Stanford Artificial Intelligence Laboratory
Memo AIM-327

April 1979

Computer Science Department
Report No. STAN-CS-79-727

THE INTERACTION OF OBSERVATION AND INFERENCE

by

Robert Filman

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: STAN-CS-79-727, AIM-327

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): The Interaction of Observation and Inference

5. TYPE OF REPORT & PERIOD COVERED: technical, March 1979

6. PERFORMING ORG. REPORT NUMBER: STAN-CS-79-727 (AIM-327)

7. AUTHOR(s): Robert Elliot Filman

8. CONTRACT OR GRANT NUMBER(s): ARPA MDA 903-76-C-0206, ARPA Order 2494

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Department of Computer Science
Stanford University
Stanford, CA 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Avenue, Arlington, VA 22209

12. REPORT DATE: March 1979

13. NO. OF PAGES: 235

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)
Mr. Philip Surra, Resident Representative
Office of Naval Research, Durand 165
Stanford University

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An intelligent computer program must have both a representation of its knowledge, and a mechanism for manipulating that knowledge in a reasoning process. This thesis is an examination of the problem of formalizing the expression and solution of reasoning problems in a machine manipulable form. It is particularly concerned with analyzing the interaction of the standard form of deductive steps with an observational analogy obtained by performing computation in a semantic model.

Consideration in this dissertation is centered on the world of retrograde

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 094 120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

analysis chess, a particularly rich domain for both observational tasks and long deductive sequences.

A formalization is embodied in its axioms, and a major portion of this dissertation is devoted to both axiomatizing the rules of chess, and discussing and comparing the representational decisions involved in that axiomatization. Consideration was given to not only the necessity for these particular choices (and possible alternatives) but also the implications of these results for designers of representational systems for other domains.

Using a reasoning system for first order logic, "FOL", a detailed proof of the solution of a difficult retrograde chess puzzle was constructed. The close correspondence between this "formal" solution to the problem, and an "informal, descriptive" analysis a human might present was shown.

The proof and axioms were then examined for their relevance to general epistemological formalisms. The importance of several different mechanisms were considered. These included: 1) retaining both the notion of "current status" (typically embodied as the current chessboard) and that of a "historical state" (a hypothetical game played to reach a desired place), 2) evaluating functional and predicate objects in the semantic model (the chess eye), 3) the value of "induction schemas" as partial solutions to frame problems, 4) the retention of explicit undefined elements within the representation, 5) the importance of manipulating multiple representations of objects, and 6) a comparison of state vector and modal representations.

Stanford Artificial Intelligence Laboratory
Memo AIM-327

April 1979

Computer Science Department
Report No. STAN-CS-78-727

THE INTERACTION OF OBSERVATION AND INFERENCE

by

Robert Filman

An intelligent computer program must have both a representation of its knowledge, and a mechanism for manipulating that knowledge in a reasoning process. This thesis is an examination of the problem of formalizing the expression and solution of reasoning problems in a machine manipulable form. It is particularly concerned with analyzing the interaction of the standard form of deductive steps with an observational analogy obtained by performing computation in a semantic model.

Consideration in this dissertation is centered on the world of retrograde analysis chess, a particularly rich domain for both observational tasks and long deductive sequences.

A formalization is embodied in its axioms, and a major portion of this dissertation is devoted to both axiomatizing the rules of chess, and discussing and comparing the representational decisions involved in that axiomatization. Consideration was given to not only the necessity for these particular choices (and possible alternatives) but also the implications of these results for designers of representational systems for other domains.

Using a reasoning system for first order logic, "FOL", a detailed proof of the solution of a difficult retrograde chess puzzle was constructed. The close correspondence between this "formal" solution to the problem, and an "informal, descriptive" analysis a human might present was shown.

The proof and axioms were then examined for their relevance to general epistemological formalisms. The importance of several different mechanisms were considered. These included: 1) retaining both the notion of "current status" (typically embodied as the current chessboard) and that of "historical state" (a hypothetical game played to reach desired place), 2) evaluating functional and predicate objects in the semantic model (the chess eye), 3) the value of "induction schemas" as partial solutions to frame problems, 4) the retention of explicit undefined elements within the representation, 5) the importance of manipulating

multiple representations of objects, and 6) a comparison of state vector and modal representations.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

Acknowledgments

I would like to take this opportunity to thank the many people who have contributed to the completion of this dissertation.

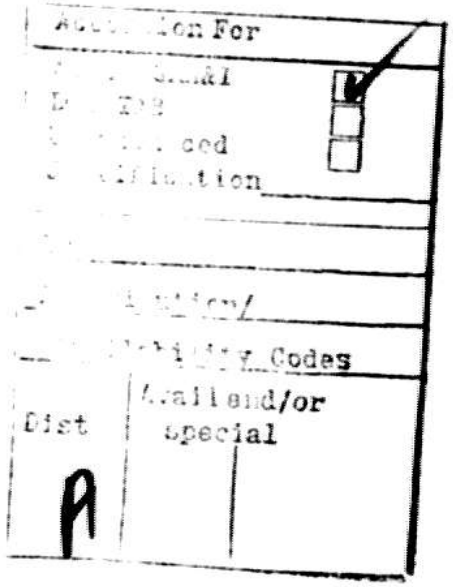
I am especially grateful to my adviser, Dr. John McCarthy, and the other members of my reading committee, Drs. Richard Weyhrauch and Terry Winograd, for their kind and patient reading and advice. Without their support and direction, this dissertation would not have been possible. Without their guidance and helpful criticism, this dissertation would have been markedly inferior.

To the other members of the FOL project, Dan Blom, Juan Bulnes, Ashok Chandra, Bill Glassmire, Chris Goad, Dave Poole, Andrew Robinson, Carolyn Talcott, Arthur Thomas, and especially Rich Weyhrauch for for providing a system where checking reasoning was possible.

To Jim Davidson, for reading even those sections I had forgotten I'd written. To the other people who have, over the years, provided discussions and suggestions for improvement to system. I can't remember all of them, but they certainly include Avra Cohn, Lew Creary, Martin Davis, Scot Drysdale, Bob Elschlager, Bill Faught, Brian Funt, Dick Gabriel, Scott Kim, Fred Knoll, Bob Moore, Allen Newell, Rich Pattis, Hanan Samet, Dave Shaw, Bob Smith, Peter Suzman, Nori Suzuki, Dave Wilkins, and Don Woods.

To the staff and system programmers of the Stanford Artificial Intelligence Laboratory, for providing such a productive working environment.

And of course to Myrna Kay, for providing the emotional support and stability I needed to get all this done.



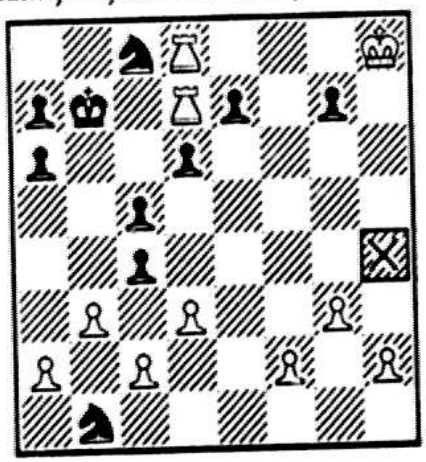
CONTENTS

Chapter 1 Introduction

Section 1.1 Synopsis: A Summary

A intelligent computer program must have both a representation of its knowledge, and a mechanism for manipulating that knowledge in a reasoning process. This paper is an examination of a difficult problem in retrograde chess, particularly with respect to formalizing the expression and solution of that problem in a machine manipulable form. In effect, this is both an exploration in the symbolic representation of knowledge and a characterization of the shape of the resulting knowledge space.

Our consideration centers on the following retrograde chess analysis puzzle (*figure 1*). Its solution (from basic chess principles) is certainly beyond the ability of any current computer program.



A piece has fallen off of the board from the square marked X. What piece was it? This position was achieved in a legal chess game, though there is no presumption that either player was playing to win.

figure 1

This problem was selected because its solution "requires" both deductive and observational inferences, in a context isolated from other issues of correctness and sufficiency.[1]

The notion of deductive inference, obtaining new proof steps by the application of syntactic inference rules, ought to be familiar to the reader. We recognize, however, that human reasoning proceeds not only by deduction, but also by the immediate recognition of results, a process we identify with *observation*. We have extended our representational system to include observational inference by performance of computation in a partial semantic model. Thus, for example, a human

---

1. The "requires" is in scare-quotes, for, technically, any of these functions can be decomposed into logical form. Any program can also be expressed in Turing machine form; it is, however, as much folly to write those things that should be evaluated as predicates as it is to write programs that should be LISP in Turing notation.

chess player might *see* a black knight checking a white king on some board. This inference is performed in our system by computing the check predicate within the semantic model. This result might be applyed in the deduction that the black knight was the last piece to move, or that it is now white's turn, by syntactic application of deductive inference rules.

Within the context of the solution of our chess problem, we attempt a synthesis of the two. In particular, we will axiomatize the rules of chess within *first order logic* (our declarative representation), but include within our system a method for evaluating (when we know how) the values of predicates and functions (which will serve as a form of procedural representation).

We shall also highlight the various representational decisions made in the process of axiomatizing retrograde chess. We will consider both the necessity for these particular choices, and their implications for designers of representational systems for other domains.

Using a proof checker for first order logic (FOL, [Weyhrauch77]), we detail a proof for the solution of the given chess puzzle. In the process, we show the close correspondence between our *formal* solution to the problem, and an *informal, descriptive* analysis.

This work should be viewed in the context of the search for *epistemologically effective* formalisms for artificial intelligence. We need representation structures that are sufficient to express those concepts we wish our computers to manipulate. However, if these formalisms are to be useful for our A.I. purposes, they must also be able to express these ideas *concisely* enough for computer manipulation.

It should be emphasized that, unlike many theses within our field, we are not demonstrating a *computer program*. Our research is on a more basic level. We are interested in the nature of the things that an artificially intelligent program would need to be able to do, without specifying the mechanism by which the program would tie these things together. We are not asserting here that creating an intelligent program is an easy task; quite to the contrary, there are numerous issues in the representation and manipulation of knowledge that require solution before a general human level intelligence could be produced. We hope here to shed some light on several of the different representation issues and ideas, and examine the power of their interaction.

Section 1.2 Paradigm: Artificial Intelligence

It is important to begin by expressing the underlying assumptions involved in this examination, to mention, in effect, "where we're coming from". We consider this thesis to be primarily centered in the subfield of computer science called *Artificial Intelligence*.

The study of *Artificial Intelligence* is an attempt to better understand the nature of intelligent processes. This endeavor is, of itself, neither unique nor novel. Understanding cognitive processes has long been the domain of many other sciences, especially philosophy, psychology and linguistics. While A.I. shares many concerns with these fields, it differs in that its primary concern is with the instrumentality of intelligent action. There exists a basic belief in A.I. that intelligent processes can be mechanized. This computer modeling of these processes has become the major paradigm of A.I.

The gross model for these experiments is that of *search through a problem space.*[2] The A.I. problem then naturally divides into two parts: defining the elements and operators of the space to be

---

2. See, for example, [Nilsson71] or [Newell72].

searched, and describing the mechanisms that the searcher uses to transverse that space. This is perhaps more familiarly represented as the distinction between the *representation of knowledge* and, perhaps anthropomorphically, *reasoning*.[3] Thus, if we are to model general intelligent behavior we must be capable of both symbolically encoding a representation of the world, and manipulating this knowledge through a reasoning process. It must be emphasized that these two cannot be regarded as separate and distinct entities; rather, the selections of particular data and control structures are strongly interrelated decisions. However, we seek some simplification through problem decomposition. Hence the emphasis in this thesis on the representation issue, rather than attempting to encompass the entire A.I. problem.

Section 1.3 Context: The Representation of Knowledge

This work is directed towards general issues in the computer representation of knowledge, not just heuristics and data structures applicable to one small domain. Many systems have been created, for instance, which apply specific knowledge to a single problem, obtaining powerful, though limited, deductions. These are typified by the "expert question answer systems". While expert behavior in a limited field can thus be had, these results do not generalize over into solving other, less well-structured problems.

A good example of purely specific knowledge representation systems are embodied in the game tree searching programs. While various stratagems and heuristics, particularly the alpha-beta heuristic[4], have been used to program competent game playing programs, the resulting programs have not been useful for solving problems outside of their limited expectations. Thus, while the typical chess program, confronted with a board, might be very good at answering the question, "What is the *best* move for white", it might well be unable to comprehend the meaning of "What is the *second best* move for white"[5]. There is no way, of course, of getting the typical chess playing program to incorporate knowledge of mathematics or geology, and therefore no way convincing the program to manipulate such knowledge.

Even confining ourselves within the chess domain, and restricting our attention to producing the "best" move from a given board, it is often quite difficult to instruct the chess program. While a suggestion like "keep your pawns in diagonal lines" or "avoid an unprotected king in the center of the board" might easily be incorporated by addition to the board evaluation functions, notions like "develop a strategy to obtain control of the center" and "work towards checkmate" are neither easily expressed nor simply implemented within the tree search paradigm.

However, this is not a paper on playing chess. Rather, we are addressing ourselves to representational issues, considering the criteria for useable knowledge representations. We would like our representation to be "general", not one for which we first select the domain of application, and then fit the knowledge structure. Our ideal representation should be able to express all

---

3. This distinction has been characterized in several different fashions. For instance, McCarthy-Hayes call it the epistemological and heuristic parts of the A.I. problem ([McCarthy69], pg 466), while Pratt (extending a notion of Chomsky on linguistics) refers to the competence/performance dichotomy [Pratt77]. Thus, one can think of "Epistemological Effectiveness" (section 1.3.1.3) as a form of "logical competence", just as Chomsky refers to a notion of grammatical competence ([Chomsky72])

4. The α-β procedure for searching games trees is described in [Nilsson71] section 5-12.

5. This example is by Allen Newell, in a personal communication

"questions" and "notions", or at least as many questions and notions as in human language is capable of expressing. Particularly important, a good representation system must have some mechanism for relating the multiple perspectives and organizations that are associated with any object. No good representational structure should have arbitrary limits on its extent. Rather, it should be an expansible system, one that can easily and uniformly include additional knowledge about both previously defined domains, and new areas. It is convenient if the selected representation is *natural*, its (basic) knowledge both readily apparent, and humanly understandable.[6] And, perhaps most germane to the current discussion, the ideal representational organization should be able to employ the most natural format for expressing each "fact", be it as a static rule, or a computational algorithm.[7]

Section 1.3.1 Declarative versus Procedural Representations

Section 1.3.1.1 The Power of Procedures

Let us consider that last qualification in some greater detail. We consider the existence of two species of knowledge. Declarative knowledge has each particular fact represented as a simple statement, such as *Laomedon was the father of Priam*, or *All red objects on the table are blocks.* Procedural knowledge embeds the given information as an algorithm. Typically, *To get to the train station, make a right at the second light, and go three blocks* or, *To find if there is a green pyramid in a blue box, check each object in each blue box, (to see if it is a green pyramid)* . What is given here is not so much a particular piece of information, as a well defined algorithm for determining the desired factor or achieving a desired state of the world. This distinction has often been characterized as the difference between *knowing what* and *knowing how.*

Procedures are algorithms; recipes for action. In this respect, they model any well learned activity. One does not do long division by reference to Peano's axioms, considering at each step the set theoretic meaning of the computation. Rather, one knows "how to divide", and does, just as one can recognize the checked king on a chessboard without considerations of orthogonality and color, or can find a phone number in the phonebook without requiring a derivation of the interpolation search algorithm. Here we speak of using procedures to model derivable, though well defined, recognitions.

Nor do we have to limit the power of our procedures to human size tasks. For most tasks requiring "intelligence", a computer is not (or, is not yet) a match for a human. However, it is fair to recognize that there are some things (well defined, complicated algorithms, preferably requiring either a long computation or a great deal of memory) which computers can do better than humans. A procedure that knows how to solve analytic integrals could use such a solution as a building block in some longer derivation. Here the solution of the integral is a single step in the larger deduction, though the actual computation involved in the integration might well be great.

We will explore these notions in some specific cases in section 1.4.

It is worthwhile mentioning that our notion of procedural knowledge differs in several important respects from a similar (and probably more familiar concept): the *procedural* a.i. languages, of which the major exemplar is PLANNER [Hewitt71]. These languages are similar to our aforementioned

---

6. Natural for: permits easier composition, verification and understanding of the represented system.

7. Algorithm here is meant to also include the employment of physical devices by our intelligent machine. For instance, a robot doing visual analysis might use a special processor, equipped with television camera, that found edges or regions in its visual field.

scheme, in that much of the *knowledge* of programs written in Planner is embedded in procedural definitions. They differ, however, in that our notion does not include the implicit control structure (particularly pattern matched invocation) dominant in the procedural languages Our functions state *how* to compute some value; there is no explicit or implicit demand when the actual computation should take place. Additionally, we shall see that our notion of the procedure to compute x is subordinate to our notion of x; we discuss this mapping in section 1.3.1.4.

Section 1.3.1.2 The Deficiencies of Procedures

One might well expect, after reading the previous section, that we are about to hoist a banner, Knowledge = Procedure. Not so. We recognize that procedural representation is sometimes appropriate. Most particularly, when one knows how to compute some value, computing it might well be the best idea.

But procedural representations have their limitations. For one thing, procedures are best written in a structured and modular form. That is, we would like the procedure that computes X to be able to do that without regard for "the rest of the world", (subject, of course, to conventions about data structures, communications, and the like). In the same spirit, and within those assumptions, we want our procedures to perform the most efficient computation possible. But this *black boxing* of a procedure presupposes that the internal structure of the procedure will not be examined. Hence, we will need some other way to express the relationships between procedures, and the invariants of the particular procedural computation. In effect, we may need to reason about some procedure, and the program semantic formalisms available to do this are not strong enough. Note that our notion of *the procedure as a black box* corresponds strongly with human limitations on introspection. For instance, no one can describe *how* he sees some scene, for example, what makes a particular object *red*.

This fixation of the procedural definition delimits the possible uses of a given piece of knowledge. Typically, procedural representations of entire predicates (as embodied by most purely procedural languages) implicitly specifies the only uses of that knowledge. Thus, if we know that *All computer science graduate students are bright and overworked* we may want to use this knowledge to prove that Tom, a computer science graduate student is bright, or that Dick, who is not overworked, can not be a computer science graduate student, or that, combined with the fact that all A.I. graduate students are computer science graduate students, there is no dumb A.I. graduate student.[8] The procedural *language* formalism demands that each possible use be associated with an explicit occurrence of that information.

Lastly, procedural representations dependent upon computation on completely specified objects, such as a complete data base of objects, will be unable to reason about situations involving incomplete knowledge and multiple representations.

To summarize, while procedural representations are often quite powerful, they retain certain inadequacies. Our list is by no means exhausted; comparison with section 1.3 show several other, obvious deficiencies. A more complete discussion of the problems of purely procedural representations can be found in [Moore75].

If the meaning of natural language (that is, English, Latin, Basque, ...) expressions were more precisely defined, and suitable for algorithmic reasoning, then perhaps a natural language

---

8. These being a slight extension of ideas of Winograd in [Winograd75].

representation would be appropriate. Language is, after all, one of the major mediums of thought. But the "pretend it's English" [Hayes77] approach to representation runs into problems of inherent ill-definition. What, after all, does this English structure mean? And how is it to be used? Language, we see, reveals itself to be too imprecise and ambiguous to serve as our representation. Rather, we need a firmer epistemological foundation for our knowledge system.

Section 1.3.1.3 A Declarative Alternative

> ***Modern formal logic is the most successful precise language every developed to express human thought and inference. Measured across any reasonably broad spectrum, including philosophy, linguistics, computer science, mathematics and artificial intelligence, no other formalism has been anything like so successful.***
>
> *P. J. Hayes*[9]

To fill the gap between a natural language system and a pure procedural representation, we propose the use of formal logic, particularly an extended first order predicate calculus.[10]

Logic was originally conceived in an attempt to precisely delimit the nature of human reasoning. This is a theme extending back through to Aristotle. It is a notion that reached its apogee by the middle of the nineteenth century, perhaps best exemplified by George Boole's *magnum opus*, **An Investigation into the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities.**

Modern logicians are not quite as dogmatic on this point. It is now recognized that there are many domains which formal logic does not (yet) model well. Particularly of interest to those of us in A.I. are the various model logics of knowledge, belief, tense and ability. These are areas of current study in both mathematics and philosophy. Until these problems are resolved, we can hardly assert the universality of formal logic as a representational system. Even so, using formal logic for computer representations has several advantages:

1. The sentences of first order logic are fairly natural. With a little practice, one has no difficulty with either composing such sentences, or understanding the meaning of a given sentence. In fact, they have a much clearer semantics than ambiguous natural language. Similarly, it is fairly easy (for humans) to translate between many natural language constructs and first order logic.

2. First order logic has explicit quantification (∀ and ∃). Some other formalisms, particularly network formalisms, have no explicit method of producing existential quantification. Other network formalisms lack explicit negation.

3. There are partial decision procedures for first order logic (procedures which can sometimes decide the validity of a WFF), and decision procedures for parts of first order logic (such as

---

9 [Hayes77]

10 The "first order logic" used in this volume is the propositional calculus (connectives ∧ ∨ ⇒ ⊃ and ¬) extended by the inclusion of predicates, quantification (∀ and ∃), functions (operators), the notion of equality, and the ability to do inference by computation in a semantic model.

propositional logic with equality and monadic predicate calculus). Here we look ahead to the heuristic side of the A.I. problem. The validity of some first order logic sentences is determinable by certain decision procedures. In particular, the tautologies of propositional logic, tautologies of propositional logic with equality without substitution in functionals, and monadic predicate calculus are all examples of decidable logics. A reasoning program using a first order logic representation could easily take advantage of these procedures. Similarly, there are heuristic procedures for first order logic. For example, the various forms of resolutions are heuristic methods for logic.[11] As automatic theorem proving progesses, these better and more powerful procedures become immediately available to a logic based system.

4. First order logic satisfies our criterion of generality. It is obviously not tailored to one particular domain. One hears a complaint from the gallery, "But logic is for mathematics." Perhaps so, but this paper is especially a demonstration of an application of first order logic to a non-mathematical (though well structured) domain.

5. Knowledge can be added to a declarative system through the addition of lemmas and theorems. There should not be any need to *know how* this new information the interaction will with the current data base, other than to insure that no contradiction, deducible by the heuristic portion of the program, is thereby introduced.[12]

6. We propose a method for keeping the power of procedures that know how within the framework of the formal logic system. In effect, have some of the best of both worlds. We consider this notion in greater detail in section 1.3.1.4.

7. We have within first order logic a good mechanism for describing the equivalence of different representations. We can do so explicitly, especially through the use of the equality relation, and universal generalization.

8. In some sense, the alternate representations currently extant are just other, and sometimes fuzzier forms of logic. For example, most of the notions currently titled *semantic networks* are as well expressed as well formed formulas; deductions and representations in one can be mapped to the other. Similarly, anyone familiar with **LISP** must recognize the interchangeability of data and functions. But there is an inherent problem with other systems that formal logic, with its strong syntactic and semantic restraints does not share. The *meaning* (semantics) of the particular constructs within these other representations are not well enough understood to be completely analyzable. Non-monotonic systems are particularly prone to this problem. The truth value of the various functions within systems of this kind is frequently tied up to the heuristic mechanisms involved in computing that value. Statements about such systems, for example, whether they can or cannot deduce some particular result, must therefore lead into a dynamic analysis of the action of the entire system. And such analysis, as our brief experience with program verification should show us, is a difficult problem. Formal logic systems, with their notion of *truth*, have the property that *anything once deducible, will remain deducible despite the addition of any other information (axioms).*

---

11. We are not asserting that one wants to reason strictly by using resolution (or that one even wants to use resolution at all). Rather, one wants to use useful inference rules, and, in certain circumstances, resolution might be useful.

12. We are, admittedly, skimming over a pair of important points. We have here proposed a consistent data base, or at least one whose inconsistencies are not apparent. But determining if a set of axioms is consistent is, in general, undecidable. Whether one wants a consistent data based is an arguable question. It is certainly not true that humans maintain a consistent knowledge system. However, we seek here to model human external actions, not internal processes. Secondly, we have pushed off to the problem space searcher the question of what to do with this additional information.

Systems such as *PLANNER* ([Hewitt71], [Winograd72]), where negation is used interchangeably with a truth value that is, effectively, *I can't find a counter example.* lack this property.[13]

We are not alone in recognizing the importance of a uniform semantics for a representation system. [Hewitt73] considers this issue in detail in his design of his procedural ACTORS formalism.

Perhaps a word on the value of first order logic, in contrast to higher order or simpler logics. It is clear that purely propositional logic is insufficient for our task; one of the major reasons we want first order logic is its ability to express quantification and predicates.

Dismissing higher order logics is not quite as easy. First order logic is capable of expressing set theory, and therefore, all of mathematics. It is not obvious that a higher level logic might not provide a more convenient expression for some real world domains. However, first order logic is complete; additional axioms can add any needed extension. Higher order logics are not complete; not every true statement has a derivation. Additionally, first order logic provides us with a well defined semantic model; the more interesting higher order logics lack this feature. One of the demonstrations of this paper is that an appropriate semantic model can be a very useful aid to the deduction process. So, we consider first-order logic here; this is a is very powerful logic, with a large existing literature on its manipulation. Note that some of the things that computer scientists think they need higher order logics for can be accomplished through the use of axiom schema (see, perhaps, sections 2.2.4.1 and A.2.1) in first order logic.

It is important to mention that use of a first order logic representation system is not at all the same as marriage to a resolution style proof mechanism. Pure resolution proof checkers have proven to be failures. While such an algorithm might be a small portion of a full artificial intelligence system, it is clear that it cannot be the sole (or even the major) inference mechanism. We mention this caveat because, unfortunately, formal logic and resolution are "married" in the minds of too many people in the A.I. community. What we are dealing with here are primarily representation issues; even to the limited extent that we touch upon heuristics, we wish to state that we are not implying a resolution style approach.

We also wish to emphasize the distinction between *representational formalisms* and *representational data structures.* A parallel to automata theory might clarify this difference. There are many machines that retain a given degree of computing power: various automata equivalent, for example, to finite state machines or Turing machines. Any class of machines can solve certain problems, though some particular machine within that class might solve the given problem more quickly or require less storage to do so.

The state of representational formalisms is similar. Certain formalisms can express certain truths about the world. Formalism have certainly not reached the *Turing* level of expression; there are many issues of representation we do not how to adequately express. Among them are the issues of representing knowledge about how to use knowledge, representing beliefs, and representing chronological developments.

For any given set of axioms and inference rules, there is a set of statements provable from those axioms and rules. In a complete logic, such as first order predicate calculus, the set of provable

---

13. This is not to assert that the truth value expressed by lack of a counter example is not a useful heuristic quantity. However, it is probably better if our representational systems recognize the distinctions between "proven" and "unable to prove, but currently assumed true".

theorems is equivalent to the set of true statements. Classically, an axiomatization is *adequate* if all of the desired truths can be derived in it.

Now, A.I. is a more practical sort of an affair. It is not merely sufficient for us that a given result be eventually obtainable; we ideally desire two other things: that the result be concisely derivable, and that there be some methodology a program could employ to find that derivation.

The second of these is the heuristic adequacy problem, and is beyond the scope of the current discussion. Rather, we are concerned with two things: finding representations in which what we want to say can be expressed, and insuring that that expression is of manageable magnitude. We call this the search for *epistemologically effective* representation formalisms.

Epistemologically effective formalisms are not a question of data structure. Rather, it is the combination of classical epistemological adequacy (expressiveness of logical language) with an appropriate set of inference rules to allow reasonable proof to be obtained.

We notice that one of the things that human problem solving does to shorten derivations is to employ both standard deduction, and a quicker *noticing a conclusion*, something we have associated with observation, and suggested can be performed by procedural computation on ground instances. We explore this combination in the next section.

Section 1.3.1.4 A Suitable Marriage

We desire a composition that will permit us both the effective advantages of procedures, and the expressive quality of declarative methods. We also want that this unification retain the mathematically valid foundation accrued by use of our original formal system.

