list_moves : list_moves 3 =
moved 0 K 6 3 6 2 p standard_move ::
moved 1 p 6 3 xz yz O p_ep_cap_6 ::
moved 2 P 5 3 7 3 O p_2 :: nil
```

[10] All subgoals become proven.

[11] Solving such types of problems is also automated in our system.

5 Summary

In this article we have shown that the *Coq* - a formal proof management system, and *Calculus of Inductive Constructions* - the underlying theory of the *Coq*, can be used for developing the environment as a base for reasoning about retrograde chess problems. This environment is comprised of axioms, definitions and hypotheses of chess objects, as well as functions for computing changes in chessboard. Apart from the available *Coq*'s tactics, in order to be able to solve these problems, new tactics (by using *Coq* tacticals) are created as well as especially heuristics in the form of more complex tacticals. Due to their complexity, these tacticals are not presented in this article but they are used for solving several presented problems.

References

[1] Bertot Y.: **Coq in a Hurry**, available at `http://coq.inria.fr`, Accessed: 20th January 2007.

[2] Bertot Y., Castéran P.: **Interactive Theorem Proving and Program Development: Coq'art: The calculus of inductive constructions**, Springer-Verlag, Berlin and Heidelberg, 2004.

[3] Delahaye D.: **A Tactic Language for the System Coq**. In Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France), Springer, LNCS/LNAI (1955), 85-95, 2000.

[4] FIDE - World Chess Federation: **Laws of Chess**, available at `http://www.fide.com`, Accessed: 22th May 2008.

[5] Gimenez E., Castéran P.: **A Tutorial on [Co-] Inductive Types in Coq**, available at `http://www.labri.fr/perso/casteran`, Accessed: 01th January 2007.

[6] INRIA - The French National Institute for Research in Computer Science and Control: **Faq about coq**, available at `http://coq.inria.fr`

[7] INRIA - The French National Institute for Research in Computer Science and Control: **The Coq Proof Assistant - A Tutorial**, available at `http://coq.inria.fr`, Accessed: 16th January 2007.

[8] INRIA - The French National Institute for Research in Computer Science and Control: **The Coq Proof Assistant Reference Manual Version v8.1**, available at `http://coq.inria.fr`, Accessed: 21th May 2008.

[9] Janko O.: **The Retrograde Analysis Corner**. `http://www.janko.at/Retros/`.

[10] Smullyan R. M.: **Chess Mysteries of Sherlock Holmes: Fifty Tantalizing Problems of Chess Detection**, Random House Inc., 1994.

[11] White K. F.: **Artificial intelligence and retrograde chess analysis: The design of a production system core rule base and interpreter**, M.S. thesis, Unpublished, University of Florida, 1990.