To perform this marriage, we turn to model theory, and postulate the following. We assume that (as an underlying structure) we have a LISP world. This world contains individuals (S-expressions, hereafter abbreviated Sexpers), and functions and predicates in the usual LISP - lambda function notation. Above this world, we have our formal first order logic system, the usual collection of individuals, variables, predicates, and functions. We then create a map between our logic constants and the LISP world model, prescribing for some constants a LISP Sexpr, which we then assert will act like that constant. Thus, in a system reasoning about arithmetic, we might map the individual constants ONE, TWO and THREE to the respective LISP atoms 1, 2, and 3, the predicate < to the LISP predicate LESSP, and the operator + to the LISP function PLUS. Our logic system would then be able to derive the validity of sentences using these constants by invoking the computational mechanism in the LISP model. For example, the sentence 2 < (1 + 3) would be seen to be *semantically* true in the LISP model, and therefore valid in the formal logic system.[14] In effect, we retain the ability to easily compute *how*, when we can compute, while still being able to reason about the computations. We have not increased what we can *say*; however, use of this device will free us to talk about more interesting things than, for example, set theory and Peano axioms.

We hope that the resulting system will retain the advantages of both; that the computational functions can be invoked when most appropriate, while retaining the powerful descriptive ability of the formal logic representation.

---

14. For a more formal explanation of this relationship, see [Weyhrauch77].

Earlier (section 1.3.1.1), we suggested that the most appropriate use of this computational ability would be to model the human ability to *observe*. Persisting in this notion, we dub our semantic procedural attachment, our Eye. Thus, for our chess problem, we have a *Chess Eye*. Similarly, an automated physician, capable of doing its own chemical analysis, might perceive this knowledge through its *Lab Eye*, or an electrical design facility might consult a simulation *Tech Eye* to verify the correctness of a circuit. Any computer individual performing a test on the real world will need to employ some sort of device; quite likely, this device will be some computer system "called" object. The intelligent part (as opposed to the perceiving part) of our computer individual could easily refer to this perception action as the evaluation of the particular device function, just as the semantic model evaluation performs simulated functional evaluation. Hence, computation can be used just like perception.

Section 1.3.2 Our Scheme

To summarize: we have, as our representational framework, a system founded on first order logic. We declare predicates, functions and constants in this logical system, and express some of our knowledge as axioms of the logic. Additionally, we will *attach* functions and constants in our LISP model to the constants of our logic system, and will use these LISP objects to compute the values of predicates and functions (attachment of semantic procedures). The legal operators of our system are fundamentally the rules of inference of the first order logic; we extend them to include *computational evaluation within the LISP model* (our EYE), and whatever decision procedures for this logic we have available. We use this system to examine the topography of long reasoning sequences.

It is perhaps useful to emphasize that this structure constitutes the framework within which we work. It is, we believe, a broad enough structure to accommodate most consistent formalisms for any particular problem. We will, in the following chapter, be demonstrating the feasibility of this framework for a particular formalization of the given chess problem; in the concluding chapter (chapter 5), we will consider how alternative formulations of the problem might be expressed within the framework of first order logic with semantic procedural attachment.

We presume the reader is more familiar with the inference rules of predicate calculus than with our particular implementation of semantic procedural attachment. Perhaps a word about simplification would be appropriate. Our simplification system employs two major computational mechanisms. As initially conceived, its purpose was to use the attached functions to compute on constants of the logic system. Thus, one could take a (constant) board, such as the puzzle board of our problem (*figure 1*) and compute a predicate (for example, *Black is in check*) on it. It has since been extented to include the ability to evaluate WFFs quantified over finite sets.[15] Thus, one can simplify a predicate that asks, *Is there a black bishop on a square of the given board.*[16] We shall consider in the section 5.8 various desirable extensions to this scheme.

Proofs of the size we contemplate would be impossible to write (correctly) were it not for the existence of a mechanical (computer) proof checking. We are fortunate to have available, for verification of our proof, **FOL**. FOL is a proof checker in the first order logic. It originally checked proofs of the *natural deduction* style of Prawitz **[Prawitz65]**; it has since been extended to include

---

15. The performance of this computation varies, of course, with the size of the sets involved. Practically, we have been patient enough to check WFF's with up to 2^{12} cases.

16. The simplification mechanism, as embodied in FOL, also performs other inference tasks, such as decision procedures on the sort hierarchy, and inferences about the membership of finite sets.

decision procedures on tautologies, and the beginings of deductive ability within a LISP model (what we have been denoting semantic procedural reasoning or an eye).

This proof checker acts much like the Missouri Program ("show me") described by McCarthy and Hayes [McCarthy69]. It "allows the experimenter to present it proof steps and checks their correctness".[17] The various decision procedures incorporated into FOL may be viewed as either making this Missouri Program more discerning, or as being steps towards the Reasoning Program mentioned in that paper.

The bulk of the remainder of this paper presumes knowledge of the FOL system. An introduction to FOL, of adequate detail for understanding the FOL used in this paper, may be found in [Filman76]. A full description of the syntax and semantics of FOL is the FOL manual [Weyhrauch77].

Section 1.4 Analogs: Other Eyes

Back at the beginning of this thesis (section 1.1) we mentioned that this is primarily (at least by the measure of physical paper use) a demonstration of the proof of a chess puzzle. However, we are concerned with the general representation issues, and find it profitable to present a few short examples of our representational scheme applied to some other domains, particularly emphasizing the employment of procedural Eyes. In contrast to our major proof, which is a highly detailed though unidirectional derivation, this detour is best perceived as speculation and hypothesis. We are not presenting a system of axioms and attachments for these worlds, but rather, a brief overview of how these techniques might be applied in them.

It is important to point out here that this section is not dealing with how perception might be performed; rather, we are describing a system that, through the semantic procedural attachments, is able to talk about its perceptions in the same language as the "rest of its thinking".

Section 1.4.1 The Mechanic's Eye

We consider first a representation to embody some of the knowledge employed by an automobile mechanic in diagnosing a malfunctioning automobile. Of course, whatever we say can be related to the maintenance of any similar machinery. What must such a person know? Primarily, the mechanic knows the interconnections and functions of the various parts and subassemblies, particularly with an eye towards recognizing malfunctions (and potential malfunctions) of individual components.

How could a computer be employed in such a task? One imagines an extension of the current engine electrical analysis systems. Instead of (or in addition to) displaying the current levels and frequencies of various wires on a CRT, such a monitor would pass the information back to the computer through an appropriate ADC. Special devices might be attached to, say the exhaust pipe or water pump, to measure composition or pressure, and convert these signals to digital values. Effectively, these devices would permit the machine to *observe* the state of the running engine. They would act (combined with appropriate functions to transmute these real time signals) as part of the computer's eye.

Typically, our automated mechanics would have axioms such as:

---

17. [McCarthy69] pg 469.

$$\forall x.(\text{Voltage}(\text{Battery}(x)) < \text{Minivoltage}(\text{Cartype}(x))) \supset \text{NEEDREPLACEMENT}(\text{Battery}(x)))$$

which would be read to mean: *for all cars x, if the voltage on the battery of x is less than the minimum voltage required for cars of x's make, then that battery needs replacement (or repair).* The mechanic program could then simply observe (by simplifying the given formula) whether the battery was performing correctly. Note that (from the point of view of the logical language level) we are able to perform both the perceptual task involved in measuring the battery's current, and checking (in the mechanics manual) the appropriate voltage for this car by employing the same mechanism. From the computer's point of view, observation inside its "head" is the same as observing the real world.

Given the prevailing technology, we can hardly expect the computer to fix the car alone. Rather, we imagine it to be the partner of a human mechanic, who could both ask help from the computer, and provide non-digital measurements. The computer might request the tire-tread wear statistics for the car, and then ask the human mechanic to push the front end up and down. His reply (and the questions) could be used in (and generated by) evaluating:

$$\forall x.\exists s.((\text{IRREGULAR-WEAR}(\text{Tire}(x,s)) \wedge \text{BOUNCES}(x,s)) \supset \text{NEEDREPLACEMENT}(\text{Shock-absorber}(x,s)))$$

That is, *if, for a car x, there is a side of x* (left-front, right-rear, ...) *whose tire is wearing irregularly, and which bounces after pushing, then the shock absorber on that side of that car needs replacement.* The simplification (computer observation) would request the appropriate information from the human mechanic, in addition to doing the calculations.

Meta-knowledge, that is, general rules applicable to systems, might also be expressed axiomatically:

$$\forall j\ k.((\text{CONNECTED}(j,k) \wedge \text{CURRENTTHROUGH}(j) \wedge \neg\text{CURRENTTHROUGH}(k)) \supset \text{NEEDREPLACEMENT}(k))$$

or, *for any two electrical components j and k, if j and k are connected electrically, and there is current at j, but not k, then k is defective.* For example, if there is current leaving the distributor, but no spark at the plug, then the ignition wire is broken. By looking at the electrical connections, our automatic mechanic *sees* the validity of any instantiation to this axiom. But it is up to the main mechanic program to (heuristically) decide what instantiation to make.

Section 1.4.2 The Personal Assistant's Eye

Here is a second example of the combination of procedural observation embedded in a formal system. One thing I would like of my computer, is for it to be my personal assistant, effectively, my secretary. It should be capable of tasks such as scheduling appointments, planning trips, and making coffee. To do these tasks most successfully involves both actions of a simple procedural nature (such as table look up or message transmission) and of deductions of a more complex, *reasoning* variety. For instance, I might want my assistant to arrange a trip to Pittsburgh for me. To accomplish this task, the program would need to look up the airline schedule, relate the information found to its knowledge of my flight preferences and other appointments, call the airline and hotel for reservations, find a way to and from the various airports, print a list of directions, and so forth. There are several different abilities involved here. The program must reason about my knowledge and desires (it doesn't need to tell me, for instance, how to get to the San Francisco airport -- but I might need information about ground transportation in Pittsburgh). It should realize that I prefer

flights with a movie and meal. It might believe that I have axioms telling it to use the least expensive flight, or to avoid a particular airline. But it does not need to do an involved reasoning sequence to find out what flights exist. Rather, it can see the flight schedule merely by looking it up in a table, an observational activity. We see that we need a formalism strong enough to be able to reason about knowledge and desires, but which can still efficiently solve simple algorithmic problems. The semantic procedural attachment mechanism to a full logic seems the appropriate solution.

My secretary program would (in this ideal, non-existent world) also communicate with other programs and machines. It could call the airline and hotel computers to arrange the reservations. Another interesting communication domain involves scheduling appointments with the programs of the other people on our system. Our ideal program can observe my schedule (table lookup), consider my preferences (avoid appointments before 11:30), and send and receive messages from the other secretary programs. Note that the acts of sending and receiving are procedural actions, naturally expressed by executing functions. The updating of various tables associated with particular states accomplishes a large portion of fixing the tense logic. Within this general formalism, this updating and searching is accomplished by attaching the executing procedures to the associated functions. And this system allows us to reason about the actions of sending messages. Our system need also be able to distinguish between thinking about sending a message, a purely *gedanken* experiment, and actually sending it. Thus, it can reason, *if I send him a message asking for an appointment tomorrow, it will probably give us a 10:00 am meeting. But if I ask for a more preferable time, like 2:30, I may get it ...* .

Making coffee involves turning on some real, physical device. Once again, it is accomplished through some function call. We imagine, perhaps, an execution of the COFFEE UUO. Once again, we seek interaction with the real world represented by the use of a function call. Needless to say, simplification of some other function permits the program to observe when the coffee is ready.

Section 1.4.3 The Engineer's Eye

As our last example, we consider the representation of knowledge for a computer engineer. Basically, we will wish to describe physical systems to this program, and have it verify properties of these systems. For example, our engineer could be given a circuit, and asked to prove some functional property of the outputs, relative to the input currents, or given a system a moving bodies in some force field, and asked to determine the possibility of collision. Such a procedure might be part of a hardware design and verification program, or a module of a computer aided instruction system.

Our system will know general laws about objects, suitably expressed as formulas of our logic. Thus, a typical axiom about moving bodies would have:

$$\forall x\ t\ v_0\ a.s(x,t)=.5\circ a\circ t^2 + v_0\circ t$$

or, the distance reached by body x, by time t, is the product of the (constant) acceleration of x between t and t_0, and t^2, plus the distance traveled by x due to its initial velocity during interval t.

The program must be capable of both manipulating such formulas as formulas, and using them to produce numeric answers. The natural rules allowing substitution of equals, and instantiation of axioms allow for the formal manipulation. When a program using such a representation needed to solve for a particular value, in could *observe* (via the simplification mechanism) and compute it.

Our engineer might also be called upon to design systems. Humans have ready access to familiar, solved subproblems. For example, *adders* and *registers* are the components from which human designers build bigger digital systems. Our computer engineer can have a list of solved subproblems of his own, and (with an appropriate procedural call), can consult this list for the correct device. Once again, we have an observational operation obtained by procedural semantic attachment within our general formalism.

Section 1.5 Why Chess

> ***It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity.***
>
> Marvin Minsky[18]

The end of our detour. Though concerned with general epistemological issues, we are presenting, primarily, one particular example of the use of our representation system. As we stated in section 1.1, this thesis pivots around the demonstration of a solution of a chess puzzle, within the first order logic (and semantic simplification) formalism. It is perhaps useful to detail some of the justification for examining puzzles about chess, and not some other problem domain.

Section 1.5.1 Structure and Search Spaces

There are several dimensions to be considered in the selection of a domain for A.I. research. The primary one, shaping the entire model, is the degree of structure inherent to the task. Recall that we described computer intelligence in terms of a *search through a problem space* (section 1.2). We introduce the notion of measuring the structure of this space, along two different dimensions. Such a space can vary both in the specificity of its elements, and the degree of definition of the operators for transferring between these elements. In general, the more limited the elements of the space, and the clearer the transference operators, the more amenable the problem is to computer solution. The current generation of A.I. programs are mostly concerned with those problems for which there is typically a fairly large number of states, but clear rules for state definition and transition. Intelligence for programs such as these lies in selecting the appropriate heuristics for navigation. Beyond the ability of present machine intelligence is negotiation of spaces with ill specified operators or states. Effectively, we have no programs that can *creatively* generate and select operators and states; we have difficultly *representing* the operators and states of ill-structured domains.

Spaces are also distinguished by the size of their solution sequences. Obviously, the fewer the number of steps needed to solve a given problem, the easier it is to obtain the solution. With several choices of applicable operators at any point, longer solutions can become exponentially difficult. Typically, current problem solvers produce long, but certainly not very long, solutions.

This measure is reflected in the current "state of the art" of generating "smart" machines. We see successful "expert" programs, dealing with well structured and relatively small problem spaces, mediocre mathematical programs, dealing with very well structured but very large spaces, but no "creative" or "common sense" programs, dealing with both large and ill-structured domains. More specifically, the better one is able to formalize the rules and structure of some domain, the more successful one's program can be at "solving" the problems of that domain.

---

18. [Minksy68], page 12.

In this thesis, we are concerned with extending the length of solution sequences, within the context of fairly well structured problem spaces. We view this activity as laying the groundwork for much longer and more complex reasoning programs. Effectively, we need to know the lie of the terrain, before sending our computer out to transverse it. We also need a measure of the obstacles and steps, to be considered in designing the right "legs" for our explorers, gauging the difficulty of the course, and, perhaps, the building of special tools.

We are not presenting specific methods for improving the most ill-structured domains. Rather, we seek to extend the present structured domains (though not artificially well-structured domains). More particularly, we want a problem space that is not purely artificial but, rather, corresponds to the irregularities of natural systems. We want a problem we can solve, not one we must defend from semantic objections and different interpretations.

Similarly, this domain should be complex enough to require long reasoning sequences. Most hard problems of the "real" world do not derive their difficulty from the depth of the reasoning required for their resolution. Rather, problems arise out of the poor structure and broad knowledge base inherent to "real" domains. The problem is not then not merely the storage of information, but, more importantly, its selection.

One domain obviously satisfies some of the above criteria: mathematics. Deduction sequences in mathematics can be arbitrarily long; mathematical proofs are presumably not (very) open to questions of semantic validity. But the mathematical domain retains shortcomings. parsimony within mathematical structures that is not paralleled within more synthetic systems. Effectively, we find mathematics *too* well structured a domain.

So we step away from orderly mathematics, and towards a more ill-structured task. By considering a game system, with rules delimiting the domain, we acquire a well specified structure. We will not be bothered with semantic quibbles, for it is clear from any state what legal transitions exist. But with as old and dynamic a game as chess, we also get an arbitrary and irregular rule system. As we shall see in chapter 2, these irregularities dramatically increase the complexity of the representation.

Chess retains yet another appeal. We profess to be interested in extending the size of deduction sequences. From the (relatively) small set of initial rules, we can produce problems of enormous complexity. Since our goal is not to test the size of initial structure we can store,[19] we find this an additional boon.

Section 1.5.2 Chess and the Eye

Chess puzzles have yet another attraction. We defined one of the purposes of this paper as an examination of the semantic simplification mechanism (our form of observation) as applied in detailed deduction sequences. Chess provides a good forum to display this notion. Our chess eye can roam freely in this world. It can, for example, be used to look at a board (or board fragment), and determine a checking or movement relation. Various theorematic knowledge, such as limits on the movements of pawns, can also be incorporated into the functions that make up the chess eye. And we permit our proof to *observe* the values of arit[illegible]ressions, rather than requiring their derivation. All these effectively parallel the observational ability of humans.

---

19. Our axioms, together with the proof checker, already tax the available memory of our computer system.

Section 1.5.3 Which Chess Puzzles

Before the reader becomes too mislead, let us state that we are (by and large) not talking of chess puzzles of the *mate in n* ($n = 1,2,3,...$) variety. For sufficiently small n, such puzzles become trivial tree search. Rather, we are examining the world of retrograde chess problems, puzzles where examination of a board fragment leads to deductions and constraints about the moves that led to that board.[20] Retrograde chess problems (and their solutions) can be extremely long and complex; a suitably difficult domain for analysis.

Let us also note that we seek these deductions from chess "first principles". That is, we will derive our solutions (by and large) from the rules of chess, rather than from the "theorems" familiar to chess puzzle solvers. This serves both to display the generality of our system, and to preserve our "honesty", for from a sufficiently powerful set of lemmas, any theorem is easily proven.

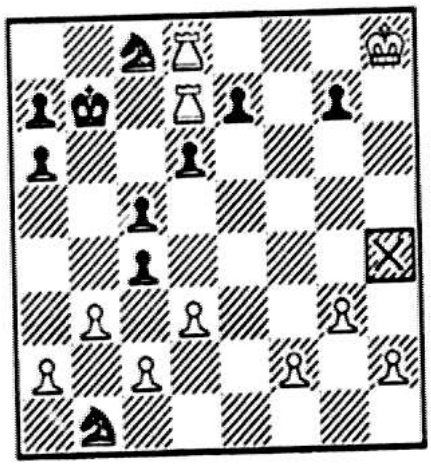
Section 1.6 Topography: The Path of Our Proof

We continue our descent from the general to the more specific. As we stated in the introductory summary (section 1.1), our attention is focused on the representation of the knowledge and reasoning implicit in the solution of one particular chess puzzle. Having kept the reader waiting long enough, it is perhaps time to state and solve that problem.

We examine the FOL solution to the chess puzzle illustrated in *figure 2*. It is a difficult problem, one whose solution requires inferences both about the given board and the game that preceded it. Deducing the identity of the fallen piece requires the use of many of the more subtle nuances of the chess rules.

---

20. The reader interested in other examples of retrograde chess problems is referred to [Dawson73]. While this book is primarily "fairy" chess problems, it also contains a number of retrograde analysis puzzles.



A piece has fallen off of the board from the square marked X. What piece was it? This position was achieved in a legal chess game, though there is no presumption that either player was playing to win.

figure 2

The reader may be unconvinced of the difficulty of this problem, and the complexity of its solution, if he has not himself attempted its solution. So we defer its answer to the next page.

Section 1.6.1 The Solution

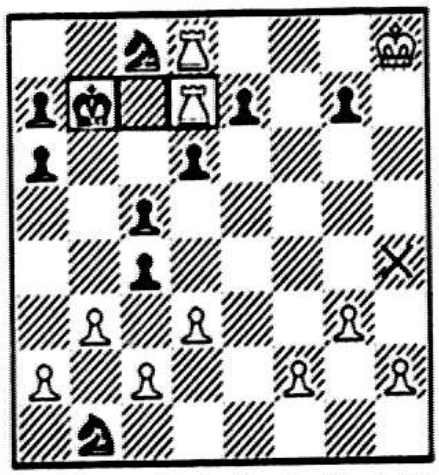
The reader has, of course, by now deduced that the piece that fell off of the X-ed square was the white queen's bishop. If the reader had reasoned the problem in sufficient detail, his analysis probably resembled the following:[21]

Section 1.6.2 The Reason

1. We see, in *figure 3* the white rook checking the black king. The king's check is a function of only the boxed three squares of *figure 3*; hence, the king will still be in check no matter what the fallen piece might have been.

1.1. It therefore must be black's turn to play.

1.2. And white must have made the last move.



The white rook checks the black king. It is black's turn to play.

figure 3

2. What was white's last move? There are several ways a check can be made. The checking piece can make the check, the check can be discovered by a piece moving out from between the checked king and the checking piece, the check can be discovered by the removal of a pawn captured en passant. To these we add a fourth method, to accommodate our (to be developed)

---

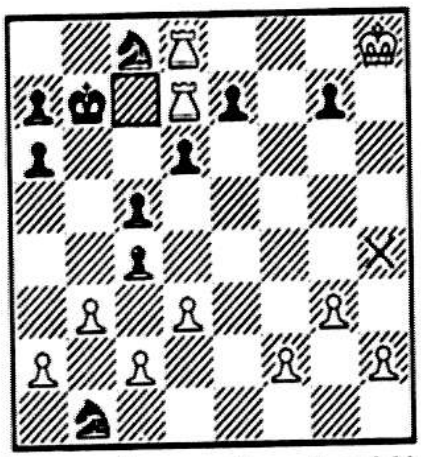
21. We observe the following chess notation in this discussion. Squares are named first by the color of the perspective side (White or Black), then differentiated as being on the King's or Queen's side. A modifying piece (Rook, kNight, Bishop) may be used to select the appropriate column, while the final digit describes the distance to that square from the edge of the board. Thus, the white queen begins the game on WQ1; she is likewise on BQ8. The square that the piece fell from, in out example, is both BKR5 and WKR4. Chesspieces are similarly named, by color, side and rank. Thus, the white king is WK; the pawn in front of black's queen side rook becomes BQRP (black queen's rook pawn). This is basically an abbreviation of the standard "English" system of chess names. We shall persist in this naming convention throughout the remainer of this paper.

formalism. As we will consider the king to be the "moving" piece of a castle, we consider the case when a just castled rook has made the check.

2.1. The last move was obviously not a castle by white. The white king is not on one of his castle destination squares. Nor is either white rook on a square reachable by a castle.

2.2. This check could not have been made after a capture en passant. En passant capture leaves a white pawn in the on the sixth rank. There is no white pawn on this row to have just captured en passant. Hence the last move was not an en passant capture.

2.3. Obviously, the only square the rook could have moved from is WQB7 (white queen's bishop seven, the distinguished square in *figure 4*). But the white rook checks the black king from that square too, and white can not begin his move with black in check.



The square between the rook and king.

figure 4

2.4. Hence, the check must have been a discovered check.

3. Well, then, what piece made the discovered check?

3.1. If the check was discovered, it must have been from a square between the rook and the king. But there is only one square between these two, WQB7 (noted in *figure 4*). Hence, the last move must have been made from that square.

3.2. What type of move was the last move? We have already concluded that it was not a castle or en passant capture. How about an ordinary move?

3.3. If the move was not a pawn promotion, one of the white pieces on the board (see *figure 5*), in its present incarnation, (that is, unpromoted), must have made that move.

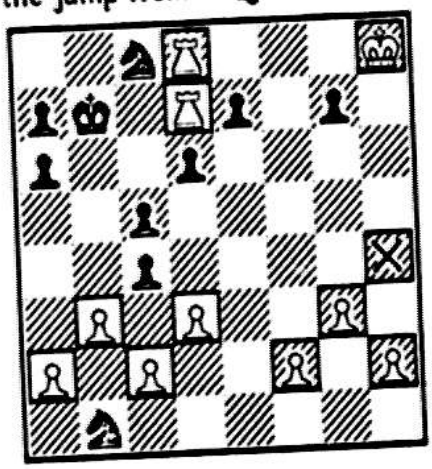
3.3.1. The white king on WKR8 certainly could not have been next to the black king on BQN2.

3.3.2. None of the white pawns could have moved from that square.

3.3.3. We have already eliminated the rook on BQ2 as a possible mover. This piece is making the check, not moving to discover it.

3.3.4. A rook on WQB8 could not have moved on that diagonal.

3.3.5. Nor could it have been the piece that fell off the board. No matter which piece it was, it could not have moved from WQN7. No piece can make the jump from WQN7 to BKR5.



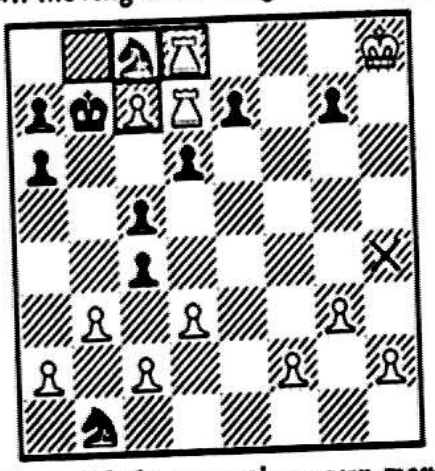
None of the (possibly) white pieces (in its present incarnation) could have moved to discover the check

figure 5

3.4. Therefore, the last move must have been a pawn promotion.

4. How did this pawn promotion go?

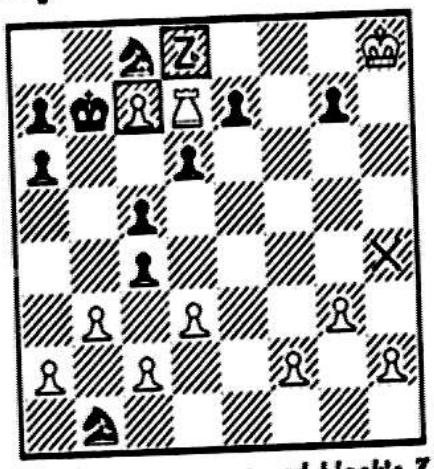
4.1. As we see in *figure 6*, a white pawn can move from WQB7 to one of three squares. Only one of these, WQ8, has a white piece on it. Thus, the last move must have had a white pawn moving from WQB7 to WQ8, promoting to a rook.



Where did the promoting pawn move?

figure 6

5. But to make this move, the white pawn must have captured a black piece. Let us call that piece Z_b (*figure 7*). What piece was Z_b?



The white pawn captured black's Z_b.

figure 7

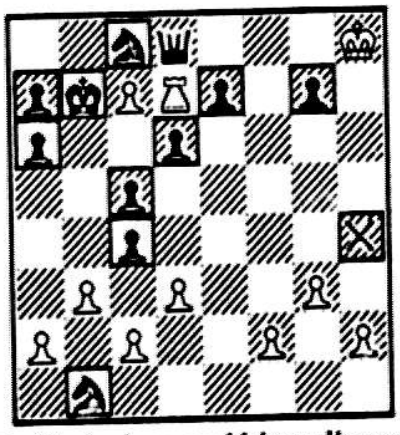
5.1. Clearly, black's last move was neither an en passant capture nor a castle. His pieces (pawns, king) are not appropriatedly arranged to have just completed one of these moves.

5.2. Perhaps Z_b was a black rook or black queen.

5.2.1. If that were the case, then white's king would be in check. And Z_b would be cornered, like the white rook on WQ7, unable to have reached that square except from another checking square.

5.3. So if Z_b was a rook or queen, it must have made that check through a discovered check.

5.3.1. But once again, none of the black pieces could have moved from between the checking piece and the white king. Nor could the piece that fell off have moved from any of those three squares.



None of the black pieces could have discovered check.

figure 8

5.4. Maybe the captured piece was a pawn?

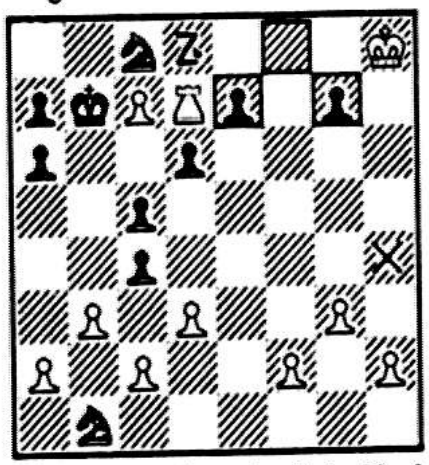
5.4.1. But pawns (at least unpromoted pawns, and here we are talking about the value of the captured piece) do not find their way to the first row.

5.5. The captured piece certainly was not the black king.

5.6. Could the captured piece have been a bishop?

5.6.1. It certainly was not the black queen's bishop. That is the black on white bishop (the black bishop that moves on the white squares). He would not be caught dead on a black square.

5.6.2. It was not the black king's bishop, either. Notice the two pawns in *figure 9* on BK2 and BKN2. They have not moved, and the bishop could not have gotten out from behind them.

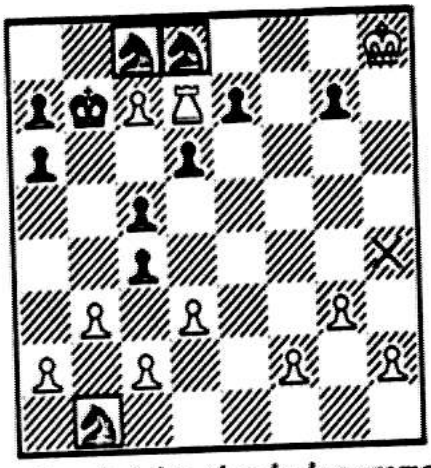


These pawns stymie the exit of the black bishop.

figure 9

5.7. Hence, if the captured piece had bishop value, it must have been a promoted pawn.

5.8. And, we can see in *figure 10*, that if the captured piece was a knight, then black had three knights on the board before white's last move. Anyone with three knights on the board at the same time (and who is not cheating) has promoted a pawn.



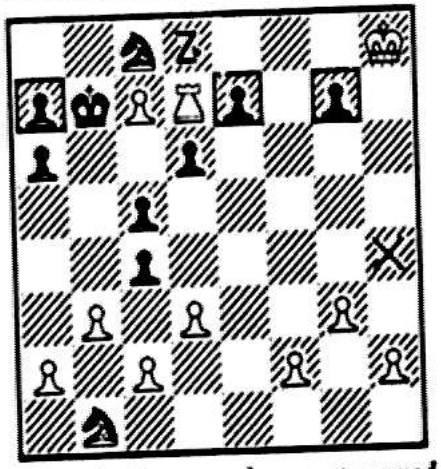
If black has three knights, then he has promoted a pawn.

figure 10

5.9. We have not learned the identity of the captured piece, but we have discovered an important fact: black must have promoted at least one of his pawns.

6. But which pawn?

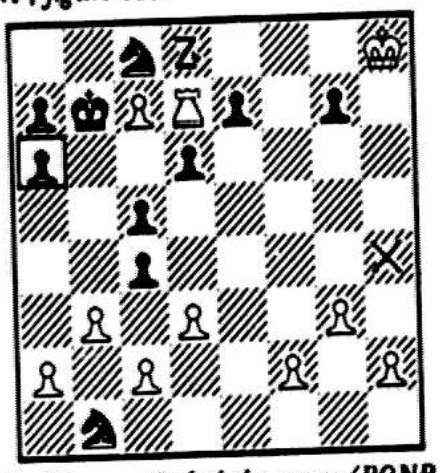
6.1. In *figure 11*, we see the three black pawns on black's second rank. These must have been the pawns that started on these squares.



These black pawns have not moved.

figure 11

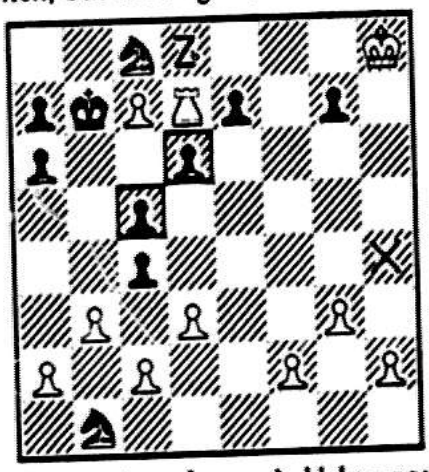
6.2. There is a pawn on BQR3. Since only two black pawns can reach this square, and we have concluded that one of them is on BQR2, this must be black's queen's knight pawn (BQNP, *figure 12*).



Black's queen's knight pawn (BQNP).

figure 12

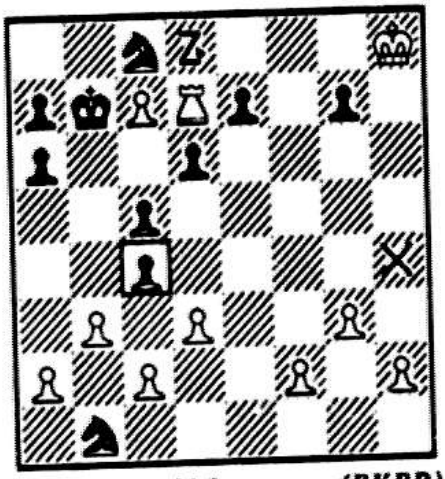
6.3. Of the remaining pawns, there are only two unaccounted for pawns that could be on BQB4 and BQ3, the black queen's bishop and queen's pawns. We have not established which is which, but then again, we do not care (*figure 13*).



Black queen's and queen's bishop pawns.

figure 13

6.4. Which means the pawn on BQB5, boxed in *figure 14*, must be the black king's bishop pawn.



Black king's bishop pawn (BKBP).

figure 14

6.5. So all the pawns except the black king's rook pawn are on the board. Hence, if a black pawn promoted (as we have already established), it must have been that pawn.

7. Could a black piece have fallen off the board?

7.1. Well, we have accounted for all the black pawns.

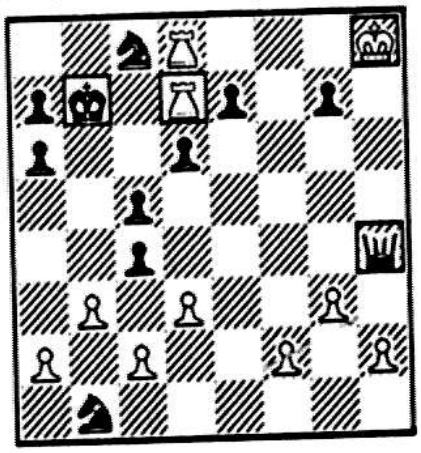
7.2. We have also determined that in the position prior to the given board, the two knights must have been on two of three squares.

7.3. The black on white bishop does not traffic on black squares.

7.4. The black on black bishop never escaped from his original square (as we have already demonstrated (figure 9)). He could not have been the piece that fell.

7.5. The black king is on BQN2.

7.6. If the fallen piece were rook or queen, then both sides would be in check on the original board (*figure 15*). This is clearly impossible.



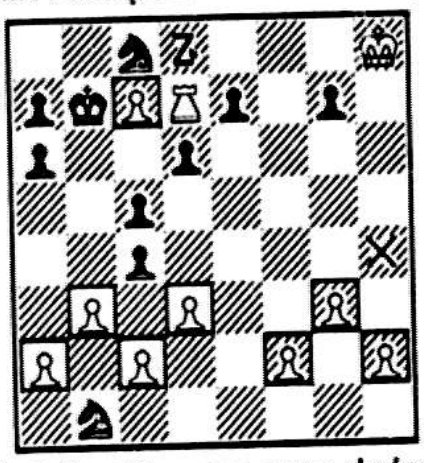
If the fallen piece were a black rook or queen, then both sides would be in check.

figure 15

8. Hence, the fallen piece must have been a white piece.

9. Could the fallen piece have been a white pawn?

9.1. By a process similar to that employed for black, we can identify all of the white pawns. In fact, just before the last move, all of them were on the board, in pawn incarnation. Hence, the fallen piece was not a white pawn (*figure 16*).



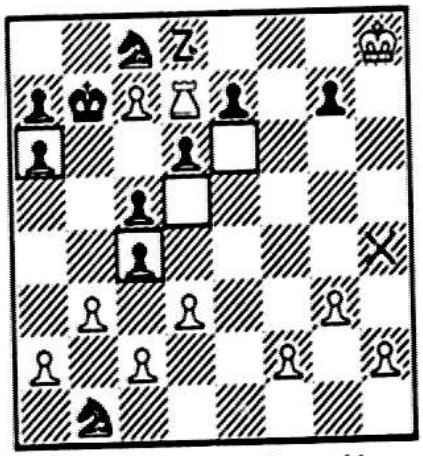
All of the white pawns are on the board.

figure 16

10. The fallen piece was obviously not the white rook on WQ7, nor was it the white king.

10.1. Thus, we have accounted for all the white pieces except the other white rook, both white knights, both white bishops, and the white queen.

11. We observe that the black queen's knight pawn, now on BQR3, and the black king's bishop pawn, now on BQB5, have captured four white pieces between them in reaching their current squares. Additionally, and most peculiarly, all of these captures have occurred on the *white* squares (*figure 17*).

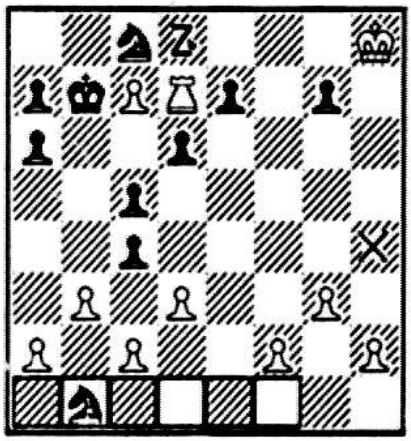


Four white pieces captured on white squares.

figure 17

12. We recall that the black king's rook pawn has promoted. What can we say about the path to his elevation?

12.1. If he promoted on any square to the left of BKN7 (*figure 18*), he would have had to make two or more captures.

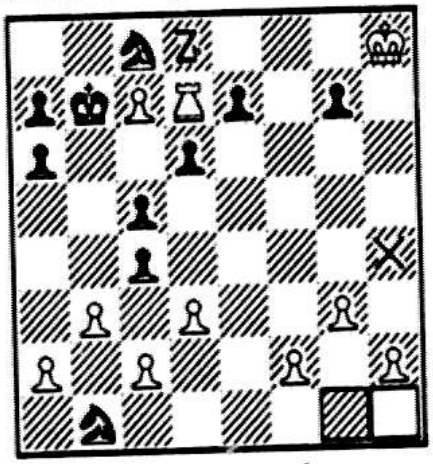


If any of these were the black promotion square then BKRP captured at least two white pieces.

figure 18

12.1.1. This would have required the capture of a total of six white pieces. There are already ten white pieces on the board. The capture of six white pieces would leave no white piece to have fallen.

12.2. Hence, the pawn must have promoted on BKR8 or BKN8 (*figure 19*).

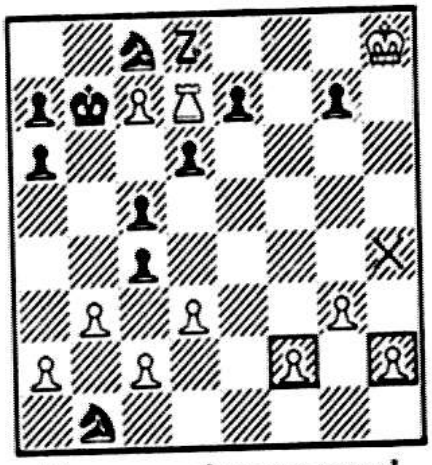


The two possible promotion squares.

figure 19

12.2.1. If the black king's rook pawn promoted to BKR8 or BKN8, then he must have moved into one of these squares on some move. What square would he have been moving from?

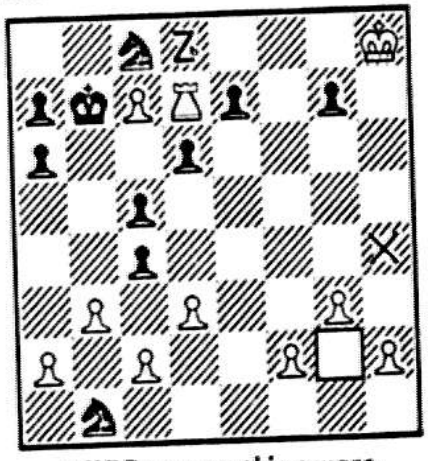
12.2.2. The white king's rook and king's bishop pawns, distinguished in *figure 20*, have not, for the duration of this game, left those squares.



These pawns have not moved.

figure 20

12.3. Therefore, the black king's rook pawn must have been on BKN7 (*figure 21*) before moving to promote.

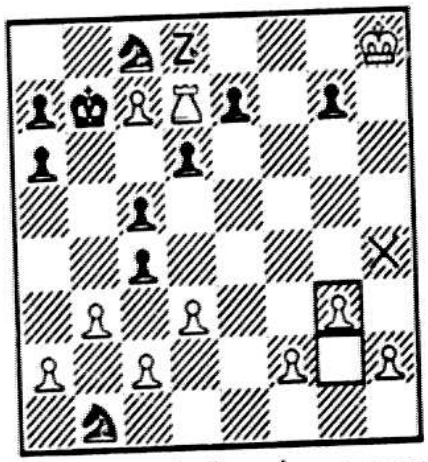


BKRP was on this square.

figure 21

13. How did that pawn get to BKN7?

13.1. The white king's knight pawn was a good deal more widely traveled than his neighbors. He has spent the game on two squares, WKN2 and WKN3 (*figure 22*).



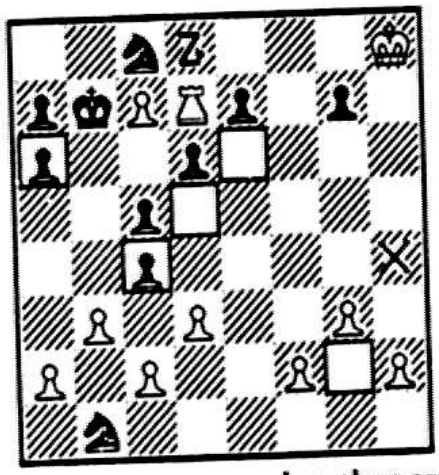
WKNP remained on these squares.

figure 22

13.2. In the move that brought the black king's rook pawn to BKN7, white king's knight pawn must have been on WKN3.

13.3. Hence, the black pawn must have made a capture in moving onto BKN7.

14. But BKN7 is a white square. Five white pieces have been captured, all by pawns, and all on white squares (*figure 23*).

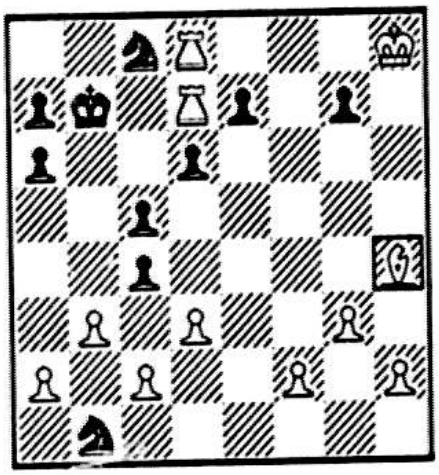


White pieces were captured on these squares.

figure 23

14.1. There are only six unaccounted for white pieces, all officers (non-pawns). Five of them have been captured on white squares. The white queen's bishop is never on a white square. Between the fallen piece, and the five captured pieces, we must arrange the falling square and the five capture squares. No piece is ever captured twice; no piece once captured, ever reappears.

15. Obviously, we can conclude that the fallen piece must have been the white bishop (*figure 24*).



The board before the Fall.

figure 24

Section 1.6.3 An Analysis

Several differences between the reader's reasoning process, and the above solution may be apparent. For one thing, quite trivially, the order of some of the steps may be permuted. This is of little consequence. Of more importance, however, is the detail to which we have developed our proof. We have included many of the steps that most humans would have avoided noting, for instance, statements to exclude kings in certain situations, where the human chess player would not even mention the possibility of their presence. This is partially an issue of heuristics; some steps are virtually automatic (and unmentioned) in a familiar reasoning sequence. But it is mostly because we presume that we are reasoning from the basic chess rules, and not from the theorems obvious to an experienced player. The restraint, the refusal to "jump to conclusions", is what permits the proof to "see" the promoted pieces as knights and rooks, when the experienced chess player (though not the chess problem solver) would quickly skip to the more "logical" conclusion that pawns promote to queens.

We have, however, availed ourselves of the ability to look at a board and "see" which pieces can move where. We cite this as an example of of the *observational knowledge* mentioned in section 1.3.1.1. Within our representational model, deductions of this kind are performed by function evaluation in the (LISP) model structure. Similarly, we leave to computation arithmetical evaluation; this is not a treatise on proving equations by Peano's axioms. Rather, mathematical calculations will be automatic, *procedural* in our system.

Section 1.6.4 Reasoning in a First Order Logic Formalism

We have a problem and a representation formalism, and with them, the assertion that the problem can be "solved" within the formalism. In some sense, much of the rest of this paper is that demonstration. We will first axiomatize the *chess world* in first order logic and then deduce, within our formalism, the unique solution of our chess puzzle.

This proof is the other side of the intelligence problem, the path through the problem space defined by our representation. Note that we are not claiming a program that can do this reasoning; this proof is human powered. Instead, we are exploring the path that a mechanized problem solver, using our formalism, would take. What results then is a map of the terrain; a guide for future explorers, an example of what is required to get through this particular "wilderness".

We assert that this proof, while not matching the level of detail of human analysis,[22] corresponds on grosser level to the human solution. That is, the individual inferences used in this proof are typically much smaller and weaker than human deductions. We will show, however, the correlation between the *chunks* of lines in our proof, and the individual steps of the natural deduction. We imply thereby the ability of our formal logic/semantic attachment system to model the human ability to accept problem solutions.

The analysis of the puzzle in section 1.6.2 will serve as the model of the "human solution".

---

22. Only the author, after talking to his proof checker all night, thinks in pure first order logic.

Section 1.7 Perspectives: Other Points of Interest

While exploration of representational systems is the dominant direction of this research, it retains several tangential interesting properties.

Section 1.7.1 Mathematics and a Chess Proof

We find this proof interesting for several mathematical reasons, unrelated to the problems of artificial intelligence.

Historically, mathematicians have used formal logic in two ways. Proofs of short mathematical theorems have occasionally been detailed within first order logic. But more commonly, mathematicians have used logic as a field to reason about, rather than in. One proves that a formal proof is possible, rather than presenting that elephantine object as a demonstration of its own existence. One writes proofs about proofs, rather than the proof itself.

This proof breaks with that tradition in both respects. It is an application of logic to a non-mathematical domain. As we will discuss in the conclusion, it exposes several strengths and weaknesses of the natural deduction system. In particular, the value of stronger inference rules, and semantic modeling will be considered. The difficulties of handling multiple representations and long proofs will also be mentioned.

It is also unusual for being a long, formal logic proof. A proof of this size, even with the help of a proof checker, has proven to be a non-trivial task. It is no surprise that there are not more of them.

Section 1.7.2 Machine Proof Generation and a Chess Proof

This paper should also be of interest to those interested in programming automatic theorem provers. We have, particularly in the appendices, numerous examples of first order proofs, which can be used as bench marks for those interested in creating their own systems. Effectively, we have a set of machine level examples that can be compared to computer deductions.

Section 1.8 Format: A Guide for Reading This Paper

This thesis is divided into several chapters and appendices. This first chapter has been the introduction, where we have presented our problem domain and motivation.

In the second chapter, we proceed to axiomatize the rules of chess in FOL. Concurrent with detailing these axioms, we describe and defend the various representation decisions embodied therein.

We begin to present FOL proofs in the third chapter. This section is a well commented sample of the proofs of several lemmas. It serves not so much to expound interesting theorems, as to familiarize the reader with FOL and our style of proof.

We present the proof of the fallen piece problem in the fourth chapter. In an important sense, this is the heart of this research. In the process, we draw the correspondence between this proof and the human proof of the first chapter.

The final chapter contains our conclusions, basically, what we have learned about the design and implementations of representational systems, with an eye towards their improvement.

There are several appendices, principally the proofs of many lemmas, and statistics about the proofs, and two indices, one for the document in general, and the other for the various labels and names used in the proofs.

The reader who has not the patience to read the whole volume is pointed towards the introductory and concluding chapters. A skimming of chapter two, the chess axioms, and a cursory glance at chapter four, the main proof, will aid in understanding the conclusions.

Section 1.8.1 The Proof Checker FOL

The reader unfamiliar with FOL will not receive the full benefit of reading this paper, though we hope the comments surrounding the various FOL sections will be of great value. For an introduction to FOL suitable for understanding this proof, the reader is referenced to [Filman76]. The complete description of FOL, including some its the mathematical motivation, can be found in [Weyhrauch77].

Section 1.8.2 Reading Proofs

Understanding a proof in first order logic is somewhat similar to reading an assembly language program. The level of detail is basically similar, and without annotation, the reader is sure to get lost.

In an attempt to avoid that tragedy, the proofs in the various chapters have been copiously commented.

Additionally, certain lexicographic and typographic conventions have be used in proofs in this paper. Any identifier in capital letters (CHESSPIECES, BKR) is either a predicate (PREDCONST) or individual (INDCONST). Functions (OPCONST) have only their initial letter capitalized. Lower case identifiers are used for variables (INDVAR). Predicate and operator parameters have been printed in script. Axiom names and labels are in capitals; theorems and lemma identifiers use both upper and lower case. A theorem name ending in an underbar (_) was obtained from a single simplification; a theorem name both begining and ending with underbars is an *unproven* theorem (section 2.2.5).

Chapter 2 The Chess Axioms

In any epistemological domain, we have a *basic* collection of information, what the system *knows* without further inference. This of course applies to our chess deduction. In a system like ours, of formal logic combined with a computational model, this knowledge takes several forms. We must first select the individual constants, predicates and operators of our formal system. Then the axioms of chess must be written. We need to organize the underlying model structure, and prescribe the mapping between the constants of our logical space, and the predicates, functions and individuals in our model. Throughout all of this definition, the correspondence between our definitions and the rules of chess should remain transparent.

We have decided to study chess partially because chess provides a well-defined set of rules. One might think that this regularity would prescribe some specific approach. But just as one can do formal proofs in arithmetic by computing on sets or Peano axioms, one has a choice in chess of the *level* of one's axioms. There exist both decisions to be made on a complexity dimension, and irregularities in the rules to complicate any organization.

The problems generated by the latter will be dealt with in depth in the remainder of this chapter, particularly as we handle each intricacy. It would be useful, however, to justify at this point the general complexity level of our approach, and the reasons for rejecting either a more or less *basic* set of axioms.

This proof is meant to be an examination of the reasoning that could be involved in the solution of retrograde chess problems. We wish to show the correspondence between the reasoning in this form, and the human deduction, while retaining the validity advantages of a formal proof.[23] It is not an attempt to prove mathematical theorems, nor do we wish to do with deduction what could be more easily observed. For these reasons, we have incorporated into the computational model functions to compute relations like individual piece movement. We have also passed to the computational model all arithmetic responsibility. In that sense, this is not a *low level* approach.

However, we also desire that our system be general in its ability to express many different kinds of retrograde analysis chess puzzles. We thereby become limited from above. We do not wish this analysis to be based on theorems applicable only to some small set of problems. Hence, we have expended considerable energy deriving general *chess theorems* from our axioms, and have used these theorems as individual steps in our main proof. These theorems are proven in chapter 3 and appendix A. But we are restrained by this generality restriction to consider chess at the piece and move level, rather than considering notions of general board geometry. A board geometric approach would express legal moves in terms of the pieces on a board, and procedures for expressing their movement ability. While easier to manipulate in the *short* term (proving things about the immediate predecessor or successor of a given board) such an approach would have difficulty expressing long term ("sometime, during this game, the following has happened") notions.

We are also bounded from above by the limitations of our proof checker. There are some things that are, by nature, observational, but nevertheless not computable within the present implementation of the proof checker. These restrictions, we might add, are discoveries of experience. We will consider possible improvements to the model computational method in section 5.8.

---

23. This is not to assert that we are modeling the way humans reason; rather, we are looking for a representation that a computer can reason with, which is still understandable (and verifiable) for a human intelligence.

Section 2.1 Declarations and Definitions

This chapter naturally divides into two sections: defining the objects of the chess world (with their FOL declarations), and expressing the rules of chess with these defined object. We begin, our course, by detailing and declaring the tokens of the chess axiomatization.

Interspersed with the description of this chapter are the text of the various FOL declarations and axioms used in generating this proof. Several of the declarations and functions here declared, while not mentioned in any of the proofs in this paper, have been included for completeness.

Section 2.1.1 Very Primitive Notions

In any axiomatization, there will be certain base notions, upon which the rest of the structure is built. Chess, of course, is no exception. We should display the distinction between the basic objects, and the less basic operations and predicates upon them. For this discussion, there exist seven basic sorts[24] of chess objects -- chesspieces, squares, piece values, positions, boards, moves and colors. While the necessity for some of these concepts is obvious (what can we say about a chess problem without referring to a chess board, or speaking of black and white?) the reasons for some of the others are more obscure, and will require some explanation. This section will detail each of these sorts, and their objects and objectives.[25]

Section 2.1.1.1 Positions

The fundamental object in this chess world is the *position*. A position is, effectively, a state vector containing all of the information needed to reconstruct an entire chess game. While this might, for instance, be conceptually encoded as a list of the moves made to reach that moment, or a list of the chess boards visited in the course of the game, it is generally not possible, in our system, to do so. More particularly, a position is not a *concrete* object (one that we can (usually) display or compute upon), but, rather, a conceptual notion.

Typically, a retrograde chess puzzle will be presented not as a position, but, rather, as an arrangement of chess pieces[26] on a chessboard, (what we will call a *board*). The puzzle is then to deduce the common factors of all possible games that could have led to such a board, effectively, the predicates true on any *position* with such a board. Nevertheless, we still wish to be able to retain our computational ability on the given (and associated boards). Hence, we see the necessity for representing what is essentially the same object (the board and the common factors of the games that lead to it) in several different representations. The lesson here for writers of programs that would seek to solve problems like this (and problems of similar complexity) is of the necessity for retaining multiple representations of objects.[27]

So, rather than having axioms manipulating positions themselves, our axioms will constrain the

---

24. A sort is a monadic predicate; one that therefore defines a set (the set of things for which it is true).

25. In the remainder of this paper, it is asserted that all individuals, except squares and dimensions, are represented uniformly as chess objects. We declare this representation with the command: declare REPRESENTATION CHESS;

26. Or, more precisely, an arrangement of chess values.

27. We will consider this result in greater detail in section 5.6.1.

transfer from one position to the next. Similarly, we will draw conclusions about the properties of successor and predecessor positions. It is worthwhile noting that from a given position, one can derive the previous position (the position on the previous move), and determine, of two positions, if one occurred in the game of the other.[28]

For example, we consider the following notions involving positions. We will have occasion to speak of the *path* some piece must have used to get to some square, without having to detail the interleaving between the moves of that piece and the other moves of that player, or to describe the capture of some particular chess piece, without detailing the particulars of the move (which piece made the capture?) involved. Thus, we will conclude, for example, that in any game played to reach a given board, there must have been another position (in that game) with some certain property, (for example, an unknown piece captured a bishop on that square) without ever having to state explicitly which prior position it was (which move during this game the captured occurred).

There is also an additional motivation for retaining the entire history of a game in our encoding. More specifically, one of the chess rules refers to the entire game. The castling rule requires that neither of the castling pieces have moved in the course of that game. That is, for some positions, the entire game must be considered to determine the legal moves.[29]

Our FOL declaration for POSITIONS:[30]

```
declare PREDCONST POSITIONS 1 [PRE];
```

Section 2.1.1.2 Pieces

Perhaps the most obvious sort needed in the solution of chess puzzles is one to represent the individual chesspieces. The implementation of this concept, however, is not so trivial. One quickly discovers[31] that not all pawns are the same; each of the thirty two chessmen has his own identity, distinguished mostly of his value and square at both the begining of the game, and at any later position. Note that we are differentiating between the identity of a chessman and his value; a pawn may promote to a queen, but in our eyes he remains a pawn in drag.

We will have need to talk of the *piece* on a *square* in a *position*. We therefore are required to add a thirty-third "piece" to our system, the EMPTY piece, the piece that sits on any square with no other occupant. Thus, the major sort of this scheme is PIECES, which includes the set of the thirty two CHESSPIECES. The FOL declarations are:

```
declare PREDCONST PIECES 1 [PRE];

declare PREDCONST CHESSPIECES(PIECES) [PRE];
```

---

28. An exception to much of what we say is, of course, the initial position. We have a complete description of the game that led to it, and can encode a particular representation for it.

29. Though the effect of this rule could be obtained by in a shorter term representation by "flagging" the "position" when one of the castling pieces moved.

30. This command declares the existence of a one place (monadic) predicate POSITIONS. POSITIONS is a prefix (PRE) predicate, and may be used without parentheses around its argument.

31. Particularly when dealing with chess puzzles, rather than playing chess.

Section 2.1.1.3 Squares

Another group of individuals is the set of squares of the chessboard. As with pieces, we have an extra member in our set, a heaven or hell for chesspieces, a place for them to be after they are captured and removed. We call this sort of extended squares EXSQUARES, and will occasionally speak of the extended square that a chesspiece is on in a given position, or which piece is on a given square in that position.

```
declare PREDCONST EXSQUARES 1;

declare PREDCONST SQUARES(EXSQUARES) [PRE];
```

Section 2.1.1.4 Values

Just as we spoke of the thirty two chessmen in a chess set, we will still often find it necessary to speak of their rank in a given position. To avoid confusion with rank and column, we shall henceforth speak of the VALUE of a chesspiece. Thus most pawns will promote to have a queen's value. We shall prove general chess theorems such as *All pawn valued pieces are pawns* and *All non-pawn (officer) pieces retain the same value through every position* (no officer ever promotes).

We also distinguish the color of a piece in its value. Thus, the *black king's pawn* (BKP) will usually have a value of *pawn black* (PB), but might occasionally[32] promote to be a *knight black* (NB). Failure to understand the fundamental distinction between the name of a piece and its value in a position will cause trouble understanding the motivation and detail of many of the proofs in this paper.

```
declare PREDCONST VALUES 1 [PRE];
```

Section 2.1.1.5 Boards

Most chess problems are stated not in terms of what we have called a *position*, but rather, as *boards* of distributed chess values. Similarly, most chess moves are defined in terms of the board structures they can to be made on, rather than the varieties of games that could precede different moves.[33] Therefore, we find it useful to have the primitive notion of a *Board* in our chess axiomatization. On the individual squares of a board, we meet the various values, including the value MT, which represents an empty square, and UD, a square on a board whose value is unknown. Note that one can speak of a pawn on a board without specifying which pawn it is. Our formalism includes partially and fully defined boards, and naturally lends itself to a partial ordering on boards by increasing definition. We speak of a fully defined board, one with no unknown squares, as being a TOTALBOARD.

```
declare PREDCONST BOARDS 1 [PRE];

declare PREDCONST TOTALBOARDS (BOARDS) [PRE];
```

It is reasonable to question the necessity for the *partially defined boards* introduced above. They

---

32. Particularly in puzzles.

33. There are, however, exceptions to this rule. The en passant capture, for instance, refers not only to the present board, but also the last move. Castling is not permitted if either the king or the rook has ever (in this game) been moved. Even more complicated are the various draw conditions, which demand the repetition of particular boards.

serve two primary purposes. First, they provide a structure for the representation of partial information about a situation. For example, we may know that a certain bishop has moved and captured, though we may not know what the captured chesspiece was. Nevertheless, through the employment of partial boards, we can compactly express the situation prior to the capture

A parallel, and perhaps more important reason, resides the in nature of FOL's simplification mechanism. Partial boards are computable objects; particularly, our LISP functions can make computations on expressions with explicitly undefined values, but not on partially defined expressions. This is similar to call-by-value LISP's inability to evaluate CAR(CONS(A x)) if x is undefined. Each of our attached functions and predicates on boards must know how to handle the partial piece. Partially defined squares typically restrict validity of predicates, for more information does not accrue from a less specified object.

Section 2.1.1.6 Moves

Our next sort is something of a pseudo-sort. A common chess notion is that of the *move*. We would like to be able to speak of the *last move* of a position as being a castling, or the white queen as the *mover* (piece that moved) of the last move of this position. Practically speaking, however, there are no occasions when a predicate or function on a move is used without first *extracting* the move from the position in question. As the state vector, the position retains all of the information in the move; hence, the sort itself is not needed; rather, it gets in the way. However, we are attempting to model reasoning, not distort it. A *move* is a natural notion, and this demands its inclusion.

```
declare PREDCONST MOVES 1[PRE];
```

Section 2.1.1.7 Colors

There remains one basic, though nevertheless trivial sort to be mentioned. Chess is organized as a competitive game; there is not much we could say without recognizing the existence of the two armies, BLACK and WHITE.

```
declare PREDCONST COLORS 1[PRE];
```

Section 2.1.2 Piece Declarations

A large sort hierarchy for pieces is declared, most of which is not used. It is worthwhile mentioning the existence of EMPTY, the piece on any not otherwise occupied square, and that the variables for pieces are the t's (t and t1), whereas the variables for chesspieces are those variables starting with the last three letters of the alphabet (x, y, and z).

The naming scheme for the constant chesspieces might also be mentioned; the encoding is color, side (king's or queen's), column or rank, and the designation p for pawns. Thus, the WKR is the white king's rook, and the BQNP is the black queen's knight pawn.

The function Piececolor, on chesspieces, returns the color of the given chesspiece.

Pieces are represented internally to the FOL simplification mechanism by the atom of the same name as the piece.

The rest of this section is a series of rather monotonous declarations.[34]

```
declare PREDCONST EMPTYPIECE (PIECES) [PRE];
declare PREDCONST WHITEPIECE BLACKPIECE (CHESSPIECES) [PRE];
declare PREDCONST PAWNS BISHOPS KNIGHTS KINGS QUEENS ROOKS
    (CHESSPIECES) [PRE];

declare PREDCONST BPAWNS WPAWNS (PAWNS) [PRE];
declare PREDCONST BBISHOPS WBISHOPS (BISHOPS) [PRE];
declare PREDCONST BKNIGHTS WKNIGHTS (KNIGHTS) [PRE];
declare PREDCONST BROOKS WROOKS (ROOKS) [PRE];
declare PREDCONST BKINGS WKINGS (KINGS) [PRE];
declare PREDCONST BQUEENS WQUEENS (QUEENS) [PRE];

declare OPCONST Piececolor (CHESSPIECES)=COLORS[PRE];

declare INDVAR t t1 ∈ PIECES;
declare INDVAR x x1 x2 x3 x4 y z xa xb xc xd ∈ CHESSPIECES;

declare INDCONST BK ∈ BKINGS,          WK ∈ WKINGS;
declare INDCONST BQ ∈ BQUEENS,         WQ ∈ WQUEENS;
declare INDCONST BKB BQB ∈ BBISHOPS,   WKB WQB ∈ WBISHOPS;
declare INDCONST BKN BQN ∈ BKNIGHTS,   WKN WQN ∈ WKNIGHTS;
declare INDCONST BKR BQR ∈ BROOKS,     WKR WQR ∈ WROOKS;
declare INDCONST WQRP WKRP WQNP WKNP WKBP WQBP WQP WKP ∈ WPAWNS;
declare INDCONST BQRP BKRP BQNP BKNP BKBP BQBP BQP BKP ∈ BPAWNS;
declare INDCONST EMPTY ∈ EMPTYPIECE;

declare INDVAR yb zb ∈ BLACKPIECE,     yyw ∈ WHITEPIECE;
declare INDVAR yk ∈ KINGS,    ywr ywr1 ∈ WROOKS;
declare INDVAR ybi∈ BISHOPS, yn ∈ KNIGHTS,      ywn ∈ WKNIGHTS;
declare INDVAR yp ∈ PAWNS,    ywp ∈ WPAWNS,     ybp ∈ BPAWNS;

mg PIECES≥ (CHESSPIECES,EMPTYPIECE);
mg CHESSPIECES≥
     (WHITEPIECE,BLACKPIECE,PAWNS,BISHOPS,KNIGHTS,KINGS,QUEENS,ROOKS);
mg WHITEPIECE≥ (WPAWNS,WBISHOPS,WKNIGHTS,WKINGS,WQUEENS,WROOKS);
mg BLACKPIECE≥ (BPAWNS,BBISHOPS,BKNIGHTS,BKINGS,BQUEENS,BROOKS);
mg PAWNS≥ (BPAWNS,WPAWNS);
mg BISHOPS≥ (BBISHOPS,WBISHOPS);
mg KNIGHTS≥ (BKNIGHTS,WKNIGHTS);
mg KINGS≥ (BKINGS,WKINGS);
mg QUEENS≥ (BQUEENS,WQUEENS);
mg ROOKS≥ (BROOKS,WROOKS);
```

Here are some attachments for the chess eye. All simplification is done in the (partial) model named CHESS; we shall usually attach (map or associate) atomic primitives to the atom of that name in this model.

---

34. In these declarations, PREDCONST's are predicate constants, OPCONST's operator constants (or functions, if you prefer), INDCONST's, individual constants, and INDVAR's, individual variables. [PRE] application terms can be written without parenthesizing their argument; [INF] terms, between their arguments. It is worthwhile pointing out that while every sort has an infinite collection of variables (theoretically) available to it, we have only declared those variables that we shall actually use.

```
attach BK  ↔ [CHESS] BK;   attach BKP  ↔ [CHESS] BKP;
attach BKB ↔ [CHESS] BKB;  attach BKBP ↔ [CHESS] BKBP;
attach BKN ↔ [CHESS] BKN;  attach BKNP ↔ [CHESS] BKNP;
attach BKR ↔ [CHESS] BKR;  attach BKRP ↔ [CHESS] BKRP;
attach BQ  ↔ [CHESS] BQ;   attach BQP  ↔ [CHESS] BQP;
attach BQB ↔ [CHESS] BQB;  attach BQBP ↔ [CHESS] BQBP;
attach BQN ↔ [CHESS] BQN;  attach BQNP ↔ [CHESS] BQNP;
attach BQR ↔ [CHESS] BQR;  attach BQRP ↔ [CHESS] BQRP;
attach WK  ↔ [CHESS] WK;   attach WKP  ↔ [CHESS] WKP;
attach WKB ↔ [CHESS] WKB;  attach WKBP ↔ [CHESS] WKBP;
attach WKN ↔ [CHESS] WKN;  attach WKNP ↔ [CHESS] WKNP;
attach WKR ↔ [CHESS] WKR;  attach WKRP ↔ [CHESS] WKRP;
attach WQ  ↔ [CHESS] WQ;   attach WQP  ↔ [CHESS] WQP;
attach WQB ↔ [CHESS] WQB;  attach WQBP ↔ [CHESS] WQBP;
attach WQN ↔ [CHESS] WQN;  attach WQNP ↔ [CHESS] WQNP;
attach WQR ↔ [CHESS] WQR;  attach WQRP ↔ [CHESS] WQRP;
attach EMPTY ↔ [CHESS] EMPTY;

attach WPAWNS [CHESS] (DE WPAWNS (x) (MEMQ x
        (QUOTE (WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP))));
attach BPAWNS [CHESS] (DE BPAWNS (x) (MEMQ x
        (QUOTE (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP))));
attach BBISHOPS [CHESS] (DE BBISHOPS (x) (MEMQ x (QUOTE (BKB BQB))));
attach WBISHOPS [CHESS] (DE WBISHOPS (x) (MEMQ x (QUOTE (WKB WQB))));
attach BKNIGHTS [CHESS] (DE BKNIGHTS (x) (MEMQ x (QUOTE (BKN BQN))));
attach WKNIGHTS [CHESS] (DE WKNIGHTS (x) (MEMQ x (QUOTE (WKN WQN))));
attach BROOKS [CHESS] (DE BROOKS (x) (MEMQ x (QUOTE (BKR BQR))));
attach WROOKS [CHESS] (DE WROOKS (x) (MEMQ x (QUOTE (WKR WQR))));
attach BKINGS [CHESS] (DE BKINGS (x) (MEMQ x (QUOTE (BK))));
attach WKINGS [CHESS] (DE WKINGS (x) (MEMQ x (QUOTE (WK))));
attach BQUEENS [CHESS] (DE BQUEENS (x) (MEMQ x (QUOTE (BQ))));
attach WQUEENS [CHESS] (DE WQUEENS (x) (MEMQ x (QUOTE (WQ))));
attach QUEENS [CHESS] (DE QUEENS (x) (MEMQ x (QUOTE (WQ BQ))));
attach ROOKS [CHESS] (DE ROOKS (x) (MEMQ x (QUOTE (BKR WKR BQR WQR))));
attach BISHOPS [CHESS] (DE BISHOPS (x) (MEMQ x (QUOTE (BKB BQB WKB WQB))));
attach KNIGHTS [CHESS] (DE KNIGHTS (x) (MEMQ x (QUOTE (WKN WQN BKN BQN))));
attach KINGS [CHESS] (DE KINGS (x) (MEMQ x (QUOTE (WK BK))));
attach BLACKPIECE [CHESS] (DE BLACKPIECE (x)(MEMQ x (QUOTE
  (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP BKB BQB BKN BQN BKR BQR BK BQ) ));
attach WHITEPIECE [CHESS] (DE WHITEPIECE (x) (MEMQ x (QUOTE
  (WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP WKB WQB WKN WQN WKR WQR WK WQ))));
attach EMPTYPIECE [CHESS] (DE EMPTYPIECE (x) (MEMQ x (QUOTE (EMPTY))));
attach PAWNS [CHESS] (DE PAWNS(x)(MEMQ x (QUOTE
  (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP
   WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP))));
attach CHESSPIECES [CHESS] (DE CHESSPIECES(x)(MEMQ x (QUOTE (BKRP BKNP
   BKBP BKP BQP BQBP BQNP BQRP BKB BQB BKN BQN BKR BQR BK BQ WKRP WKNP
   WKBP WKP WQP WQBP WQNP WQRP WKB WQB WKN WQN WKR WQR WK WQ))));
attach PIECES [CHESS] (DE PIECES (x) (MEMQ x (QUOTE (BKRP BKNP BKBP BKP BQP
   BQBP BQNP BQRP BKB BQB BKN BQN BKR BQR BK BQ EMPTY WKRP WKNP WKBP WKP
   WQP WQBP WQNP WQRP WKB WQB WKN WQN WKR WQR WK WQ))));

attach Piececolor [CHESS→CHESS] (DE Piececolor(x)
    (COND((WHITEPIECE x)(QUOTE WHITE))((BLACKPIECE x)(QUOTE BLACK))));
```

```
extension  BKINGS    (BK);        extension  WKINGS    (WK);
extension  BQUEENS   (BQ);        extension  WQUEENS   (WQ);
extension  BROOKS    (BKR BQR);   extension  WROOKS    (WKR WQR);
extension  BBISHOPS  (BKB BQB);   extension  WBISHOPS  (WKB WQB);
extension  BKNIGHTS  (BKN BQN);   extension  WKNIGHTS  (WKN WQN);
extension  WPAWNS    (WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP);
extension  BPAWNS    (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP);

extension  KINGS     (WK BK);
extension  QUEENS    (WQ BQ);
extension  ROOKS     (BKR WKR BQR WQR);
extension  BISHOPS   (BKB BQB WKB WQB);
extension  KNIGHTS   (WKN WQN BKN BQN);
extension  PAWNS     (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP
                      WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP);

extension  BLACKPIECE  (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP
                        BKB BQB BKN BQN BKR BQR BK BQ);
extension  WHiTEPIECE  (WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP
                        WKB WQB WKN WQN WKR WQR WK WQ);
extension  EMPTYPIECE  (EMPTY);
extension  CHESSPIECES
  (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP BKB BQB BKN BQN BKR BQR BK BQ
   WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP WKB WQB WKN WQN WKR WQR WK WQ);
extension  PIECES
  (BKRP BKNP BKBP BKP BQP BQBP BQNP BQRP BKB BQB BKN BQN BKR BQR BK BQ EMPTY
    WKRP WKNP WKBP WKP WQP WQBP WQNP WQRP WKB WQB WKN WQN WKR WQR WK WQ);
```

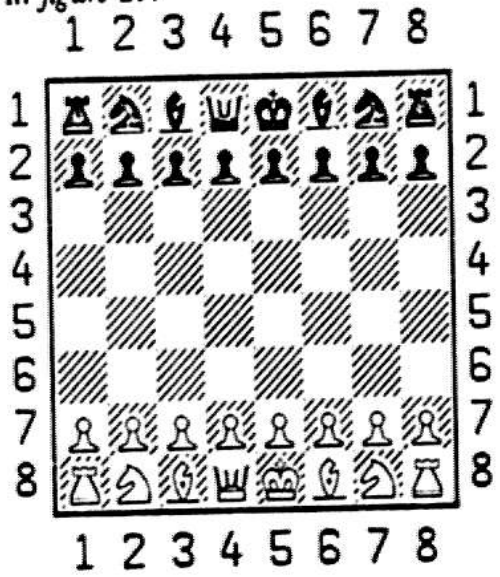
Note that were it not for a small bug in the FOL implementation, the functional definitions of various sorts would not have been required. This has since been corrected.

Section 2.1.3 Squares and Dimensions

Section 2.1.3.1 Square declarations

Squares are represented in FOL as the acronym of the square's name. Thus, the FOL INDCONST for black queen's rook one is BQR1. The perspective (relative to white and black) is chosen to be the nearer side. Black king's knight five (which is also white king's knight four) therefore becomes WKN4. This notation is seen to be a subset of the standard English system.

Internal to the LISP chess model, squares are represented at the dotted pair formed of the squares coordinates (row, column). Thus, BQ1 in FOL (black queen's one) is (1 . 4) to the LISP model. The coordinates used are illustrated in *figure 25*.



The LISP internal numbering scheme for squares.

figure 25

There are also several sub-species of squares. We identify the WHITESQUARES and BLACKSQUARES, by the traditional checkerboard pattern of the squares. Squares are also specialized by their row. We thereby acheive sorts such as WLASTRANK and BLASTRANK (white and black's last rank (row)).

```
declare PREDCONST WHITESQUARES BLACKSQUARES BLASTRANK WLASTRANK
    (SQUARES) [PRE];

declare INDVAR sqx sq sq1 sq2 sq3 sq4 sq5 sq6 sq7 sq8 ‹ SQUARES;
declare INDCONST
BQR1 BQN1 BQB1 BQ1 BK1 BKB1 BKN1 BKR1 BQR2 BQN2 BQB2 BQ2 BK2 BKB2 BKN2 BKR2
BQR3 BQN3 BQB3 BQ3 BK3 BKB3 BKN3 BKR3 BQR4 BQN4 BQB4 BQ4 BK4 BKB4 BKN4 BKR4
WQR4 WQN4 WQB4 WQ4 WK4 WKB4 WKN4 WKR4 WQR3 WQN3 WQB3 WQ3 WK3 WKB3 WKN3 WKR3
WQR2 WQN2 WQB2 WQ2 WK2 WKB2 WKN2 WKR2 WQR1 WQN1 WQB1 WQ1 WK1 WKB1 WKN1 WKR1
        ‹ SQUARES;

mg EXSQUARES≥ (SQUARES);
mg SQUARES≥ (WHITESQUARES, BLACKSQUARES, WLASTRANK, BLASTRANK);
```

And the various sorts have the obvious attachments.

```
attach SQUARES [CHESS] (DE SQUARES(x)
   (AND (NOT (ATOM x))
        (NUMBERP(CDR x)) (GREATERP (CDR x)0) (LESSP (CDR x) 9)
        (NUMBERP(CAR x)) (GREATERP (CAR x)0) (LESSP (CAR x) 9)));
attach BLASTRANK [CHESS] (LAMBDA (x)(EQ 8 (CAR x)));
attach WLASTRANK [CHESS] (LAMBDA (x)(EQ 1 (CAR x)));
```

```
attach WHITESQUARES [CHESS] (LAMBDA(S)(AND (SQUARES S)
                                    (ZEROP(REMAINDER(PLUS(CAR S)(COR S))2))));
attach BLACKSQUARES [CHESS] (LAMBDA(S)(AND (SQUARES S)
                                  (NOT(ZEROP(REMAINDER(PLUS(CAR S)(COR S))2)))));

extension SQUARES (BQR1 BQN1 BQB1 BQ1 BK1 BKB1 BKN1 BKR1 BQR2 BQN2 BQB2
BQ2 BK2 BKB2 BKN2 BKR2 BQR3 BQN3 BQB3 BQ3 BK3 BKB3 BKN3 BKR3 BQR4 BQN4 BQB4
BQ4 BK4 BKB4 BKN4 BKR4 WQR4 WQN4 WQB4 WQ4 WK4 WKB4 WKN4 WKR4 WQR3 WQN3 WQB3
WQ3 WK3 WKB3 WKN3 WKR3 WQR2 WQN2 WQB2 WQ2 WK2 WKB2 WKN2 WKR2 WQR1 WQN1 WQB1
WQ1 WK1 WKB1 WKN1 WKR1);

attach BQR1↔[CHESS](1.1);attach BQN1↔[CHESS](1.2);attach BQB1↔[CHESS](1.3);
attach BQ1 ↔[CHESS](1.4);attach BK1 ↔[CHESS](1.5);attach BKB1↔[CHESS](1.6);
attach BKN1↔[CHESS](1.7);attach BKR1↔[CHESS](1.8);attach BQR2↔[CHESS](2.1);
attach BQN2↔[CHESS](2.2);attach BQB2↔[CHESS](2.3);attach BQ2 ↔[CHESS](2.4);
attach BK2 ↔[CHESS](2.5);attach BKB2↔[CHESS](2.6);attach BKN2↔[CHESS](2.7);
attach BKR2↔[CHESS](2.8);attach BQR3↔[CHESS](3.1);attach BQN3↔[CHESS](3.2);
attach BQB3↔[CHESS](3.3);attach BQ3 ↔[CHESS](3.4);attach BK3 ↔[CHESS](3.5);
attach BKB3↔[CHESS](3.6);attach BKN3↔[CHESS](3.7);attach BKR3↔[CHESS](3.8);
attach BQR4↔[CHESS](4.1);attach BQN4↔[CHESS](4.2);attach BQB4↔[CHESS](4.3);
attach BQ4 ↔[CHESS](4.4);attach BK4 ↔[CHESS](4.5);attach BKB4↔[CHESS](4.6);
attach BKN4↔[CHESS](4.7);attach BKR4↔[CHESS](4.8);attach WQR4↔[CHESS](5.1);
attach WQN4↔[CHESS](5.2);attach WQB4↔[CHESS](5.3);attach WQ4 ↔[CHESS](5.4);
attach WK4 ↔[CHESS](5.5);attach WKB4↔[CHESS](5.6);attach WKN4↔[CHESS](5.7);
attach WKR4↔[CHESS](5.8);attach WQR3↔[CHESS](6.1);attach WQN3↔[CHESS](6.2);
attach WQB3↔[CHESS](6.3);attach WQ3 ↔[CHESS](6.4);attach WK3 ↔[CHESS](6.5);
attach WKB3↔[CHESS](6.6);attach WKN3↔[CHESS](6.7);attach WKR3↔[CHESS](6.8);
attach WQR2↔[CHESS](7.1);attach WQN2↔[CHESS](7.2);attach WQB2↔[CHESS](7.3);
attach WQ2 ↔[CHESS](7.4);attach WK2 ↔[CHESS](7.5);attach WKB2↔[CHESS](7.6);
attach WKN2↔[CHESS](7.7);attach WKR2↔[CHESS](7.8);attach WQR1↔[CHESS](8.1);
attach WQN1↔[CHESS](8.2);attach WQB1↔[CHESS](8.3);attach WQ1 ↔[CHESS](8.4);
attach WK1 ↔[CHESS](8.5);attach WKB1↔[CHESS](8.6);attach WKN1↔[CHESS](8.7);
attach WKR1↔[CHESS](8.8);
```

The predicates LASTRANKER (is square the last rank (pawn promotion rank) of the given color), SAMEDIAG (are the arguments on the same diagonal), and SQUARE_BETWEEN (is the middle argument between the other arguments, either orthogonally or diagonally) are also declared. Attachments are provided for the latter two.

```
declare PREDCONST LASTRANKER (SQUARES COLORS);
declare PREDCONST SAMEDIAG (SQUARES,SQUARES);
declare PREDCONST SQUARE_BETWEEN (SQUARES,SQUARES,SQUARES);

attach SAMEDIAG [CHESS,CHESS] (DE SAMEDIAG (x y)(AND
        (SQUARES x)(SQUARES y)(NOT(EQUAL x y))
        (EQ(ABS(DIFFERENCE (CAR x)(CAR y)))
           (ABS(DIFFERENCE (COR x)(COR y))))));

attach SQUARE_BETWEEN [CHESS,CHESS,CHESS] (DE SQUARE_BETWEEN(q r S)
(AND (SQUARES q)(SQUARES r)(SQUARES S)(OR
(AND (EQ (CAR q)(CAR r))(EQ (CAR r)(CAR S))(BETWEEN (COR q)(COR r)(COR S)))
(AND (EQ (COR q)(COR r))(EQ (COR r)(COR S))(BETWEEN (CAR q)(CAR r)(CAR S)))
(AND(SAMEDIAG q r)(SAMEDIAG q S)(SAMEDIAG r S)(BETWEEN(CAR q)(CAR r)(CAR S)
)))));
```

Section 2.1.3.2 Coordinate Declarations

We will also have occasion to refer to the individual coordinates of particular squares, and to prove lemmas about these coordinates. We call the class of square coordinates *dimensions*, and speak of the *row* and *column* of a particular square. The numbering scheme for rows and columns corresponds to the numbering in the internal LISP model. This, we might axiomatically have, *If Square is in White's last row then its row is equal to 1.* Dimensions are represented in the LISP model as natural numbers.

A compositor, Makesquare, for taking a row-column pair, and producing the appropriate square, is also declared. This compositor is stated to be equivalent to the LISP function CONS in the computational model.

```
declare PREDCONST ISDIMENSION (NATNUM)[PRE];
declare PREDCONST ISROW ISCOLUMN (ISDIMENSION)[PRE];
declare PREDCONST BLASTROW WLASTROW (ISROW);

represent  (ISDIMENSION ISROW ISCOLUMN BLASTROW WLASTROW) as NATNUMREP;

declare OPCONST Row (SQUARES)=ISROW[PRE];
declare OPCONST Column (SQUARES)=ISCOLUMN[PRE];
declare OPCONST Makesquare (ISROW ISCOLUMN)=SQUARES;

declare PREDCONST IS_EVEN (ISROW,ISCOLUMN);

declare INDVAR dx dx1 dx2 <ISDIMENSION;
declare INDVAR drx drx1 drx2 < ISROW;
declare INDVAR dcx dcx1 dcx2 < ISCOLUMN;

mg ISDIMENSION≥(ISROW,ISCOLUMN);
mg NATNUM≥(ISDIMENSION);
```

Successor functions are defined on the rows, succession being relative to the moving side. A black pawn in row drx moves to row BSUC(drx) on his next (single square) move. The last row is the pawn promotion row.

```
declare PREDCONST BSUC WSUC (ISROW,ISROW);
declare PREDCONST BETWEEN (ISDIMENSION,ISDIMENSION,ISDIMENSION);
declare OPCONST Bsucf Wsucf (ISROW)=ISROW[PRE];
```

It should be noted that the operators Bsucf, Wsucf (and, similarly L2touchf and R2touchf, section 2.2.1.2) are functions of convenience, not definition. There are no axioms that mention these functions. However, we can (and do) use the simplification mechanism to compute the value of these functions in every (interesting) case, and thereby produce useful inference steps involving their use.

Attachments to implement rows, columns and successors.

```
extension ISROW (1,2,3,4,5,6,7,8);
extension ISCOLUMN (1,2,3,4,5,6,7,8);
extension ISDIMENSION (1,2,3,4,5,6,7,8);

attach Row (CHESS→NATNUMREP) CAR;
attach Column (CHESS→NATNUMREP) COR;
attach Makesquare (NATNUMREP,NATNUMREP→CHESS) CONS;
attach ISDIMENSION (NATNUMREP) (DE ISDIMENSION(x)
        (AND(NUMBERP x)(LESSP x 9)(GREATERP x 0)));
attach ISROW    (NATNUMREP) ISDIMENSION;
attach ISCOLUMN (NATNUMREP) ISDIMENSION;
attach Bsucf (NATNUMREP→NATNUMREP) (DE Bsucf(r)(COND((EQ r 8)8)(T(ADD1 r))));
attach Wsucf (NATNUMREP→NATNUMREP) (DE Wsucf(r)(COND((EQ r 1)1)(T(SUB1 r))));
attach BLASTROW (NATNUMREP) (DE BLASTROW(r)(EQ r 8));
attach WLASTROW (NATNUMREP) (DE WLASTROW(r)(EQ r 1));
attach BETWEEN (NATNUMREP,NATNUMREP,NATNUMREP) (DE BETWEEN (x y z)
    (AND (NUMBERP x) (NUMBERP y) (NUMBERP z)
         (OR (AND (LESSP x y)(LESSP y z))
             (AND (LESSP z y)(LESSP y x)))));
```

Notice that we can easily *observe* (simplify) that the predicates ISROW, ISCOLUMN and ISDIMENSION are *equivalent*. However, we find it more natural to retain the distinction, for, after all, rows and columns are hardly equivalent in their chess interpretations.

Section 2.1.4 Value Declarations

There are fourteen VALUES in this system, corresponding to the twelve different incarnations of the chessmen on the chessboard, an empty value, and an undefined value. It is perhaps worthwhile to emphasize that the value of a given chesspiece is a function of the position in which we are considering that chesspiece. Of course, the value of non-pawn pieces does not change during a game (and we shall prove a theorem to that effect (section 3.3.1).

Chessboards, being a manifestation of the current situation in a chess game, rather than a description of the history of that game, have values filling their squares. Our desire to have partially defined chessboards leads to the existence of the undefined value (UD) of our system.

The naming scheme for values is the converse of that of pieces. Thus, QW is a value of any piece that is a white queen. In competitive chess, a promoted white pawn would therefore be likely to have the value QW after his promotion. Value variables begin with the letter v. Each value is represented in the internal LISP world as the atom of the same name.

```
declare PREDCONST VVALUES NVALUES (VALUES) [PRE];
declare PREDCONST PIECEVALUES EVALUES (VVALUES) [PRE];
declare PREDCONST WVALUES BVALUES (PIECEVALUES) [PRE];
declare PREDCONST PROMVALUES VALUEK VALUEQ VALUEB VALUEP
                  VALUER VALUEN (PIECEVALUES) [PRE];

declare INDCONST KW QW BW NW RW PW < WVALUES ;
declare INDCONST KB QB BB NB RB PB < BVALUES;
declare INDCONST MT < EVALUES, UD < NVALUES;
```

```
declare INDVAR vw ∈ WVALUES, vb ∈ BVALUES;
declare INDVAR v v1 ∈ VALUES, vpc vpc1 vpc2 ∈ PIECEVALUES;
declare INDVAR vvx ∈ VVALUES;
declare INDVAR vbi ∈ VALUEB;


attach MT ↔ [CHESS] MT; attach NB ↔ [CHESS] NB; attach KW ↔ [CHESS] KW;
attach RW ↔ [CHESS] RW; attach PB ↔ [CHESS] PB; attach PW ↔ [CHESS] PW;
attach KB ↔ [CHESS] KB; attach RB ↔ [CHESS] RB; attach QB ↔ [CHESS] QB;
attach QW ↔ [CHESS] QW; attach UD ↔ [CHESS] UD; attach BB ↔ [CHESS] BB;
attach NW ↔ [CHESS] NW; attach BW ↔ [CHESS] BW;

mg PIECEVALUES≥
    (PROMVALUES,WVALUES,BVALUES,VALUEK,VALUEQ,VALUEB,VALUEN,VALUER,VALUEP);
mg PROMVALUES≥  (VALUEQ,VALUEB,VALUEN,VALUER);
mg VVALUES≥(PIECEVALUES,EVALUES);
mg VALUES≥(NVALUES,VVALUES);

extension VALUEK (KW KB); extension VALUEQ (QW QB);
extension VALUEB (BW BB); extension VALUEN (NW NB);
extension VALUER (RW RB); extension VALUEP (PW PB);
extension WVALUES (KW,QW,BW,NW,RW,PW);
extension BVALUES (KB,QB,BB,NB,RB,PB);
extension EVALUES (MT); extension NVALUES (UD);
extension PIECEVALUES (KB QB RB BB NB PB KW QW RW BW NW PW);
extension VVALUES (MT KB QB RB BB NB PB KW QW RW BW NW PW);
extension VALUES (UD MT KB QB RB BB NB PB KW QW RW BW NW PW);
extension PROMVALUES (QB RB BB NB QW RW BW NW);

attach VALUEK [CHESS] (DE VALUEK (x) (MEMQ x (QUOTE (KW KB))));
attach VALUEQ [CHESS] (DE VALUEQ (x) (MEMQ x (QUOTE (QW QB))));
attach VALUEB [CHESS] (DE VALUEB (x) (MEMQ x (QUOTE (BW BB))));
attach VALUEN [CHESS] (DE VALUEN (x) (MEMQ x (QUOTE (NW NB))));
attach VALUER [CHESS] (DE VALUER (x) (MEMQ x (QUOTE (RW RB))));
attach VALUEP [CHESS] (DE VALUEP (x) (MEMQ x (QUOTE (PW PB))));
attach WVALUES [CHESS] (DE WVALUES (x) (MEMQ x(QUOTE(KW QW BW NW RW PW))));
attach BVALUES [CHESS] (DE BVALUES (x) (MEMQ x(QUOTE(KB QB BB NB RB PB))));
attach EVALUES [CHESS] (DE EVALUES (x) (EQ x (QUOTE MT)));
attach NVALUES [CHESS] (DE NVALUES (x) (EQ x (QUOTE UD)));
attach VVALUES [CHESS] (DE VVALUES (x)
                              (OR(BVALUES x)(WVALUES x)(EQ x(QUOTE MT))));
attach PROMVALUES [CHESS] (DE PROMVALUES (x)
    (MEMQ x(QUOTE(QB RB NB BB QW RW NW BW))));
attach PIECEVALUES [CHESS] (DE PIECEVALUES (x)(OR (BVALUES x)(WVALUES x)));
attach VALUES [CHESS] (DE VALUES (x) (OR (VVALUES x)(EQ x (QUOTE UD))));
```

PROMVALUES are the values a pawn can promote to. More specifically, a pawn can promote to be a queen, rook, bishop or knight.

The Valuecolor of any PIECEVALUES is the color of that value. Thus, the Valuecolor of KW is WHITE.

```
declare OPCONST Valuecolor (PIECEVALUES)→COLORS[PRE];

attach Valuecolor [CHESS→CHESS] (DE Valuecolor(v)(COND
     ((WVALUES v)(QUOTE WHITE))((BVALUES v)(QUOTE BLACK))));
```

Section 2.1.5 Board Declarations

We have several interesting functions and predicates defined on boards. Two of the most complex are the predicates WHITEINCHECK and BLACKINCHECK. These are true when the given side is in check on the given board.[35]

Similarly, we have the composite predicate, SIDEINCHECK, on boards and colors. SIDEINCHECK on WHITE and a board is true if and only if WHITEINCHECK is true for that board. The corresponding statement about BLACK and BLACKINCHECK also holds.

Since a position is a state vector, we are theoretically able to obtain the total board (board with no undefined squares) of any position. The function which extracts that board is Tboard. However, as a position is almost invariably a variable (rather than a constant), we will never actually compute the Tboard of any position.

One board is a SUBOARD of another if the second is equal to the first, on every square the first is not undefined (UD). SUBOARD is therefore a partial ordering relation on boards. We state that the predicate BOARD, on positions and boards, is true if the given board is a SUBOARD of the Tboard (total board) of that position. Thus, this predicate is true if the undefined squares of the given board could be filled in to make the board obtained by playing the game that the position defines.[36] The predicate BOARD is particularly appropriate for the kinds of puzzles we solve. Typically, we shall be presented a board or board fraction, and need to reason about any POSITION which has this board fragment as one of its boards.

We also have a constructor for boards, Makeboard, which takes a board, a square and a value, and constructs the new board formed by inserting that value on the stated square.

The function Valueon, on boards and squares, returns the value on that square of that board. The predicate MOVETO, on boards, values, squares and squares, is true if the given value could move, on the given board, from the first square to the second. MOVETO encompasses our notion of *ordinary* movement. If the piece in question is, for example, a rook, then MOVETO will be true for that piece and board, if, the two squares share a row or column (but not both), and every square between them is unoccupied (MT, not UD).

```
declare PREDCONST WHITEINCHECK BLACKINCHECK (BOARDS) [PRE];
declare PREDCONST BOARD (POSITIONS,BOARDS);

.group
declare OPCONST Tboard (POSITIONS)=TOTALBOARDS[PRE];
declare OPCONST Valueon (BOARDS,SQUARES)=VALUES;

declare INDVAR a b b1 b2 b3 ∈ BOARDS, bt ∈ TOTALBOARDS;

declare PREDCONST SIDEINCHECK (BOARDS,COLORS);
declare PREDCONST MOVETO (BOARDS,VALUES,SQUARES,SQUARES);

mg BOARDS≥(TOTALBOARDS,WHITEINCHECK,BLACKINCHECK);
```

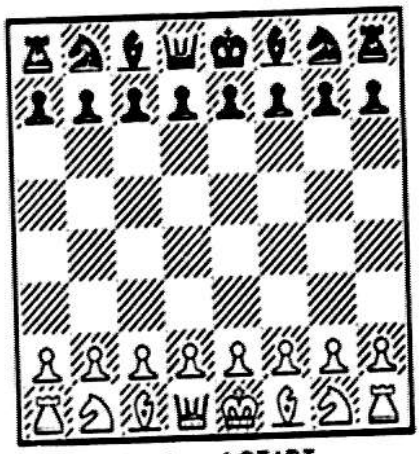
---

35. The attachments to these predicates are in section 2.2.2.

36. Therefore, (and trivially) the totally undefined board is a BOARD of every position.

```
declare OPCONST Makeboard (BOARDS,SQUARES,VALUES)=BOARDS;
declare PREDCONST SUBOARD (BOARDS,BOARDS);
```

We shall call initial board, the configuration of pieces before the start of the game, START. This board is illustrated in *figure 26*.



The board START.

figure 26

```
declare INDCONST START ∈ TOTALBOARDS;
```

We represent a board in the internal LISP system as a list of the eight rows, each row being a list of the eight values on it. This is illustrated in *figure 27*.

```
((BQR1 BQN1 BQB1 BQ1 BK1 BKB1 BKN1 BKR1)
 (BQR2 BQN2 BQB2 BQ2 BK2 BKB2 BKN2 BKR2)
 (BQR3 BQN3 BQB3 BQ3 BK3 BKB3 BKN3 BKR3)
 (BQR4 BQN4 BQB4 BQ4 BK4 BKB4 BKN4 BKR4)
 (WQR4 WQN4 WQB4 WQ4 WK4 WKB4 WKN4 WKR4)
 (WQR3 WQN3 WQB3 WQ3 WK3 WKB3 WKN3 WKR3)
 (WQR2 WQN2 WQB2 WQ2 WK2 WKB2 WKN2 WKR2)
 (WQR1 WQN1 WQB1 WQ1 WK1 WKB1 WKN1 WKR1))
```

LISP arrangement of a board, with square locations

figure 27

With this representation in mind, we make the appropriate attachments.

```
attach START ↔ [CHESS] ((RB NB BB QB KB BB NB RB)
                        (PB PB PB PB PB PB PB PB)
                        (MT MT MT MT MT MT MT MT)
                        (MT MT MT MT MT MT MT MT)
                        (MT MT MT MT MT MT MT MT)
                        (MT MT MT MT MT MT MT MT)
                        (PW PW PW PW PW PW PW PW)
                        (RW NW BW QW KW BW NW RW));

attach Valueon [CHESS,CHESS→CHESS]
  (DE Valueon (b S)(CAR (NTH (CAR (NTH b (CAR S))) (CDR S))));

attach BOARDS [CHESS] (DE BOARDS(b) (AND (EQ (LENGTH b) 8)(ALLROWS  b )));
FUNCTION (DE ALLROWS (b) (COND ((NULL b) T)
      ((AND (EQ (LENGTH (CAR b)) 8)(MEMBOARD (CAR b))(ALLROWS (CDR b))))));
FUNCTION (DE MEMBOARD (Row)(COND ((NULL Row) T)
      ((AND (VALUES (CAR Row))(MEMBOARD (CDR Row))))));
attach TOTALBOARDS [CHESS]
       (DE TOTALBOARDS (b) (AND (EQ (LENGTH b) 8)(ALLTROWS b)));
FUNCTION(DE ALLTROWS (b) (COND  ((NULL b) T)((AND (EQ (LENGTH (CAR b)) 8)
       (MEMTBOARD (CAR b))(ALLTROWS (CDR b))))));
FUNCTION (DE MEMTBOARD (Row)(COND ((NULL Row) T)
       ((AND (VVALUES (CAR Row))(MEMTBOARD (CDR Row))))));
attach SUBOARD [CHESS,CHESS] (DE SUBOARD (a b)(COND
  ((NULL a)T)
  ((EQUAL (CAR a)(CAR b))(SUBOARD (CDR a)(CDR b)))
  ((OR (EQ (CAAR a)(CAAR b))
       (EQ (CAAR a) (QUOTE UD)))
   (SUBOARD (CONS (CDAR a)(CDR a))
       (CONS (CDAR b)(CDR b))))));
```

Section 2.1.6 Color Declarations

We develop a much richer set of predicates and variables on colors than a two element sort deserves.

```
declare PREDCONST WHT BLK (COLORS)[PRE];
declare INDCONST WHITE ∈ WHT, BLACK ∈ BLK;
declare INDVAR   c ∈ COLORS;

mg COLORS ≥(WHT,BLK);

extension BLK (BLACK);
extension WHT (WHITE);
extension COLORS WHT ∪ BLK;

attach BLACK ↔ [CHESS] BLACK;
attach WHITE ↔ [CHESS] WHITE;
attach WHT [CHESS] (DE WHT(c)(EQ c (QUOTE WHITE)));
attach BLK [CHESS] (DE BLK(c)(EQ c (QUOTE BLACK)));
```

Section 2.1.7 More on Positions

Section 2.1.7.1 Position declarations

It is worthwhile to emphasize that these chess axioms apply only to situations that might arise in a legal game. Just as formal logic is very sensitive to inconsistency, allowing a proof of any WFF from a false premise, so these axioms, when presented with, for example, an impossible board, do not know which of their axioms to doubt, and will permit the proof of any conclusion about that board. Therefore, the use of the word *position* in this paper should be understood to mean *legal position*. If it were necessary to consider *almost legal positions*, then these axioms could be suitable subverted to reflect whatever the subverter felt were the more fundamental legalities (see section 5.2.1).

Our system recognizes another major distinction between positions. For almost every position, one can speak of the move that was made to get to that position. The exception, of course, is the initial position, the position before the game begins. We therefore have the subsort of GAMEPOSITION, which is every position except the initial one.

We refer to the initial position as P0. Its LISP representation is as the list whose only element is the (arbitrarily selected) atom STARTING.

Positions also naturally, and somewhat more evenly, divide themselves by the color of the player who is to move next in that position. We therefore have the subsort of WHITETURN, those positions for which white is on move. Consistent with the rules of chess, P0 will be a WHITETURN position.

The variables r, r1 and r2 are over the domain of POSITIONS. All variables beginning with the letters p and q range over the GAMEPOSITIONS.

```
declare PREDCONST WHITETURN GAMEPOSITION (POSITIONS) [PRE];
mg POSITIONS≥ (WHITETURN, GAMEPOSITION);

declare INDCONST P0 ∈ POSITIONS;
attach P0 ↔ [CHESS] (STARTING);

declare INDVAR r r1 r2 ∈POSITIONS;
declare INDVAR p q p1 p2 p3 p4 px py pz qx qy qz q1 q2 q3 ∈ GAMEPOSITION;
```

We speak of one position as being a SUCCESSOR to another if there is a legal move from the first to the second. We also recognize the function which takes a position, and returns the previous position (position prior to the last move), Prevpos. Thus, for all GAMEPOSITIONs, p, SUCCESSOR (Prevpos p, p) will be true.

```
declare PREDCONST SUCCESSOR (POSITIONS, GAMEPOSITION);
declare OPCONST Prevpos (GAMEPOSITION)→POSITIONS [PRE];
```

As positions are conceptually built of moves, we have the function Move, on GAMEPOSITIONs, which extracts the last move made to get to that position. A compositor, Nextpos, on moves and positions, yielding the ALLPOSITION obtained by making that move, is also provided. Two things should be noted about this function. It produces elements of the sort ALLPOSITION, which includes both "legal" and "illegal" positions, depending upon whether the given move was legal in the argument position. Secondly, and perhaps more germanely, we are dealing exclusively with retrograde analysis chess; the function Nextpos and sort ALLPOSITION are nowhere used in the following proofs.

```
declare OPCONST Move (GAMEPOSITION)=MOVES[PRE];
declare OPCONST Nextpos (MOVES,POSITIONS)=ALLPOSITION;
```

One position is PREDEGAME (predecessor in this game) to another if the first occurred in the game played to reach the second. We will also use similar kinship terms, such as *ancestor* and *descendant* in describing positions played in the same game. The initial position is, of course, a predecessor to every GAMEPOSITION. It is seen, therefore, that POSITIONS form themselves into a tree, with P0 at the root, with respect to Prevpos operator.

```
declare PREDCONST PREDEGAME (POSITIONS,GAMEPOSITION);
```

In going from one position to a successor position, one can employ one of three different moves -- a *castle*, a *capture en passant*, or a *simple, legal move*.[37] Castles are distinguished by moving two pieces with the same move, en passant capture by the capture of a piece on a square other than the one moved to.

```
declare PREDCONST SIMPLELEGALMOVE EN_PASSANT CASTLING
                        (POSITIONS,GAMEPOSITION);
```

Another useful predicate on positions and colors is POSITIONINCHECK. If the given color is in check in the stated position, this predicate is true.

```
declare PREDCONST POSITIONINCHECK (POSITIONS,COLORS);
```

As positions are state variables, it is possible to extract information about the status of individual chesspieces or squares from them. The function Pos says which piece is on a given square at a given time. Its *almost* inverse is Pospcf (position-piece function) which takes a position and a chesspiece, and returns the extended square occupied by that piece[38] One can also ask for the value of a piece in a position (Val) or the color of the position itself (Color), a WHITETURN position having a WHITE color.

```
declare OPCONST Pos (POSITIONS,SQUARES)=PIECES;
declare OPCONST Pospcf (POSITIONS,CHESSPIECES)=EXSQUARES;
declare OPCONST Val (POSITIONS,PIECES)=VVALUES;
declare OPCONST Color (POSITIONS)=COLORS[PRE];
```

The predicate PROMOTEDPAWN is true if the argument pawn has been promoted before or by the given position (no longer has a pawn value.)

```
declare PREDCONST PROMOTEDPAWN (GAMEPOSITION,PAWNS);
```

Section 2.1.7.2 Positional Attachments

The following attachments have been made to the position predicates.[39] These functions are, to the minimal extent that they have been implemented, the *obvious* attachments for handling the objects in

---

37. A general classification, meant to subsume everything else.

38. The piece may, of course, no longer be on any "real" square. In that case, the value of the function is not defined.

39. Note the use of the I_DONT_KNOW response (a special FOL construct) when the author didn't feel like writing the rest of these complicated functions.

the world we have defined. The notion of a position as a list of moves has been minimally incorporated into these attachments. The attachment for POSITIONS is only used to uniformly recognize the initial position. We will construct no other positions, for we will not be speaking about any other entire game.

```
attach POSITIONS [CHESS] (DE POSITIONS (L)(COND
    ((EQUAL (QUOTE (STARTING)) L)) (T (QUOTE I_DONT_KNOW)) ));

attach Pos [CHESS,CHESS→CHESS]
   (LAMBDA (p S)(COND((EQUAL p (QUOTE (STARTING)))(GIVENF S))
                              (T (QUOTE I_DONT_KNOW)) ));
attach Pospcf [CHESS,CHESS→CHESS]
   (LAMBDA (p x)(COND((EQUAL p (QUOTE (STARTING)))(GIVENPCF x))
                              (T (QUOTE I_DONT_KNOW)) ));

attach WHITETURN[CHESS](DE WHITETURN(L)(NOT(ZEROP(REMAINDER(LENGTH L)2))));
attach Color [CHESS→CHESS]
    (DE Color(L)(COND((WHITETURN L)(QUOTE WHITE))(T(QUOTE BLACK))));

FUNCTION (DE GIVENF (S)(COND
((EQ (CAR S) 1)(CAR (NTH (QUOTE (BQR BQN BQB BQ BK BKB BKN BKR)) (CDR S))))
((EQ (CAR S) 2)(CAR (NTH
                  (QUOTE (BQRP BQNP BQBP BQP BKP BKBP BKNP BKRP))(CDR S))))
((EQ (CAR S) 7)(CAR (NTH
                  (QUOTE (WQRP WQNP WQBP WQP WKP WKBP WKNP WKRP)) (CDR S))))
((EQ (CAR S) 8)(CAR (NTH (QUOTE (WQR WQN WQB WQ WK WKB WKN WKR)) (CDR S))))
(T (QUOTE EMPTY))));
FUNCTION (DE GIVENPCF (x)(CADR (ASSOC x (QUOTE(
     (BQR (1 . 1)) (BQN (1 . 2)) (BQB (1 . 3)) (BQ (1 . 4))
     (BK (1 . 5)) (BKB (1 . 6)) (BKN (1 . 7)) (BKR (1 . 8))
     (BQRP (2 . 1)) (BQNP (2 . 2)) (BQBP (2 . 3)) (BQP (2 . 4))
     (BKP (2 . 5)) (BKBP (2 . 6)) (BKNP (2 . 7)) (BKRP (2 . 8))
     (WQRP (7 . 1)) (WQNP (7 . 2)) (WQBP (7 . 3)) (WQP (7 . 4))
     (WKP (7 . 5)) (WKBP (7 . 6)) (WKNP (7 . 7)) (WKRP (7 . 8))
     (WQR (8 . 1)) (WQN (8 . 2)) (WQB (8 . 3)) (WQ (8 . 4))
     (WK (8 . 5)) (WKB (8 . 6)) (WKN (8 . 7)) (WKR (8 . 8)))))));
attach Val [CHESS,CHESS→CHESS] (DE Val(p x)(COND((EQUAL p(QUOTE(STARTING)))
  (COND ((SETQ TEMPORARYXXX (ASSOC x (QUOTE
((WK.KW) (WKN.NW) (WKB.BW) (WKR.RW) (WQ.QW) (WQN.NW) (WQB.BW) (WQR.RW)
 (WQP.PW) (WQNP.PW) (WQBP.PW) (WQRP.PW) (WKP.PW) (WKNP.PW) (WKBP.PW) (WKRP.PW)
 (BK.KB) (BKN.NB) (BKB.BB) (BKR.RB) (BQ.QB) (BQN.NB) (BQB.BB) (BQR.RB)
 (BQP.PB) (BQNP.PB) (BQBP.PB) (BQRP.PB) (BKP.PB) (BKNP.PB) (BKBP.PB) (BKRP.PB)
 (EMPTY.MT)))))(CDR TEMPORARYXXX))) )
 (T(QUOTE I_DONT_KNOW)) ));
```

Section 2.1.8 Move Declarations

Section 2.1.8.1 Predicates on Moves

The sort of MOVES is redundant in the axioms and proof, replacable by the positions themselves. However, the notion of a *move* is a natural concept in itself, and was therefore included in the axiomatization.

There are, of course, various kinds of moves. For example, we can classify the last position by whether it was an en passant capture, castle or ordinary move:

```
declare PREDCONST ORDINARY CASTLE ENPASSANT (MOVES)[PRE];
```

There are several kinds of ordinary moves. They divide between the capturing and non-capturing moves (CAPTURE and SIMPLE), and may also fall into the pawn promotions (PAWNPROM). Compound classifications such as simple pawn promotion (SIMPP) or captures that are not promotion moves (CAP). The predconst TAKINGS covers all capture moves, including en passant capture. This rich structure uses the following declarations:

```
declare PREDCONST PAWNPROM TAKINGS (MOVES)[PRE];
declare PREDCONST CAPTURE SIMPLE (ORDINARY)[PRE];
declare PREDCONST SIM SIMPP (SIMPLE)[PRE];
declare PREDCONST CAP CAPPP (CAPTURE)[PRE];
mg CAPTURE ≥ {CAP CAPPP};
mg SIMPLE ≥ {SIM SIMPP};
mg PAWNPROM ≥ {SIMPP CAPPP};
mg ORDINARY ≥ {PAWNPROM CAPTURE SIMPLE};
mg TAKINGS ≥ {CAPTURE ENPASSANT};
mg MOVES ≥ {ORDINARY CASTLE ENPASSANT TAKINGS};
```

And, of course, each of these sorts needs a variable to call its own.

```
declare INDVAR m ∈ MOVES, mc ∈ CAPTURE, mo ∈ ORDINARY, mpp ∈ PAWNPROM,
        ms ∈ SIMPLE,  mtx ∈ TAKINGS, mspp ∈ SIMPP;
```

Section 2.1.8.2 Functions on Moves

For all moves, we can speak of the square from which the move was made, the square to which it was made, and the chesspiece that did the moving. For certain other classes of moves, we can state the chesspiece captured, the value a pawn promoted to, how the rook of a castling move moved, or where an en passant capture took place. Collectively, these produce the following declarations.

```
declare OPCONST From To (MOVES) → SQUARES [PRE];
declare OPCONST Mover (MOVES) → CHESSPIECES [PRE];
declare OPCONST Taken (TAKINGS) → CHESSPIECES [PRE];
declare OPCONST Promoted (PAWNPROM) → PROMVALUES [PRE];
declare OPCONST Alsofrom Alsoto (CASTLE) → SQUARES [PRE];
declare OPCONST Alsomover (CASTLE) → CHESSPIECES [PRE];
declare OPCONST Takenon (ENPASSANT) → SQUARES [PRE];
```

It should be noted that the *ALSO* part of the castling move functions refer to the actions of the rook in the castling move.

We can also have *move constructor functions*, which take the various determiners of a move, and produce the move corresponding to those requirements. For example, a simple (SIM) move constructor would be declared:

```
declare OPCONST Makesimplemove (SQUARES,SQUARES,CHESSPIECES)→SIM;
```

And would produce the move resulting from that chesspiece moving from the first square to the second. However, as we never construct any moves, we will not need these constructors.

Section 2.1.9 Definitional Axioms

Section 2.1.9.1 Miscellaneous axioms

In the last several sections, we have defined several predicates and functions in terms of other predicates and functions. FOL does not, of course, know about these relationships unless we explicitly axiomatize them. For example, the rule that white moves first, is expressed by axiomatically stating that the initial position (P0) be a WHITETURN.

```
axiom INITIAL_MOVER:WHITETURN P0;;
```

Similarly, the fact that the inaugural board is the board START, is specified as:

```
axiom STARTING_BOARD:Tboard P0 =START;;
```

As you can see, definitional axioms are not very exciting.

We gave a large hierarchy for move typing. It is important to state both the inclusive (all moves are of certain sorts) and exclusive (a move is in only one of several classes) properties of this move structure in an axiom.

```
axiom MOVETYPES:
        ∀m.(ENPASSANT m ∨CASTLE m ∨ORDINARY m ),
        ∀m.¬(ENPASSANT m ∧CASTLE m ),
        ∀m.¬(ENPASSANT m ∧ORDINARY m ),
        ∀m.¬(CASTLE m ∧ORDINARY m ),
        ∀mo.¬(CAPTURE mo ≡SIMPLE mo ),
        ∀mc.¬(CAPPP mc ≡CAP mc ),
        ∀ms.¬(SIMPP ms ≡SIM ms ),
        ∀mpp.¬(CAPPP mpp ≡SIMPP mpp ),
        ∀mtx.¬(ENPASSANT mtx ≡CAPTURE mtx );;
```

We claimed that the Makesquare operator, on rows and columns, produced the appropriate square, that the LASTRANKER predicate on squares and colors is decomposable in terms of WLASTRANK and BLASTRANK, and that SQUARE_BETWEEN representes the *betweenness* relation, both orthogonally and diagonally. Each of these definitions entails the appropriate defining axiom. Note, however, we include the axiom for SQUARE_BETWEEN only for reference; this axiom is not subsequently invoked. Rather, all uses of SQUARE_BETWEEN are done through simplification.

```
axiom SQUARED:
    ∀ sq. (sq=Makesquare(Row sq, Column sq)),
    ∀ sq c.(LASTRANKER(sq,c)≡
                ((c=WHITE∧WLASTRANK sq)∨(c=BLACK∧BLASTRANK sq)));;

axiom SQBETWEEN:
    ∀sq1 sq2 sq3.(SQUARE_BETWEEN(sq1 sq2 sq3)≡(
        (Row sq1 =Row sq2 ∧Row sq2 =Row sq3 ∧
                BETWEEN(Column sq1, Column sq2, Column sq3 ))∨
        (Column sq1 =Column sq2 ∧Column sq2 =Column sq3 ∧
                BETWEEN(Row sq1, Row sq2, Row sq3 ))∨
        (SAMEDIAG(sq1 sq2)∧SAMEDIAG(sq2 sq3)∧SAMEDIAG(sq1 sq3)∧
                BETWEEN(Row sq1, Row sq2, Row sq3 ))));;
```

Section 2.1.9.2 Positional Axioms

In the section on position declarations, we made several assertions about the relations between the predicates we declared. We here axiomatize these assertions.

We stated that every position except the initial position was a GAMEPOSITION.[40]

```
axiom POSITION_TYPES:
        ∀r.¬(r=P0=GAMEPOSITION r);;
```

Every GAMEPOSITION is the successor of its predecessor position; every GAMEPOSITION is a descendant of the initial position, P0.

```
axiom POSITION_RULES:
        ∀p.(SUCCESSOR(Prevpos p ,p)∧PREDEGAME(P0,p));;
```

Much like a number system, we can axiomatize the a partial ordering relation (PREDEGAME) on positions. PREDEGAME is true if its first argument occurred in the game that produced its second. It acts much like any partial ordering relation, such as <.[41] If the reader keeps this correspondence in mind, the following axioms will seem transparently valid.

In reading these axioms, one should also recall that the variables r, r1, and r2 range over all POSITIONS (including the initial position, P0), and that p and q are on the domain of GAMEPOSITIONs only.

```
axiom GAMERELATIONS:
        ∀r q.(PREDEGAME(r q)=(SUCCESSOR(r q)∨
                        ∃p.(PREDEGAME(r p)∧PREDEGAME(p q)))),
        ∀p r1 r2.((PREDEGAME(r2 p)∧PREDEGAME(r1 p))⊃
                (PREDEGAME(r1 r2)∨PREDEGAME(r2 r1)∨r2=r1)),
        ∀r1 r2.¬(PREDEGAME(r1 r2)∧PREDEGAME(r2 r1)),
        ∀r1 q r2.(SUCCESSOR(r1 q)⊃¬(PREDEGAME(r1 r2)∧PREDEGAME(r2 q)));;
```

These next three axioms relate the translation between functions and predicates. The first states that the Color function is equivalent to the WHITETURN predicate. The second defines the range over which Pos (the piece on the given square in the given position), and Pospcf (the square on which the given chesspiece rests) are inverses, to wit, when the chesspiece is still on the board (not yet captured). The third states the equivalence of the Val (value) function on pieces, with the Valueon function of the corresponding boards.

```
axiom POS_COLORS: ∀r c.(Color r=c ≡ (WHT c ≡ WHITETURN r));;

axiom POS_TRANSLATION: ∀r sq x.(Pos(r sq)=x ≡ Pospcf(r x)=sq);;

axiom VALUETRANSPOSITION:∀r t sq b.((Pos(r sq)=t∧BOARD(r b))⊃
        (Valueon(b,sq)=Val(r,t)∨Valueon(b sq)=UD));;
```

---

40. Note that the variable "r" ranges over POSITIONS, not merely GAMEPOSITIONS.

41. Remembering, of course, that POSITIONS have a tree like, rather than linear structure.

Section 2.1.10 Miscellaneous Declarations

We shall also have occasion to use a few universal elements, particularlly for the axiom *Substitution*.[42] We declare some universal variables (j and k), and functional parameters β (for single argument functions) and β2 (for two argument functions).

```
declare INDVAR j j1 j2 k k1 k2;
declare OPPAR β 1 [PRE];
declare OPPAR β2 2;
```

Section 2.2 Axioms

Section 2.2.1 Movement axioms

Section 2.2.1.1 Successor definition

Having cleared away most of the definitional rubble (with the exception of a few scattered bricks and window shards, still to be presented) we are ready to express the rules of chess in first order logic. The major vehicle for this task are the movement consequence axioms, (*MCONSEQ*). These detail some of the requirements and consequences of a given position being a successor (legal move away) to another position. In many ways, SUCCESSOR is the fundamental predicate of this axiom system.

```
axiom MCONSEQA:
        ∀r q.(SUCCESSOR(r q)⊃
                ((¬WHITETURN(r)≡WHITETURN(q))∧
                Prevpos(q)=r∧
                ¬POSITIONINCHECK(q,Color r)∧
                (WHITEPIECE Mover Move q≡WHITETURN r)∧
                Pos(r From Move q)=Mover Move q∧
                Pos(q,To Move q)=Mover Move q∧
                Pos(q,From Move q)=EMPTY∧
                (CAPTURE Move q⊃Pos(r,To Move q)=Taken Move q)∧
                (CASTLING(r q)∨EN_PASSANT(r q)∨SIMPLELEGALMOVE(r q))));;
```

This axiom states a series of conditions on positions needed to satisfy the SUCCESSOR predicate.[43]

For two positions to have the successor relationship, they must, of course, be of opposite color. As positions retain the history of their derivation, the first must be the previous position of the second. A caveat against moving and remaining in check is specified. The piececolor of the mover is the same as the side that made the move (you only move your own pieces), and the Mover moved from the From square to the To square of the move. The square he left is then vacant. If the move was an ordinary capture, the captured piece was on the square moved to. Any move is either a *castling* move, an *en passant* capture, or a *simple, legal move*.

---

42 Section A.2.1.

43 This predicate, (though we shall not explicitly do so), would be defined as the conjunction of some of the conditions we will state in this chapter. As all of the analysis we have applied these axioms to is retrograde analysis, forward construction of successors has not been needed. Like so many other things, never being required, it has not been done.

Having defined the fate of the moving piece in any move, we reveal that any taken piece is nowhere to be found (or, at least, is not on any square).

```
axiom MCONSEQF:∀r sq x.(Taken Move r=x⊃¬Pos(r sq)=x);;
```

It is also necessary to state what does not change during a move. Any piece that did not move or was not captured is still on the same square; any square that was not the From or To or Takenon square of the last move retains its identical contents.

```
axiom MCONSEQD:∀r q sq.((SUCCESSOR(r q)∧¬sq=From Move q∧¬sq=To Move q∧
        ¬(CASTLE Move q∧(sq=Alsofrom Move q∨sq=Alsoto Move q))∧
        ¬(ENPASSANT Move q∧sq=Takenon Move q))⊃
                Pos(r sq)=Pos(q sq));;
axiom MCONSEQE:∀r q x.(SUCCESSOR(r q)⊃((¬x=Mover Move q∧
        ¬(TAKINGS Move q∧x=Taken Move q)∧
        ¬(CASTLE Move q∧x=Alsomover Move q))⊃
                Pospcf(r x)=Pospcf(q x)));;
```

There are also the loose ends of these functions to be tied. We wish these functions to be defined only on the appropriate positions; to speak of the Takenon square of a castling move is meaningless. While the need for this axiom is probably not obvious, its restrictions are required in the proofs of several later theorems.

```
axiom MCONSEQG:∀r t sq.((¬TAKINGS Move r⊃¬t=Taken Move r)∧
    (¬ENPASSANT Move r⊃¬sq=Takenon Move r)∧(¬GAMEPOSITION r⊃¬MOVES Move r)∧
    (¬GAMEPOSITION r⊃¬sq=To Move r)∧(¬GAMEPOSITION r⊃¬sq=From Move r)∧
    (¬GAMEPOSITION r⊃¬t=Mover Move r)∧
    (¬CASTLE Move r⊃(¬sq=Alsoto Move r∧¬t=Alsomover Move r)));;
```

These next three axioms deal with the special circumstances of pawn promotions. The first states that the only way a piece can change its VALUE is by being the mover of a pawn promotion; we use this fact, for instance, to prove that any non-pawn chess piece always has the same value.[44] The second is definitional for the predicate PROMOTEDPAWN. The third places limitations on pawn promotions, specifying that a pawn promotion moves a pawn to the last rank of his color, by a simple, legal move, that the piece must have pawn value when he starts the move, and must have a value from the set of possible promotion values (queen, rook, bishop and knight) when done. The axiom bars chameleon promotions; the pawn retains its color though the move.

```
axiom MCONSEQH:∀r q t.((SUCCESSOR(r q)∧
        (¬PAWNPROM Move q∨¬t=Mover Move q))⊃Val(r t)=Val(q t));;
axiom MCONSEQI:
    ∀r t.(PROMOTEDPAWN(r t)≡
        ∃q.(PAWNPROM(Move(q))∧(PREDEGAME(q r)∨q=r)∧Mover(Move q)=t));;
axiom MCONSEQL:
    ∀p .(PAWNPROM Move p≡(LASTRANKER(To Move p,Color Prevpos p)∧
            SIMPLELEGALMOVE(Prevpos p p)∧
            PAWNS Mover Move p∧
            VALUEP Valueon(Tboard Prevpos p,From Move p)∧
          ((BVALUES Promoted Move p=BVALUES Val(Prevpos p Mover Move p))∧
           (WVALUES Promoted Move p=WVALUES Val(Prevpos p Mover Move p)))∧
            Val(p Mover Move p)=Promoted Move p));;
```

44 Section 3.3.1.

The definitional equivalence of the three types of successions, and their respective moves is declared.

```
axiom MCONSEQM:
    Vp.((CASTLE Move p=CASTLING(Prevpos p p))∧
        (ENPASSANT Move p=EN_PASSANT(Prevpos p p))∧
        (ORDINARY Move p=SIMPLELEGALMOVE(Prevpos p p)));;
```

The above axioms are not quite strong enough in their limitations of that special position, the initial position. So we include this additional axiom.

```
axiom MCONSEQ0:Vt sq.(¬Mover Move P0=t∧¬From Move P0=sq∧
                ¬To Move P0=sq∧¬Taken Move P0=t∧
                ¬MOVES Move P0);;
```

Section 2.2.1.2 Simple legal motion

We have split the chess move world into three parts, castling, en passant and ordinary moves. We must now define each of these classifications. Let us start with the last, certainly the most common.

The definition of a SIMPLELEGALMOVE is given in the axiom *MCONSEQK*. It demands that the move source (From) square differ from the destination (To) square, that in non-capturing moves, the move always go to an empty square, and that in capturing moves, the captured piece always be a member of the opposing army. The predicate MOVETO, on the (total) board of the *moving from* position, need also be satisfied. Notice that, in some important sense, we are not cheating; for retrograde analysis, it would be much more convenient to define the move in terms of the destination board. However, this is not the way the rules of chess are naturally expressed. MOVETO defines the different moves of the individual values.

```
axiom MCONSEQK:
  Vr q.(SIMPLELEGALMOVE(r q)=
      (¬From Move q=To Move q∧
      MOVETO(Tboard r Valueon(Tboard r,From Move q) From Move q To Move q)∧
      (    (SIMPLE Move q∧Valueon(Tboard r,To Move q)=MT)∨
           (CAPTURE Move q∧PIECEVALUES(Valueon(Tboard r,To Move q))∧
                ¬Valuecolor(Valueon(Tboard r,To Move q))=Color r))));;
```

The predicate MOVETO is, of course, the composite of five different predicates, representing the possible major movement types of chess. Chess pieces can move orthogonally, like rooks and queens, on a bishop's (and queen's) diagonal, to the king's adjacent square, by the knight's jump, or in the slow, advancing move of a pawn. A predicate for each of these styles is declared; it is true when that move is legal on the given board, from the first square to the second. Notice that MOVETO, as we have defined it here, does not include consideration of the *end* squares of the move. This is because we wish to more easily conclude, *if that unknown piece is a rook, then it could move to that square.* However, this makes it subtly and slightly more difficult to prove moves about completely defined situations. Life is a trade off.

The auxiliary predicate TWOTOUCHING (are the column arguments next to each other; that is, can a pawn capture from the first column to the second) is also declared. The functions L2touchf and R2touchf embody the next column left and next column right notions.

```
declare PREDCONST ORTHO (BOARDS, SQUARES, SQUARES);
declare PREDCONST DIAG (BOARDS, SQUARES, SQUARES);
declare PREDCONST PAWNMOVE (BOARDS, VALUES, SQUARES, SQUARES);
declare PREDCONST KINGMOVE (SQUARES, SQUARES);
declare PREDCONST KNIGHTMOVE (SQUARES, SQUARES);

declare PREDCONST TWOTOUCHING (ISDIMENSION, ISDIMENSION);
declare OPCONST L2touchf R2touchf (ISDIMENSION)=ISDIMENSION [PRE];
```

The attachments to the next column touching functions have a convenient inversion; when the function would be otherwise undefined (at the edge of the board) the opposite direction is selected

```
attach L2touchf [NATNUMREP→NATNUMREP]
        (DE L2touchf(r)(COND ((EQ r 1)2)(T(SUB1 r))));
attach R2touchf [NATNUMREP→NATNUMREP]
        (DE R2touchf(r)(COND ((EQ r 8)7)(T(ADD1 r))));
```

Orthogonality and diagonality are given predicate logic definitions, in the obvious manner. Pawn moves are broken into black and white pawn movements, and the three types of pawn moves (single space ahead, capture diagonal advance, and two space first move) are described for each of black and white. As the geometry of king and knight moves are purely a function of the squares involved (at least in the sense that the limitations are imposed elsewhere), we do not need a formal logic definition of their potential actions. Rather, we invariably rely upon our chess eye for decisions of this kind. The axioms that we would have defined for king and knight moves are derivable from the chess eye's functions.

Note that if our chess eye were capable of computing on incompletely defined quantities (variable objects with known properties, for example), we might be able to avoid having definitions of ORTHO and DIAG. That is, if FOL permitted the passing of a *variable* board to these functions, then many of the derivations that use the definitions of ORTHO and DIAG could be done merely by simplification. However, in the more complex cases, simplify might have to consider four thousand square pairs or a quarter of a million triplets. The former, while painful, is computationally feasible. The latter is not. Hence, the *definitions* of these predicates.

```
declare PREDCONST WPAWNMOVE (BOARDS, SQUARES, SQUARES);
declare PREDCONST BPAWNMOVE (BOARDS, SQUARES, SQUARES);

axiom MOVING:
        ∀b v sq1 sq2.(MOVETO (b v sq1 sq2)≡
                ((VALUER(v)∧ORTHO(b sq1 sq2)) ∨
                 (VALUEB(v)∧DIAG(b sq1 sq2)) ∨
                 (VALUEQ(v)∧ORTHO(b sq1 sq2)) ∨
                 (VALUEQ(v)∧DIAG(b sq1 sq2)) ∨
                 (VALUEK(v)∧KINGMOVE(sq1 sq2)) ∨
                 (VALUEN(v)∧KNIGHTMOVE(sq1 sq2)) ∨
                 (VALUEP(v)∧PAWNMOVE(b v sq1 sq2)))),
```

```
        ∀b sq1 sq2.(ORTHO(b sq1 sq2)≡
            (¬sq1=sq2∧
                ((Column sq1=Column sq2 ∧
                  ∀sq3.((BETWEEN (Row sq1, Row sq3, Row sq2)
                        ∧Column sq3=Column sq1)⊃
                        Valueon(b sq3)=MT))∨
                 (Row sq1=Row sq2∧
                  ∀sq3.((BETWEEN(Column sq1, Column sq3, Column sq2)∧
                         Row sq3=Row sq1)⊃
                        Valueon(b sq3)=MT))))),
        ∀b sq1 sq2.(DIAG (b sq1 sq2) ≡
                (SAMEDIAG (sq1 sq2) ∧
                 ∀sq3.  ((SAMEDIAG (sq1 sq3) ∧
                          SAMEDIAG (sq2 sq3) ∧
                          BETWEEN (Row  sq1, Row  sq3, Row  sq2))⊃
                        Valueon (b sq3) = MT)));;
axiom PAWNMOVING:
        ∀b v sq1 sq2.(PAWNMOVE(b v sq1 sq2)≡
                ((WPAWNMOVE (b sq1 sq2) ∧ WVALUES v) ∨
                 (BPAWNMOVE (b sq1 sq2) ∧ BVALUES v))),
        ∀ b sq1 sq2.(WPAWNMOVE(b sq1 sq2)≡
                 ((Column sq1=Column sq2∧
                   WSUC(Row sq1, Row sq2)∧
                   Valueon(b sq2)=MT)  ∨
                  (Column sq1=Column sq2∧
                   Row sq1=7∧
                   Valueon(b sq2)=MT∧
                   Valueon(b Makesquare(6, Column sq1))=MT∧
                   Row sq2=5)∨
                  (TWOTOUCHING(Column sq1, Column sq2)∧
                   WSUC(Row sq1, Row sq2)∧
                   BVALUES Valueon(b sq2)))),
        ∀ b sq1 sq2.(BPAWNMOVE(b sq1 sq2)≡
                 ((Column sq1=Column sq2∧
                   BSUC(Row sq1, Row sq2)∧
                   Valueon(b sq2)=MT)  ∨
                  (Column sq1=Column sq2∧
                   Row sq1=2∧
                   Valueon(b sq2)=MT∧
                   Valueon(b Makesquare(3, Column sq1))=MT∧
                   Row sq2=4)∨
                  (TWOTOUCHING(Column sq1, Column sq2)∧
                   BSUC(Row sq1, Row sq2)∧
                   WVALUES Valueon(b sq2))));;
```

Each of the possible moves also has an attachment in the LISP model. These attachments are to be part of our *Chess Eye*.[45] The chess eye functions are defined and explained in the following sections; their correspondence to these definitional axioms should be obvious.

---

45 Section 1.5.2.

Section 2.2.1.2.1 Ortho Attachments

The auxiliary LISP function ALLFREER (all free (empty) in the row) is given. It takes a *board*, a *row*, a *from* column, and a *to* column, (*from* being arithmetically less than *to*) and returns t (true) if every square on that board, in the given row, between the given columns, is empty (has MT as its Valueon), NIL (false) otherwise. The function ALLFREEC performs the corresponding action for columns.

```
FUNCTION (DE ALLFREER (b r from to)(COND
    ((EQ from (SUB1 to))T)
    ((EQ (Valueon b (CONS r(SUB1 to)))(QUOTE MT))
     (ALLFREER b r from (SUB1 to)))));

FUNCTION (DE ALLFREEC (b r from to)(COND
    ((EQ from (SUB1 to))T)
    ((EQ (Valueon b (CONS (SUB1 to)r))(QUOTE MT))
     (ALLFREEC b r from (SUB1 to)))));
```

Using these two functions, the orthogonality check for two squares merely becomes a check to see if they share a row or column, and if all the squares between the argument squares are free. Note that no square is ORTHO to itself.

```
attach ORTHO [CHESS,CHESS,CHESS] (DE ORTHO(b r S)(OR
(AND (EQ (CAR r) (CAR S))
     (OR (AND (LESSP (CDR r)(CDR S))
              (ALLFREER b (CAR r) (CDR r) (CDR S)))
         (AND (GREATERP (CDR r)(CDR S))
              (ALLFREER b (CAR r) (CDR S) (CDR r)))))
(AND (EQUAL (CDR r) (CDR S))
     (OR (AND (LESSP (CAR r)(CAR S))
              (ALLFREEC b (CDR r) (CAR r) (CAR S)))
         (AND (GREATERP (CAR r)(CAR S))
              (ALLFREEC b (CDR r) (CAR S) (CAR r)))))));
```

Section 2.2.1.2.2 Diag Attachments

Diagonal movement attachment is similar to orthogonal. The predicate ALLFREED checks if all the squares on the diagonal between two given squares are empty. SAMEDIAG (defined earlier) is true if the two squares lie on a diagonal. SIGN is simply the sign function of mathematics.

```
FUNCTION (DE SIGN(x)(COND((MINUSP x)(SUB1 0))((ZEROP x)0)(T 1)));

FUNCTION (DE ALLFREED(b r1 c1 r2 c2)(PROG (x y)
         (SETQ x (SIGN (DIFFERENCE r2 r1)))
         (SETQ y (SIGN (DIFFERENCE c2 c1)))
    LOOP (SETQ r1 (PLUS r1 x))
         (SETQ c1 (PLUS c1 y))
         (COND ((EQ r1 r2)(RETURN T))
               ((EQ (Valueon b (CONS r1 c1))(QUOTE MT))(GO LOOP)))));
```

The attachment to DIAG then simply becomes:

```
attach DIAG [CHESS,CHESS,CHESS] (DE DIAG(b sq1 sq2)
        (AND (SAMEDIAG sq1 sq2)
             (ALLFREED b (CAR sq1)(CDR sq1)(CAR sq2)(CDR sq2))));
```

Section 2.2.1.2.3 Knightmove Attachments

As a knight can effectively *jump* over other chesspieces, the function that computes the KNIGHTMOVE between two squares does not need to refer to any board. Rather, two squares have this relationship purely geometrically; if the differences of their coordinates are two and one, the squares are a knights jump apart.

```
attach KNIGHTMOVE [CHESS,CHESS]
  (DE KNIGHTMOVE(x y)
   (AND
     (SQUARES x)
     (SQUARES y)
     (OR (AND(EQ 1 (ABS(DIFFERENCE (CAR x)(CAR y))))
             (EQ 2 (ABS(DIFFERENCE (CDR x)(CDR y)))))
         (AND(EQ 2 (ABS(DIFFERENCE (CAR x)(CAR y))))
             (EQ 1 (ABS(DIFFERENCE (CDR x)(CDR y))))))));
```

Section 2.2.1.2.4 Kingmove Attachments

Like the knight's move, the king's move is not limited by any squares beside the source and destination.

```
attach KINGMOVE [CHESS,CHESS] (DE KINGMOVE (x y)(AND
  (NOT(EQUAL x y))
  (SQUARES x)
  (SQUARES y)
  (LESSP(ABS(DIFFERENCE(CAR x)(CAR y)))2)
  (LESSP(ABS(DIFFERENCE(CDR x)(CDR y)))2)));
```

Section 2.2.1.2.5 Pawn Moves

Attachments are given for the predicates used in the pawn move axioms. These functions are a fairly straightforward translation of their definitional axioms.

```
attach TWOTOUCHING [NATNUMREP,NATNUMREP] (DE TWOTOUCHING (x y)
        (AND(NUMBERP x)(NUMBERP y)(EQ 1 (ABS (DIFFERENCE x y)))));
attach WSUC [NATNUMREP,NATNUMREP]
        (DE WSUC(x y)(AND(NUMBERP x)(NUMBERP y)(EQ 1 (DIFFERENCE x y))));
attach BSUC [NATNUMREP,NATNUMREP] (DE BSUC (x y)(WSUC y x));
```

```
attach WPAWNMOVE [CHESS,CHESS,CHESS] (DE WPAWNMOVE(b sq1 sq2)
(OR (AND(EQ(CDR sq1)(CDR sq2))
        (EQ(Valueon b sq2)(QUOTE MT))
        (OR (EQ (CAR sq1)(ADD1 (CAR sq2)))
            (AND (EQ (CAR sq1) 7)
                 (EQ (CAR sq2) 5)
                 (EQ (Valueon b (CONS 6 (CDR sq1)))(QUOTE MT)))))
    (AND (OR (EQ (CDR sq1)(ADD1 (CDR sq2)))
             (EQ (ADD1(CDR sq1))(CDR sq2)))
         (EQ (CAR sq1)(ADD1 (CAR sq2)))
         (BVALUES (Valueon b sq2)))));
attach BPAWNMOVE [CHESS,CHESS,CHESS] (DE BPAWNMOVE(b sq1 sq2)
(OR (AND(EQ(CDR sq1)(CDR sq2))
        (EQ(Valueon b sq2)(QUOTE MT))
        (OR (EQ (ADD1(CAR sq1))(CAR sq2))
            (AND (EQ (CAR sq1) 2)
                 (EQ (CAR sq2) 4)
                 (EQ (Valueon b (CONS 3 (CDR sq1)))(QUOTE MT)))))
    (AND (OR (EQ (CDR sq1)(ADD1 (CDR sq2)))
             (EQ (ADD1(CDR sq1))(CDR sq2)))
         (EQ (ADD1(CAR sq1))(CAR sq2))
         (WVALUES (Valueon b sq2)))));
attach PAWNMOVE [CHESS,CHESS,CHESS,CHESS] (DE PAWNMOVE (b v sq1 sq2)
(COND ((WVALUES v)(WPAWNMOVE b sq1 sq2))
      ((BVALUES v)(BPAWNMOVE b sq1 sq2))));
```

Section 2.2.1.2.6 Bringing It All Together

With the above functions, the definition of a LISP attachment for MOVETO becomes quite trivial. For efficiency's sake, we take the liberty of using our knowledge that chess piece sorts are disjoint in the translation of the axiom. Otherwise, the initial COND would be an OR.

```
attach MOVETO [CHESS,CHESS,CHESS,CHESS] (DE MOVETO (b v sq1 sq2)(COND
    ((VALUEQ v)(OR (ORTHO b sq1 sq2)(DIAG b sq1 sq2)))
    ((VALUER v)(ORTHO b sq1 sq2))
    ((VALUEB v)(DIAG b sq1 sq2))
    ((VALUEK v)(KINGMOVE sq1 sq2))
    ((VALUEN v)(KNIGHTMOVE sq1 sq2))
    ((VALUEP v)(PAWNMOVE b v sq1 sq2))));
```

Section 2.2.1.3 Castling

An axiomatic definition of the CASTLING predicate is given. One position obtains from another by castling under the following conditions. The mover of the move must be a king, and the Alsomover, a function peculiar to the castle move, a rook. The rook is constrained to be in an Alsofrom square before the move, just as the general movement rules constrain the Mover to the From square. Similarly, the Alsomover moves to the Alsoto square.

The next two conjuncts state that both the rook and the king must have been on these squares in every move that preceded this position. Every square between the rook and the king must be empty.

Castling can not be used when in check, nor can the king pass through check in making a castle.

The last two conditions specify the destination squares in castlings.

Fortunately, we will not have to use this hairy axiom to prove that a castle took place in any given situation. However, we will frequently have to prove that a castle did not take place. We will develop theorems to make this easier.

```
axiom CASTLEMOVES:∀r p.(CASTLING(r p)≡
 (KINGS Mover Move p∧
  ROOKS Alsomover Move p∧
  Pos(r, Alsofrom Move p)=Alsomover Move p∧
  Pos(p, Alsofrom Move p)=EMPTY∧
  Pos(p, Alsoto Move p)=Alsomover Move p∧
  ∀r1.(PREDEGAME(r1 p)⊃Pos(r From Move p)=Mover Move p)∧
  ∀r1.(PREDEGAME(r1 p)⊃Pos(r Alsofrom Move p)=Alsomover Move p)∧
  ∀sq3.((Row sq3=Row From Move p∧
      BETWEEN(Column From Move p, Column sq3, Column Alsofrom Move p))⊃
          Pos(r sq3)=EMPTY)∧
  ¬POSITIONINCHECK(r, Color r)∧
  ∀sq1 x.¬(Pos(r sq1)=x∧ MOVETO(Tboard r, Val(r x), sq1, Alsoto Move p)∧
          Piececolor x=Color p)∧
  (WHITETURN r⊃((Alsomover Move p=WKR∧Alsoto Move p=WKB1∧To Move p=WKN1)∨
        (Alsomover Move p=WQR∧Alsoto Move p=WQ1∧To Move p=WQB1)))∧
  (¬WHITETURN r⊃((Alsomover Move p=BKR∧Alsoto Move p=BKB1∧To Move p=BKN1)∨
        (Alsomover Move p=BQR∧Alsoto Move p=BQ1∧To Move p=BQB1)))));;
```

Section 2.2.1.4 Capture En Passant

As chess was *originally* defined, pawns moved forward only one rank at a time. In an effort to quicken the opening, the rules were modified to allow a pawn to step two spaces on its first move. To avoid permitting a pawn to thereby jump and pass an opposing pawn in an adjacent column (and thereby, perhaps, become a valuable passed pawn), the *en passant* capture rule was introduced. This permitted a player whose pawn could have captured a two stepping pawn (if it had taken only a single step) to do so, effectively, move the pawn back and capture it, though this right was only extended for the immediately subsequent move.

A complicated rule produces a complicated axiom. This axiom must refer to both the current position (q) and the move that reached the previous position (r). Here we refer to the Takenon square as the square the captured piece moved to on the previous move. After the move, the square that the captured pawn occupied is now vacant.

The previous move must have satisfied several conditions; the mover must have been the piece captured, it must have moved to the Takenon square, it must have done so with a simple (SIM) move, which stayed in the same column. The capture move will move into that column. The mover and the captured piece both have the value pawn when the capture takes place. The actual rows of the particular moves are given, depending upon the side making the capture. From the row and column information, it is possible to reconstruct the various relevant squares.

```
axiom ENPASS:∀r q.(EN_PASSANT(r q)≡(
    GAMEPOSITION r∧
    Pos(q Takenon Move q)=EMPTY∧
    To Move r=Takenon Move q∧
    Mover Move r=Taken Move q∧
    SIM Move r∧
    Column From Move r=Column To Move r∧
    Column To Move r= Column To Move q∧
    TWOTOUCHING(Column From Move q, Column To Move q)∧
    (WHITETURN q⊃(Val(q Mover Move q)=PB∧
                  Val(r Mover Move r)=PW∧
                  Row From Move q=5∧
                  Row To Move q=6∧
                  Row From Move r=7∧
                  Row To Move r=5))∧
    (¬WHITETURN q⊃(Val(q Mover Move q)=PW∧
                   Val(r Mover Move r)=PB∧
                   Row From Move q=4∧
                   Row To Move q=3∧
                   Row From Move r=2∧
                   Row To Move r=4))));;
```

We (fortunately) shall not use this axiom, except to prove the last move was not an *en passant* capture. For this purpose, we will develop several simplifying lemmas.

Section 2.2.2 In Check Definitions

Having specified the different moves of the chesspieces, we can now define what it means to be in check on a board or in a position. The axiom *CHECKERS* states the necessary conditions.

```
axiom CHECKERS:
 ∀b.(WHITEINCHECK(b)≡
   ∃vb sq1 sq2.(Valueon(b sq2)=KW∧Valueon(b sq1)=vb∧MOVETO(b vb sq1 sq2))),
 ∀b.(BLACKINCHECK(b)≡
   ∃vw sq1 sq2.(Valueon(b sq2)=KB∧Valueon(b sq1)=vw∧MOVETO(b vw sq1 sq2))),
 ∀b c.(SIDEINCHECK(b c)≡
   ((WHITEINCHECK(b)∧WHT(c))∨(BLACKINCHECK(b)∧BLK(c)))),
 ∀r c.(POSITIONINCHECK(r c)≡∃b.(SIDEINCHECK(b c)∧BOARD(r b)));;
```

The attachments to WHITEINCHECK, BLACKINCHECK and SIDEINCHECK differ somewhat, in spirit, from the other attachments. Here we use our knowledge of the *unique king* of any chessboard to simplify the computation. Note also the *scanning* of the board to find possible checking pieces used in the auxiliary functions.

```
FUNCTION (DE FINDKING (x b)(PROG (rw c1)
         (SETQ rw 1)
         (SETQ c1 1)
ROWLOOP  (COND ((NULL b)(RETURN NILL))
               ((NULL (CAR b))(SETQ b(CDR b))(SETQ rw(ADD1 rw))(SETQ c1 1)))
         (COND ((EQ (CAAR b) x)(RETURN (CONS rw c1))))
         (SETQ b (CONS (CDAR b)(CDR b)))
         (SETQ c1 (ADD1 c1))
         (GO ROWLOOP)));
```

```
FUNCTION (DE INCHECK(b kingsq Colormovingf)(PROG (bl rw cl)
        (COND ((NULL kingsq)(RETURN NILL)))
        (SETQ rw 1)(SETQ cl 1)(SETQ bl b)
RWLP    (COND ((NULL (CAR bl))
               (SETQ bl (CDR bl))(SETQ rw (ADD1 rw))(SETQ cl 1)))
        (COND ((NULL bl)(RETURN NILL)))
        (COND ((Colormovingf (CAAR bl))
            (COND ((MOVETO b (CAAR bl) (CONS rw cl) kingsq) (RETURN T)))))
        (SETQ bl (CONS (CDAR bl)(CDR bl)))
        (SETQ cl (ADD1 CL))
        (GO RWLP)));

attach WHITEINCHECK [CHESS]
  (DE WHITEINCHECK (b) (INCHECK b (FINDKING (QUOTE KW) b)(QUOTE BVALUES)));
attach BLACKINCHECK [CHESS]
  (DE BLACKINCHECK (b) (INCHECK b (FINDKING (QUOTE KB) b)(QUOTE WVALUES)));
attach SIDEINCHECK [CHESS,CHESS] (DE SIDEINCHECK (b c)(COND
  ((EQ c(QUOTE WHITE))(WHITEINCHECK b))
  ((EQ c(QUOTE BLACK))(BLACKINCHECK b))));
```

Section 2.2.3 Board Axioms

The SUCCESSOR definitions determine the effect of the various moves on the total boards (Tboard) of positions. However, we still require primitives for the manipulation of the partial boards, those with undefined (UD) squares.

Section 2.2.3.1 Sub-board Definition

In the section on board declarations, we asserted various properties for sub-boards and board constructors. These need axiomatization. The axiom SUB_BOARDS consists of four such definitions. The first WFF defines the Makeboard function. This functions takes a board, a square, and a value, and creates a board identical to the original board on every square except the argument square. On this square, Makeboard places the given value.

The second part of the axiom states that every total board has no undefined squares. The third defines the relation BOARD, between a position and a partial board, in terms of the SUBOARD predicate and Tboard function on that position. The last part of the axiom defines the SUBOARD relationship. One board subsumes another if they are everywhere the same, except on those squares where the less defined board is undefined.

```
axiom SUB_BOARDS:
  ∀r b sq t v.((Val(r t)=v∧Pos(r sq)=t∧BOARD(r b))⊃
                                          BOARD(r,Makeboard(b sq v))),

  ∀bt sq.¬Valueon(bt sq)=UD,
  ∀r b .(BOARD(r b)≡SUBOARD(b Tboard(r))),
  ∀a b.(SUBOARD(a b)≡∀sq.(Valueon(a sq)=Valueon(b sq)∨Valueon(a sq)=UD)),
  ∀b1 b2 sq1 v1.(b1=Makeboard(b2,sq1,v1)≡
        (∀sqx.(¬sqx=sq1⊃
                  Valueon(b1,sqx)=Valueon(b2,sqx))∧Valueon(b1,sq1)=v1));;
```

An attachment for the Makeboard operator is declared.

```
FUNCTION (DE MKBOARD1 (r N v)(COND
    ((EQ 1 N)(CONS v (CDR r)))
    ((CONS (CAR r)(MKBOARD1 (CDR r)(SUB1 N)v)))));

attach Makeboard [CHESS,CHESS,CHESS→CHESS] (DE Makeboard (b S v)(COND
    ((EQ (CAR S) 1)(CONS (MKBOARD1 (CAR b)(CDR S) v)(CDR b)))
    ((CONS (CAR b)(Makeboard (CDR b)(CONS (SUB1 (CAR S))(CDR S)) v)))));
```

Section 2.2.3.2 Board Manipulation

We have provided a mechanism for building boards *up* from less well defined boards. However, unless we want to be limited to always constructing from the totally undefined board, a very long and painful process, we need board *decomposers*, to take a board and the information from a move, and produce what can be determined of the previous board

For example, we declare the following four move *unmakers*, which take move information and a board, and compute a sub-board of the previous position.

```
declare OPCONST Unmkmove       (BOARDS, SQUARES, SQUARES)→BOARDS;
declare OPCONST Unmkcapmove    (BOARDS, SQUARES, SQUARES, VALUES)→BOARDS;
declare OPCONST Unmksppmove    (BOARDS, SQUARES, SQUARES)→BOARDS;
declare OPCONST Unmkcapppmove (BOARDS, SQUARES, SQUARES, VALUES)→BOARDS;
```

The functions undo, respectively, all ordinary, non-pawn promotion moves, ordinary, non-promotional captures, simple pawn promotions, and capturing pawn promotions. The more specific a decomposer function used, the better defined the resulting board, of course. These are just a sample of the possible set of decomposer functions; however, combined with the Makeboard function, they are powerful enough for our needs.

These functions come with both attachments, and axioms dictating their use. The attachments rely heavily on the Makeboard function.

```
attach Unmkmove [CHESS,CHESS,CHESS→CHESS] (DE Unmkmove (b r S)
    (Makeboard(Makeboard b S (QUOTE UD))r (Valueon b S)));
attach Unmkcapmove [CHESS,CHESS,CHESS,CHESS→CHESS] (DE Unmkcapmove(b r S v)
    (Makeboard(Makeboard b S v)r (Valueon b S)));
attach Unmksppmove [CHESS,CHESS,CHESS→CHESS] (DE Unmksppmove(b r S)
    (Makeboard(Makeboard b S (QUOTE MT))r
        (COND ((WVALUES (Valueon b S))(QUOTE PW))(T(QUOTE PB)))));
attach Unmkcapppmove [CHESS,CHESS,CHESS,CHESS→CHESS]
    (DE Unmkcapppmove(b r S v)
        (Makeboard (Makeboard b S v) r
            (COND ((WVALUES (Valueon b S))(QUOTE PW))(T(QUOTE PB)))));
```

The use of these functions is delimited by this axiom, *UNDO*.

```
axiom UNDO:
    ∀r q b sq1 sq2.((SUCCESSOR (r q)∧BOARD(q b)∧ORDINARY Move q∧
        ¬PAWNPROM Move q∧From Move q=sq1∧To Move q=sq2)⊃
            BOARD(r Unmkmove(b sq1 sq2))),
    ∀r q b sq1 sq2 v.((SUCCESSOR(r q)∧BOARD(q b)∧CAP Move q∧
        From Move q=sq1∧To Move q=sq2∧Val(r,Taken Move q)=v)⊃
            BOARD(r Unmkcapmove(b sq1 sq2 v))),
```

```
∀r q b sq1 sq2.((SUCCESSOR(r q)∧BOARD(q b)∧SIMPP Move q∧
    From Move q=sq1∧To Move q=sq2)⊃
        BOARD(r Unmksppmove(b sq1 sq2))),
∀r q b sq1 sq2 v.((SUCCESSOR(r q)∧BOARD(q b)∧CAPPP Move q∧
    Val(r,Taken Move q)=v∧From Move q=sq1∧To Move q=sq2)⊃
        BOARD(r Unmkcapppmove(b sq1 sq2 v)));;
```

Section 2.2.4 Global Notions

So far, the definitions and axioms presented have been of a local nature. That is, they detail the transition from one position to the next, or the effect of a given move on a board. We now lay the groundwork for more global notions, useful for proving what must have happened during the game that reached some position.

Section 2.2.4.1 Chess Induction

Perhaps the most aesthetically pleasing notion of the entire axiomatization is that of *Chess Induction*. Chess induction is a natural extension of the correspondence between the numerical predicate *less than* on the natural numbers, and the chess predicate PREDEGAME of the chess world. A mathematical induction proof has two parts, the first a proof in an initial state, the second a proof on a transition from state to state by successor function. For these premises, mathematical induction concludes a predicate true of all states.

Chess induction action is similar in principle. A proof of some proposition on POSITIONS, α, is first a proof on some specific position, r, that α(r) is true, then a proof that the proposition α holds over the SUCCESSOR relation. We can then conclude that the proposition holds for all positions which have r as one of their ancestors. As all GAMEPOSITIONS have the initial position, P0, in their history, we will often use P0 as the position r. The resulting theorem will then be true of all POSITIONS.

Just as many powerful mathematical theorems are proven through the use of mathematical induction, so we will be able to prove many interesting chess theorems by chess induction.

Chess induction is an axiom schema. For axiom schema, we need a predicate parameter, declared:

```
declare PREDPAR α (POSITIONS)[PRE];
```

Note that α is a prefix predicate, and can be used without parentheses around its arguments.

The axiom schema itself is written:

```
axiom CHESS_INDUCTION:
∀r.((α r∧
   ∀r1 p2.((α r1∧(PREDEGAME(r r1)∨r=r1)∧SUCCESSOR(r1 p2))⊃α p2))⊃
        ∀r2.((PREDEGAME(r r2)∨r=r2)⊃α r2)));;
```

Our explanation, so far, has paralleled one of the more general explanations of ordinary mathematical induction. Mathematical induction, while is an expression of *if it is true of* x, *(and the induction hypothesis is satisfied) then, for all* y>x, *it must also be true.* In practice, however, mathematical induction is almost invariably used with x=0, resulting in validity on all integers. Our

practice with chess induction is similar. Almost all of our proofs involving chess induction pick P0 for an initial case; they thereby produce proofs valid on all positions. This simplified for of chess induction we call *ChsInd*:

$$(\alpha\ P0 \wedge \forall r\ p.((\alpha\ r \wedge SUCCESSOR(r,p)) \supset \alpha\ p)) \supset \forall r.\alpha\ r$$

The derivation of *ChsInd* from *CHESS_INDUCTION* is section A.2.

Section 2.2.4.2 The Mathematics of Pawn Captures

Section 2.2.4.2.1 Pawn Capture Definitions

There is another, though perhaps more parochial, group of theorems and predicates still to be considered. Any aficionado of chess problems knows that the position of a pawn on a board puts a minimum on the number of pieces it had to capture to reach that square. Basically speaking, a pawn must have captured at least one opposing piece for each column it is away from its initial column (presuming, of course, that the pawn has not promoted in the meantime). While this is both an extremely useful and interesting limit, practically, it leans more towards mathematics than we would prefer to go. Consequently, though we declare the predicates to FOL, we leave the actual computations to the attachment mechanism.

No pawn can move more than seven columns from his original column; nor less than none. We call this set, zero through seven, the NUMBERS. We also need the mathematical predicate ≥, (less than or equals) to compare our numbers. The declarations and attachments look like:

```
declare PREDCONST NUMBERS (NATNUM);
declare PREDCONST ≥ (NUMBERS,NUMBERS) [INF];

extension NUMBERS (0 1 2 3 4 5 6 7);
mg NATNUM≥ (NUMBERS);
attach NUMBERS [NATNUMREP]
        (LAMBDA (x) (AND (NUMBERP x) (LESSP x 9) (NOT (LESSP x 0))));
attach ≥ [NATNUMREP,NATNUMREP] (LAMBDA (x y) (NOT (LESSP x y)));
```

We also desire a function to compute the pawn capturing distance between two squares.

```
declare OPCONST Pawncaptures (SQUARES,SQUARES)→NUMBERS;
attach Pawncaptures [CHESS,CHESS→NATNUMREP]
        (LAMBDA (x y) (ABS (DIFFERENCE (COR x) (COR y))));
```

Lastly, we declare two predicates on squares and colors. The first, MAY_PAWNCAPTURES, takes two squares and a color, and returns true if a pawn of that color could have reached the second square from the first. The second, MUST_PAWN_CAPTURES is true if a pawn of that color, in going from the first to the second square, must have captured a piece every time it moved.

```
declare PREDCONST MAY_PAWN_CAPTURES (SQUARES,SQUARES,COLORS);
declare PREDCONST MUST_PAWN_CAPTURES (SQUARES,SQUARES,COLORS);
```

```
attach MAY_PAWN_CAPTURES [CHESS,CHESS,CHESS]
  (DE MAY_PAWN_CAPTURES (x y c)(COND
   ((GREATERP
      (DIFFERENCE (TIMES (COND ((EQ c (QUOTE WHITE)) (SUB1 0))(T 1))
                         (DIFFERENCE(CAR y)(CAR x)))
                  (ABS(DIFFERENCE(CDR y)(CDR x))))
      (SUB1 0)))));

attach MUST_PAWN_CAPTURES [CHESS,CHESS,CHESS]
  (DE MUST_PAWN_CAPTURES (x y c)(AND (NOT (EQUAL x y))(COND
       ((ZEROP (DIFFERENCE (TIMES (COND ((EQ c (QUOTE WHITE)) (SUB1 0))
                                        (T 1))
                                  (DIFFERENCE(CAR y)(CAR x)))
                           (ABS(DIFFERENCE(CDR y)(CDR x))) ))))));
```

Section 2.2.5 Asserted Theorems

There are several theorems in this volume which we have not proven. These are theorems, in that they are provable from the axioms we have given, (with, perhaps, a little help from standard mathematics). However, a certain misguided sense of honesty compels their mention in the axioms chapter. For some of these, the proof is so trivial (perhaps a change from white's point of view to black's, or a proof that knights do not promote, when we already have one for bishops) as to make the actual proof more an exercise in mindless substitution than in logic. Obviously, the value of detailing another special example is minimal. Hence, we shall simply declare such theorems when we prove the associated, similar theorem, rather than presenting their proofs.

There are also a few moderately complex theorems, of a general nature, obviously true in themselves, but for which time and energy constraints have not allowed proofs. We present these theorems in this section.

Section 2.2.5.1 Pawn Capture Theorems

The first of these are the theorems using the pawn capture functions and predicates, *PawnStructure*. *PawnStructure1* is a sufficient condition on the satisfaction of the MAY_PAWN_CAPTURES predicate. It states that if a pawn can move from one square to another in the course of a legal game, without promoting, then the predicate MAY_PAWN_CAPTURES is true between those two squares. We could have defined MAY_PAWN_CAPTURES to be the predicate satisfying this relationship; however, the justification for the associated attachment would then have been much more complex.

The second "theorem" consists of seven parts, of which we have listed three.[46] It states that the value of Pawncaptures for two squares places a minimum bound on the number of capturing moves a pawn has to make to get from its first argument to its second. For each such move, there is a position with the properties such as the pawn made a capture to get to that position, and that these positions are distinct. This theorem is difficult to prove with complete generality within our chess predicate logic system, as it involves both the axiomatization of elementry numerical properties, and a correspondence between a count of objects, and an existentially quantified WFF. Wishing to avoid this hassle, we leave this as an unproven theorem.

---

46 PawnStructure2, PawnStructure3 and PawnStructureX1.

The last pawn capture theorem states that if a pawn is diagonally extended from an earlier position (MUST_PAWN_CAPTURES is true of the two squares) then the pawn must have made a capture on every square within that diagonal segment. The proof of this theorem would involve showing that the MUST_PAWN_CAPTURES predicate remains true if a capture occurs, but goes false, forever to stay false, if the pawn makes any other move. Chess induction would surely be needed, and the difficulty of the resulting proof would, in some sense, be inverse to the specificity of the definition of MUST_PAWN_CAPTURES. Unfortunately, the more usable said definition, the less closely it would correspond to the attachment. A more basic proof would again involve more mathematics then we want to approach. Hence, it is also an unproven theorem.

```
axiom _PawnStructure_:
  ∀r p x sq1 sq2.((Pos(p sq1)=x∧Pos(r sq2)=x∧PREDEGAME(r p)∧
       VALUEP(Val(Prevpos p x)))⊃MAY_PAWN_CAPTURES(sq2 sq1 Piececolor(x))),
  ∀r p x sq1 sq2.((VALUEP(Val(Prevpos p x))∧Pos(p sq1)=x∧Pos(r sq2)=x∧
       PREDEGAME(r p)∧Pawncaptures(sq1 sq2)≥1)⊃
    ∃q1 x1.((PREDEGAME(q1 p)∨q1=p)∧PREDEGAME(r q1)∧
    TAKINGS(Move(q1))∧Mover(Move(q1))=x∧Taken(Move(q1))=x1)),
  ∀r p x sq1 sq2.((VALUEP(Val(Prevpos p x))∧Pos(p sq1)=x∧PREDEGAME(r p)∧
       Pos(r sq2)=x∧Pawncaptures(sq1 sq2)≥2)⊃
    ∃q1 q2 x1 x2.((PREDEGAME(q1 p)∨q1=p)∧PREDEGAME(q2 q1)∧PREDEGAME(r q2)∧
    TAKINGS(Move(q1))∧TAKINGS(Move(q2))∧Mover(Move(q1))=x∧
    Mover(Move(q2))=x∧Taken(Move(q1))=x1∧Taken(Move(q2))=x2));;
axiom _PawnStructureX_:
  ∀r p x sq1 sq2.((VALUEP(Val(Prevpos p x))∧Pos(p sq1)=x∧Pos(r sq2)=x∧
       PREDEGAME(r p)∧Pawncaptures(sq1 sq2)≥3)⊃
    ∃q1 q2 q3 x1 x2 x3.((PREDEGAME(q1 p)∨q1=p)∧PREDEGAME(q2 q1)∧
    PREDEGAME(q3 q2)∧PREDEGAME(r q3)∧TAKINGS(Move(q1))∧
    TAKINGS(Move(q2))∧TAKINGS(Move(q3))∧Mover(Move(q1))=x∧
    Mover(Move(q2))=x∧Mover(Move(q3))=x∧Taken(Move(q1))=x1∧
    Taken(Move(q2))=x2∧Taken(Move(q3))=x3)),
  ∀r p x sq1 sq2.((MUST_PAWN_CAPTURES(sq2 sq1 Piececolor(x))∧
       VALUEP(Val(Prevpos p x))∧Pos(p sq1)=x∧PREDEGAME(r p)∧Pos(r sq2)=x)⊃
    ∀sq3.((sq3=sq1∨(SAMEDIAG(sq1 sq3)∧SAMEDIAG(sq3 sq2)∧
    BETWEEN(Row(sq1),Row(sq3),Row(sq2))))⊃
       ∃q3 x3.((PREDEGAME(q3 p)∨q3=p)∧PREDEGAME(r q3)∧TAKINGS(Move(q3))∧
       Mover(Move(q3))=x∧To(Move(q3))=sq3∧Taken(Move(q3))=x3)));;
```

These theorems enable the usual problem solver tricks involving pawn structures.

Section 2.2.5.2 Other Unproven Theorems

Several other unproven theorems remain to be mentioned. *CheckTypes* states the well known chess fact that on any check, either the piece doing the checking was moved last, an en passant move captured a piece with discovered check, the rook in a castling move made the check, or an ordinary move was made, and the check was a discovered check.

The proof of this theorem would be just a large and painful case analysis, to show that the only change to the board can occur on those squares, so that the conclusion naturally follows.

```
axiom _CheckTypes_:
  ∀p b sq1 sq2 x vpc1 vpc2.((POSITIONINCHECK(p Color(p))∧BOARD(p b)∧
       MOVETO(b vpc1 sq1 sq2)∧Pos(p sq1)=x∧Valueon(b sq1)=vpc1∧
       Valueon(b sq2)=vpc2∧VALUEK(vpc2)∧¬Valuecolor vpc1 =Valuecolor vpc2)⊃
     (Mover Move p=x∨
     (EN_PASSANT(Prevpos p ,p)∧(SQUARE_BETWEEN(sq1,From Move p,sq2)∨
                                SQUARE_BETWEEN(sq1,Takenon Move p,sq2)))∨
     (CASTLING(Prevpos p,p)∧Alsomover Move p=vpc1)∨
     (ORDINARY Move p∧SQUARE_BETWEEN(sq1,From Move p,sq2))));;
```

The theorem ***MoveBack*** states that all moves, except pawn moves, are symmetric. That is, if the piece could move from the first square to the second on a board, then it could move back again.

This theorem could be proven by the use of the simplify mechanism and a lot of manipulation. Particularly useful would be the commutation theorems (section A.9.5).

```
axiom _MoveBack_:
     ∀r p v sq1 sq2.((SUCCESSOR(r p)∧ORDINARY Move p∧¬VALUEP(v))⊃
          (MOVETO(Tboard p,v sq1 sq2)=MOVETO(Tboard r,v sq2 sq1)));;
```

Chapter 3 Chess Lemmas and Theorems

Of course, no complicated proof is achieved without the use of some theorems and lemmas. These serve several functions, somewhat similar to the functions served by procedures in a programming language. They provide structure, pointing out the natural conclusions and breaking points, and increasing general proof readability. They also serve to reduce the actual volume of the proof, permitting the condensation of several similar inferences into a single general scheme and the avoidance of repetition of an identical computation. In the case of computer proof checking, where the memory size of the program and its data structures often needs be minimized, the use of lemmas serves to remove from the main computation large sections of proof, replacing them with only a reference to the desired conclusion. For our axioms, with their many equivalences between the defined terms, they also aid by rephrasing an axiom into a form more usable in another part of the proof. This chapter is devoted to the proof of several representative general chess theorems and lemmas from the basic chess axioms described in the previous chapter. We present several thoughly explained sample lemmas and theorems, with the proofs of the other chess lemmas and theorems, less thoughly explained, in the appendices.

Section 3.1 Simplification Lemmas

Among these lemmas are several that are both trivially deduced, and frequently referenced. These are the lemmas obtained through a single application of the simplify command (a single call to the semantic computation of the chess eye). Many of these refer to the extension of various sorts. For example, the simplification:

```
LABEL WhitepieceAre_;
SIMPLIFY Vt.(WHITEPIECE t=(
      t=WKPvt=WQPvt=WKNPvt=WKBPvt=WKRPvt=WQBPvt=WQNPvt=WQRPv
      t=WK vt=WQ vt=WKN vt=WKB vt=WKR vt=WQB vt=WQN vt=WQR));
```

Gives a membership definition for the set of white pieces. This lemma is henceforth referred to as *WhitepieceAre_*. These simplifications would arise from definition or observation, rather than deduction. In the cases where a large number of individuals must be considered to establish the validity of some WFF, they are also quite slow. These qualities have led us to list them in their own section, section A.1, rather than repeatedly computing them in the different theorems. We will use these lemmas freely in the rest of this paper. In no case should they express any fact whose validity is not observationally apparent to the reader.

Section 3.2 Simple Proofs

Section 3.2.1 Proofs on Positions

We begin with several simple example proofs. First, three lemmas about the PREDEGAME relation. Recall that PREDEGAME(p1,p2) is true if position p1 occurred sometime in the play that reached position p2. In this usage, the PREDEGAME relation is like the predicate < (less than) on a partially ordered, half closed set. The first lemma we prove on this relationship is its transitivity. PREDEGAME is defined either directly by the successor relationship, or recursively in terms of the existence of an intermediate position satisfying PREDEGAME with the original arguments. The position p, common in PREDEGAME to r and q in the assumption, is shown to be the intermediate position for them of the definition.

We shall include comments about the structure of the proof checker in italics.

```
*****label L1;
*****assume PREDEGAME(r,p)∧PREDEGAME(p,q);
1 PREDEGAME(r,p)∧PREDEGAME(p,q)  (1)

*****∃I ↑ p;
2 ∃p.(PREDEGAME(r,p)∧PREDEGAME(p,q))  (1)
```

Here, step one creates a line with the assumption that r *came before* p, *and* p *before* q. *Step two generalizes this to some individual* p. *A series of* n ↑*s implies the* n*th previous step.*

Assigning a label *to a line gives us another method for referring to it. All of the theorems in this volume have the label used in their proof associated with them; temporary label, such as L1 and L2, have been used in many different proofs.*

Part of the definition of the PREDEGAME relation is given in the axiom *GAMEREL1* (game relation 1).

```
*****∀E GAMERELATIONS1 r,q;
3 PREDEGAME(r,q)≡(SUCCESSOR(r,q)∨∃p.(PREDEGAME(r,p)∧PREDEGAME(p,q)))
```

∀E *is used to specialize a universally quantified statement (usually an axiom) to a specific list of individuals.*

PREDEGAME is therefore obviously true of r and q.

```
*****taut PREDEGAME(r,q) ↑↑:↑;
4 PREDEGAME(r,q)  (1)
```

FOL has two tautology deciders, TAUT, *for tautologies of the propositional calculus, and* TAUTEQ *for tautologies of the propositional calculus, including equality. We give the deciders the WFF to be proven, and the reasons (list of previous steps and axioms) from which it follows.*

As we will do for all the theorems in this volume, we remove dependencies, and generalize:

```
*****⊃I L1⊃↑;
5 (PREDEGAME(r,p)∧PREDEGAME(p,q))⊃PREDEGAME(r,q)
```

⊃*I is a natural deduction rule, one that individually introduces (I) or eliminates (E) propositional connectives.* ⊃*I is also useful for removing dependencies.*

```
*****label TransitiveGenealogy;
***** ∀I ↑ r p q;
6 ∀r p q.((PREDEGAME(r,p)∧PREDEGAME(p,q))⊃PREDEGAME(r,q))
```

This last, sixth step is a universal quantification. It asserts that, as the statement is true of some general r, p, *and* q, *it must be true for* all r, p, *and* q.

The next lemma we prove is about the predecessors of positions immediately preceding a given position. If, of two positions, r and p, p is a successor to r, then, for any position that came before p, it either also came before r, or is equal to r. We employ three parts of the defining axiom for

PREDEGAME, *GAMERELATIONS*. The first part (*GAMEREL4*) states that there is no position *between* two successor positions. The second part is a law of the excluded middle for the PREDEGAME relation. If two positions are in the game tree of another position, then they are either equal, or one came before the other. The third part is a repetition of *GAMEREL1*, used above. This time, we employ the fact that the relation SUCCESSOR implies that of PREDEGAME. Together, these three imply that a position has, by and large, the same predecessors as its successors.

```
*****VE GAMERELATIONS4 r1,q,r2;
7 SUCCESSOR(r1,q)⊃¬(PREDEGAME(r1,r2)∧PREDEGAME(r2,q))

*****VE GAMERELATIONS2 q,r1,r2;
8 (PREDEGAME(r2,q)∧PREDEGAME(r1,q))⊃(PREDEGAME(r1,r2)∨(PREDEGAME(r2,r1)∨r2=
r1))

*****VE GAMERELATIONS1 r1,q;
9 PREDEGAME(r1,q)≡(SUCCESSOR(r1,q)∨∃p.(PREDEGAME(r1,p)∧PREDEGAME(p,q)))

*****taut (SUCCESSOR(r1,q)∧PREDEGAME(r2,q)) ⊃ (PREDEGAME(r2,r1)∨r2=r1)  ↑↑↑;
*↑;
10 (SUCCESSOR(r1,q)∧PREDEGAME(r2,q))⊃(PREDEGAME(r2,r1)∨r2=r1)
```

We call this lemma *ParentGenealogy*.

```
*****label ParentGenealogy;
***** ∀ ↑ r2 r1 q;
11 ∀r2 r1 q.((SUCCESSOR(r1,q)∧PREDEGAME(r2,q))⊃(PREDEGAME(r2,r1)∨r2=r1))
```

The last part of our triplet concludes that, since all GAMEPOSITIONS have the initial position in their game trees, and the PREDEGAME relation is defined to be anti-reflexive, that no position can precede the initial position. This lemma is called *GameRelations5*.

```
*****VE GAMERELATIONS3 r,P0;
12 ¬(PREDEGAME(r,P0)∧PREDEGAME(P0,r))

*****VE POSITION_RULES r;
13 GAMEPOSITION r⊃(SUCCESSOR(Prevpos r,r)∧PREDEGAME(P0,r))

*****VE POSITION_TYPES r;
14 ¬(r=P0≡GAMEPOSITION r)

*****tauteq ¬PREDEGAME(r,P0) ↑↑↑;↑;
15 ¬PREDEGAME(r,P0)

*****label GameRelations5;
***** ∀ ↑ r;
16 ∀r.¬PREDEGAME(r,P0)
```

Section 3.2.2 Simple Theorems on Values

As an example of the use of the simplify command on the extensions of sorts, we present the proofs of the lemmas *EmptyIsMT* and *ChesspieceValueThm*.

EmptyIsMT states that having a value of MT is equivalent to being the EMPTY piece. This result is obtained using the theorem *RetainValueColor*, which states that the blackness or whiteness of the value of any piece in any pair of positions is the same. The proof of *RetainValueColor* is in section A.8.3. This proof also twice employs the simplification mechanism. We first check that, in the initial position, having value MT is the same as being the piece EMPTY. Then, each of the VVALUES (values a piece can have) is checked to show that the value MT is the only value that is neither a black value (BVALUES) nor a white value (WVALUES).

```
*****simplify ∀t.(t=EMPTY≡Val(P0 t)=MT);
1 ∀t.(t=EMPTY≡Val(P0,t)=MT)

*****label L1;
*****simplify ∀vvx.((¬BVALUES vvx∧¬WVALUES vvx)≡vvx=MT);
2 ∀vvx.((¬BVALUES vvx∧¬WVALUES vvx)≡vvx=MT)
```

These two facts are certainly true of our typical piece, t, and its values in the initial position, Val(P0, t), and in a general position r (Val(r, t)).

```
*****∀E ↑↑t;
3 t=EMPTY≡Val(P0,t)=MT

*****∀E ↑↑ Val(P0 t);
4 (¬BVALUES Val(P0,t)∧¬WVALUES Val(P0,t))≡Val(P0,t)=MT

*****∀E ↑↑↑ Val(r t);
5 (¬BVALUES Val(r,t)∧¬WVALUES Val(r,t))≡Val(r,t)=MT
```

Our lemma *RetainValueColor* tells us that the color of the value of any piece is the same in all positions.

```
*****∀E RetainValueColor P0 r t;
6 (BVALUES Val(r,t)≡BVALUES Val(P0,t))∧(WVALUES Val(r,t)≡WVALUES Val(P0,t))
```

But if this is the case, then the equivalence between having MT value in the initial position, and being the EMPTY piece, must also hold in the position r.

```
*****taut ↑↑↑↑:#1≡↑↑:#2 ↑↑↑↑:↑;
7 t=EMPTY≡Val(r,t)=MT
```

We generalize this to all POSITIONS and PIECES. Let us call this theorem EmptyIsMT.

```
*****label EmptyIsMT;
*****∀I ↑ r t;
8 ∀r t.(t=EMPTY≡Val(r,t)=MT)
```

We next attempt the lemma *ChesspiecePiecevalueThm*, which states that the value of any CHESSPIECE in any POSITION is always one of the PIECEVALUES. Recall that PIECEVALUES are the VALUES less the empty value (MT), and the undefined value (UD). We first inquire of simplify if all BVALUES and WVALUES are PIECEVALUES.

```
*****simplify ∀vb.PIECEVALUES vb;
9 ∀vb.PIECEVALUES vb
```

```
*****simplify ∀vw.PIECEVALUES vw;
10 ∀vw.PIECEVALUES vw
```

As our lemma EmptyIsMT is true of all PIECES, it must therefore be true of all CHESSPIECES.

```
*****VE EmptyIsMT r x;
11 x=EMPTY∨Val(r,x)=MT
```

And the two simplifications on PIECEVALUES must also be true on the value of x in r. Note the conditions inserted by the VE command.

```
*****VE ↑↑↑ Val(r x);
12 BVALUES Val(r,x)⊃PIECEVALUES Val(r,x)

*****VE ↑↑↑ Val(r x);
13 WVALUES Val(r,x)⊃PIECEVALUES Val(r,x)
```

Simplification can also be used to obtain SORT information.

```
*****simplify CHESSPIECES x∧¬CHESSPIECES EMPTY;
14 CHESSPIECES x∧¬CHESSPIECES EMPTY
```

We also need the simplification of line two, applying it now to Val(r,x).

```
*****VE L1 Val(r x);
15 (¬BVALUES Val(r,x)∧¬WVALUES Val(r,x))∨Val(r,x)=MT
```

Since x is a chesspiece, it is not EMPTY. Therefore, it does not have value MT in any position. But if the value of x in a position is neither black nor white, then it is MT. Hence, x must have either a black or white value in every position. As all such values are PIECEVALUES, x must always have a PIECEVALUES value. A single TAUTEQ gives us this result.

```
*****tauteq PIECEVALUES Val(r x) ↑↑↑↑↑:↑;
16 PIECEVALUES Val(r,x)
```

We generalize, calling the result ChesspiecePieceValueThm.

```
*****label ChesspiecePieceValueThm;
*****VI↑r x;
17 ∀r x.PIECEVALUES Val(r,x)
```

Section 3.3 Chess Inductive Proofs

Section 3.3.1 Only Pawns Promote

Having sampled several simple, small proofs, we next attempt the proof of a more complex and interesting theorem. We want to prove the theorem *OnlyPawnsPromote*, which states that if any piece has a non-pawn value at some point in a game, its value will not subsequently change. This theorem implies that the only piece whose value ever change is a pawn, these only by promotion, and that not pawn ever promotes twice (in one game). *OnlyPawnsPromote* is an interesting example of a *Chess inductive* proof. From this theorem will spin off several useful corollaries, including the fact that pieces of value pawn are always pawns.

We have not explained in detail several of the lemmas used in this proof. Their proofs, with some commentary, may be found in appendix A. In many cases, these lemmas are merely a rephrasing of some axiom.

This proof uses the simplified form of chess induction, which we call *ChsInd*. The general chess induction theorem refers to predicates true in the descendants of a position. The simplified form assumes that the ancestor position is the initial one. As all GAMEPOSITIONs are descended from P0, theorems true of all GAMEPOSITIONs can be easily proven from this form.[47]

The predicate we wish to prove is:

```
∀r r1 t.((¬VALUEP Val (r1 t)∧PREDEGAME(r1 r))⊃Val(r t)=Val(r1 t))
```

That is, if in some position r1, a piece t does not have the value of pawn, then, in any descendant of r1, r, then the value of t is the same in r1 as r1.

We substitute this predicate for the predicate parameter α in *ChsInd*.

```
*****label L1;
******∧I ChsInd[α←λr.∀r1 t.((¬VALUEP Val(r1 t)∧PREDEGAME(r1 r))⊃Val(r t)=Val(
*r1 t))];
1 (∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,P0))⊃Val(P0,t)=Val(r1,t))∧∀r p.((∀
r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))∧SUCCESSOR(r,p)
)⊃∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,p))⊃Val(p,t)=Val(r1,t))))⊃∀r r1 t.(
(¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))
```

First, we must establish the validity of the proposition in the initial position (P0). As no position is a predecessor to the initial position, this is trivial.

```
*****∀E GameRelations5 r1;
2 ¬PREDEGAME(r1,P0)

*****taut L1:#1#1#1#1 ↑;
3 (¬VALUEP Val(r1,t)∧PREDEGAME(r1,P0))⊃Val(P0,t)=Val(r1,t)

*****label L2;
***** ∀I ↑ r1 t;
4 ∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,P0))⊃Val(P0,t)=Val(r1,t))
```

We now make two assumptions. As the inductive form is *assume its true of n, prove it is true of n+1*, we assume the validity of the theorem in position r, trying to prove its validity in its successor p. Secondly, as the sentence we are trying to prove of p is also of the form $A \supset B$, we assume the A part, seeking B. Note this sequence; it is our general schema for chess inductive proofs.

```
*****label L3;
***** assume L1:#1#2#1#1#1;
5 ∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))∧SUCCESSOR(r
,p)  (5)

*****assume L1:#1#2#1#1#2#1#1#1;
6 ¬VALUEP Val(r1,t)∧PREDEGAME(r1,p)  (6)
```

47. A derivation of ChsInd from CHESS_INDUCTION is in section A.2.

```
*****∧E L3:#1;
7 ∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))  (5)
```

There are three pertinent positions in this proof. We seek to prove that the Val of t is the same in both positions r1 and p. We have assumed that the Val is the same between r1 and r, the position previous to (Prevpos) of p.

```
*****∀E ↑ r1,t;
8 (¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t)  (5)
```

We need to show that r1 is also a predecessor of p. Our lemma ***ParentGenealogy***[48] is used to establish this.

```
*****∀E ParentGenealogy r1,r,p;
9 (SUCCESSOR(r,p)∧PREDEGAME(r1,p))⊃(PREDEGAME(r1,r)∨r1=r)
```

The heart of this proof lies with the axiom that states that pieces change value only when they move in a pawn promotion. The axiom *MCONSEQH*, part of the move definitional axioms, tells us that, between a position r and its successor p, if the piece, t, was not the mover of p, or p was not a pawn promotion, then t retains the same value from r to p.

```
*****label L4;
***** ∀E MCONSEQH r,p,t;
10 (SUCCESSOR(r,p)∧(¬PAWNPROM Move p∨¬(t=Mover Move p)))⊃Val(r,t)=Val(p,t)
```

We have a special case to consider: when the position r1 is the same as the position r (line 9). TAUTEQ will not make the substitution in functions for us, we must do it ourselves. Assume they are the same.

```
*****label L5;
***** assume r=r1;
11 r=r1  (11)

*****subst L5 in L3↑1;
12 ¬VALUEP Val(r,t)∧PREDEGAME(r,p)  (6 11)

*****substr L5 in L4;
13 (SUCCESSOR(r1,p)∧(¬PAWNPROM Move p∨¬(t=Mover Move p)))⊃Val(r1,t)=Val(p,t)
  (11)
```

SUBST *and* SUBSTR *substitute for equals in WFFs.*

If they are the same, then, as t does not have a pawn value in r1, it will not have one in r.

```
*****label L6;
***** ⊃I L5⊃↑↑;
14 r=r1⊃(¬VALUEP Val(r,t)∧PREDEGAME(r,p))  (6)

*****⊃I L5⊃↑↑;
15 r=r1⊃((SUCCESSOR(r1,p)∧(¬PAWNPROM Move p∨¬(t=Mover Move p)))⊃Val(r1,t)=
Val(p,t))
```

---

48 Section 3.2.1.

By the definition of pawn promotional moves, the moving piece is a pawn on the total board of the move.

```
*****VE MCONSEQL p;
16 PAWNPROM Move p=(LASTRANKER(To Move p,Color Prevpos p)∧(SIMPLELEGALMOVE(
Prevpos p,p)∧(PAWNS Mover Move p∧(VALUEP Valueon(Tboard Prevpos p,From Move
p)∧(((BVALUES Promoted Move p=BVALUES Val(Prevpos p,Mover Move p))∧(WVALUES
Promoted Move p=WVALUES Val(Prevpos p,Mover Move p)))∧Val(p,Mover Move p)=
Promoted Move p)))))
```

By the definition of SUCCESSOR, the previous position (Prevpos) of a position shares the SUCCESSOR relation.

```
*****VE MCONSEQA r,p;
17 SUCCESSOR(r,p)⊃((¬WHITETURN r=WHITETURN p)∧(Prevpos p=r∧(¬POSITIONINCHECK
(p,Color r)∧((WHITEPIECE Mover Move p=WHITETURN r)∧(Pos(r,From Move p)=Mover
 Move p∧(Pos(p,To Move p)=Mover Move p∧(Pos(p,From Move p)=EMPTY∧((CAPTURE
Move p⊃Pos(r,To Move p)=Taken Move p)∧(CASTLING(r,p)∨(EN_PASSANT(r,p)∨
SIMPLELEGALMOVE(r,p)))))))))))

*****taut Prevpos p=r L3,↑;
18 Prevpos p=r  (5)

*****substr ↑ in ↑↑↑;
19 PAWNPROM Move p=(LASTRANKER(To Move p,Color r)∧(SIMPLELEGALMOVE(r,p)∧(
PAWNS Mover Move p∧(VALUEP Valueon(Tboard r,From Move p)∧(((BVALUES Promoted
 Move p=BVALUES Val(r,Mover Move p))∧(WVALUES Promoted Move p=WVALUES Val(r,
Mover Move p)))∧Val(p,Mover Move p)=Promoted Move p)))))  (5)
```

A mention of the equivalence of the Val and Valueon functions.

```
*****VE ValueTranspositionA r,Mover Move p,From Move p;
20 Pos(r,From Move p)=Mover Move p⊃Valueon(Tboard r,From Move p)=Val(r,Mover
 Move p)
```

More substitutions for the sake of TAUTEQ.

```
*****label L7;
***** assume t=Mover Move p;
21 t=Mover Move p  (21)

*****subst L7 in ↑↑ occ 2;
22 Pos(r,From Move p)=Mover Move p⊃Valueon(Tboard r,From Move p)=Val(r,t)  (
21)

*****⊃I L7⊃↑;
23 t=Mover Move p⊃(Pos(r,From Move p)=Mover Move p⊃Valueon(Tboard r,From
Move p)=Val(r,t))
```

We have a form that can be handled by TAUTEQ. One invocation produces our desired identity.

```
*****tauteq Val(p,t)=Val(r1,t) L3:L4,L6:20,↑;
24 Val(p,t)=Val(r1,t)  (5 6)
```

We remove the dependencies, and insert the universal quantifiers in the proper order so as to obtain the theorem.

```
*****⊃I 6⊃↑;
25 (¬VALUEP Val(r1,t)∧PREDEGAME(r1,p))⊃Val(p,t)=Val(r1,t)  (5)

*****∀I ↑ r1 t;
26 ∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,p))⊃Val(p,t)=Val(r1,t))  (5)

*****⊃I L3⊃↑;
27 (∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))∧SUCCESSOR
(r,p))⊃∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,p))⊃Val(p,t)=Val(r1,t))

*****∀I ↑ r p;
28 ∀r p.((∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))∧
SUCCESSOR(r,p))⊃∀r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,p))⊃Val(p,t)=Val(r1,t
)))
```

We have satisfied the two conditions of chess induction. Our theorem naturally follows.

```
*****label OnlyPawnsPromote;
***** taut L1:#2    L1,L2,↑;
29 ∀r r1 t.((¬VALUEP Val(r1,t)∧PREDEGAME(r1,r))⊃Val(r,t)=Val(r1,t))
```

Section 3.3.2 Mobility

Another example of a proof by chess induction. We wish to prove that if any chesspiece is on a square differing from the one it started upon, then there must have existed a position, earlier in that game, where that piece moved.[49] We take this proposition, and substitute it for the predicate parameter α in *ChsInd*.

```
*****label L1;
*****∀I ChsInd[α←λp.∀sq x.((Pos(p,sq)=x∧¬Pos(P0,sq)=x)⊃ ∃q.((PREDEGAME(q,p)∨
*q=p)∧ ((Mover Move q=x∧To Move q=sq)∨ (CASTLE Move q∧Alsomover Move q=x∧
*Alsoto Move q=sq)))))];
1 (∀sq x.((Pos(P0,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,P0)∨q=P0)∧((Mover
Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))
)))∧∀r p.((∀sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((
Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move
q=sq)))))∧SUCCESSOR(r,p))⊃∀sq x.((Pos(p,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((
PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover
 Move q=x∧Alsoto Move q=sq)))))))⊃∀r sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.
((PREDEGAME(q,r)∨q=r)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq)))))
```

As this theorem refers to a position where the piece is on a different square from the initial position, it automatically is true of the initial position.

```
*****taut ↑:#1#1#1#1 ;
2 (Pos(P0,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,P0)∨q=P0)∧((Mover Move q=x
∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))))
```

---

49 The notion of moved, in this context, includes both being the mover of a particular position, or being the rook in a castling move.

```
*****∀I↑sq x;
3 ∀sq x.((Pos(P0,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,P0)∨q=P0)∧((Mover
Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))
)))
```

Following the form of the other chess inductive proofs, we make two assumptions. The first assumption is that the theorem is true in some position r; we then seek to prove its validity in a successor of r, p. The theorem itself is of the form A⊃b, we assume A, and work to conclude b.

```
*****label L2;
*****assume L1:#1#2#1#1#1;
4 ∀sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((Mover Move
 q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq)))))∧
SUCCESSOR(r,p)  (4)

*****assume L1:#1#2#1#1#1#2#1#1#1;
5 Pos(p,sq)=x∧¬(Pos(P0,sq)=x)  (5)
```

Let us call the chesspiece in question x, and the square it is on in p, sq. The first half of the first assumption is therefore true of x and sq.

```
*****∧E↑↑ :#1 ;
6 ∀sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((Mover Move
 q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq)))))
 (4)

*****label L5;
*****∀E ↑ sq , x ;
7 (Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((Mover Move q=x∧To
 Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))))  (4)
```

We have to consider, in this problem, two cases. Either the piece x is one the same square in both p and r, or it has changed location in the transition between positions. We examine first the occasion when it is on the same square in each.

```
*****label L3;
*****assume Pos(p sq)=Pos(r sq);
8 Pos(p,sq)=Pos(r,sq)  (8)
```

By our assumption, there exists some position, a predecessor of r, in which x was the moving piece, and it moved to sq. Let us call this position q.

```
*****tauteq ↑↑:#2 ↑,↑↑↑↑,↑↑;
9 ∃q.((PREDEGAME(q,r)∨q=r)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5 8)

*****∃E↑q;
10 (PREDEGAME(q,r)∨q=r)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq)))  (10)
```

But by the lemma *GrandparentGenealogyY*, this position q is also a predecessor to p. Hence, we have a position to satisfy the theorem for p.

```
*****∀E GrandparentGenealogyY q r p;
```

```
11 (SUCCESSOR(r,p)∧(PREDEGAME(q,r)∨q=r))⊃PREDEGAME(q,p)

*****taut (↑:#2∨q=p)∧↑↑:#2 ↑,↑↑,L2;
12 (PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq)))  (4 10)

*****∃I↑q;
13 ∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5 8)

*****label L4;
*****⊃I L3⊃↑;
14 Pos(p,sq)=Pos(r,sq)⊃∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=
sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5)
```

We consider the other possibility. If the occupant of sq in r is not the same as in p.

```
*****label L6;
*****assume ¬Pos(p sq)=Pos(r sq);
15 ¬(Pos(p,sq)=Pos(r,sq))  (15)
```

We consider the various ways the piece x could have changed squares. We have a theorem that states that the only way the contents of a square changes between positions is if it is either the source or destination of a move (or castle), or is the square vacated by a piece captured en_passant.

```
*****∀E MCONSEQD r,p,sq;
16 (SUCCESSOR(r,p)∧(¬(sq=From Move p)∧(¬(sq=To Move p)∧(¬(CASTLE Move p∧(sq=
Alsofrom Move p∨sq=Alsoto Move p))∧¬(ENPASSANT Move p∧sq=Takenon Move p)))))
⊃Pos(r,sq)=Pos(p,sq)
```

We know from the axioms about successors, that after a move the source square (From, Alsofrom) is occupied by the piece EMPTY. The square of a piece captured en_passant is likewise vacant. And EMPTY is not a chesspiece (it is, of course, one of the PIECES).

```
*****∀E MCONSEQA r,p;
17 SUCCESSOR(r,p)⊃((¬WHITETURN r=WHITETURN p)∧(Prevpos p=r∧(¬POSITIONINCHECK
(p,Color r)∧((WHITEPIECE Mover Move p=WHITETURN r)∧(Pos(r,From Move p)=Mover
 Move p∧(Pos(p,To Move p)=Mover Move p∧(Pos(p,From Move p)=EMPTY∧((CAPTURE
Move p⊃Pos(r,To Move p)=Taken Move p)∧(CASTLING(r,p)∨(EN_PASSANT(r,p)∨
SIMPLELEGALMOVE(r,p))))))))))

*****∀E NotChesspieceEmpty_ x;
18 ¬CHESSPIECES x=x=EMPTY
```

All moves are of one of the three types.

```
*****∀E MconseqmX r,p;
19 SUCCESSOR(r,p)⊃((CASTLE Move p=CASTLING(r,p))∧((ENPASSANT Move p=EN_
PASSANT(r,p))∧(ORDINARY Move p=SIMPLELEGALMOVE(r,p))))
```

And no piece is on two different squares in the same position.

```
*****∀E Unique p,sq,To Move p,x;
20 Pos(p,sq)=x⊃(Pos(p,To Move p)=x=sq=To Move p)
```

```
*****∀E Unique p,sq,From Move p,x;
21 Pos(p,sq)=x⊃(Pos(p,From Move p)=x=sq=From Move p)
```

We search the castling and en_passant rules for their special cases.

```
*****∀E CASTLEMOVES r,p;
22 CASTLING(r,p)=(KINGS Mover Move p∧(ROOKS Alsomover Move p∧(Pos(r,Alsofrom
 Move p)=Alsomover Move p∧(Pos(p,Alsofrom Move p)=EMPTY∧(Pos(p,Alsoto Move p
)=Alsomover Move p∧(∀r1.(PREDEGAME(r1,p)⊃Pos(r,From Move p)=Mover Move p)∧(∀
r1.(PREDEGAME(r1,p)⊃Pos(r,Alsofrom Move p)=Alsomover Move p)∧(∀sq3.((Row sq3
=Row From Move p∧BETWEEN(Column From Move p,Column sq3,Column Alsofrom Move
p))⊃Pos(r,sq3)=EMPTY)∧(¬POSITIONINCHECK(r,Color r)∧(∀sq1 x.¬(Pos(r,sq1)=x∧(
MOVETO(Tboard r,Val(r,x),sq1,Alsoto Move p)∧Piececolor x=Color p))∧((
WHITETURN r⊃((Alsomover Move p=WKR∧(Alsoto Move p=WKB1∧To Move p=WKN1))∨(
Alsomover Move p=WQR∧(Alsoto Move p=WQ1∧To Move p=WQB1))))∧(¬WHITETURN r⊃((
Alsomover Move p=BKR∧(Alsoto Move p=BKB1∧To Move p=BKN1))∨(Alsomover Move p=
BQR∧(Alsoto Move p=BQ1∧To Move p=BQB1)))))))))))))))

*****∀E Unique p,sq,Alsoto Move p,x;
23 Pos(p,sq)=x⊃(Pos(p,Alsoto Move p)=x=sq=Alsoto Move p)

*****∀E Unique p,sq,Alsofrom Move p,x;
24 Pos(p,sq)=x⊃(Pos(p,Alsofrom Move p)=x=sq=Alsofrom Move p)

*****∀E ENPASS r,p;
25 EN_PASSANT(r,p)=(GAMEPOSITION r∧(Pos(p,Takenon Move p)=EMPTY∧(To Move r=
Takenon Move p∧(Mover Move r=Taken Move p∧(SIM Move r∧(Column From Move r=
Column To Move r∧(Column To Move r=Column To Move p∧(TWOTOUCHING(Column From
 Move p,Column To Move p)∧((WHITETURN p⊃(Val(p,Mover Move p)=PB∧(Val(r,Mover
 Move r)=PW∧(Row From Move p=5∧(Row To Move p=6∧(Row From Move r=7∧Row To
Move r=5))))))∧(¬WHITETURN p⊃(Val(p,Mover Move p)=PW∧(Val(r,Mover Move r)=PB
∧(Row From Move p=4∧(Row To Move p=3∧(Row From Move r=2∧Row To Move r=4)))))
))))))))))

*****∀E Unique p,sq,Takenon Move p,x;
26 Pos(p,sq)=x⊃(Pos(p,Takenon Move p)=x=sq=Takenon Move p)

*****simplify(CASTLE Move p⊃SQUARES(Alsoto Move p))∧(CASTLE Move p⊃SQUARES(
*Alsofrom Move p))∧ (ENPASSANT Move p⊃SQUARES(Takenon Move p))∧ CHESSPIECES
*x;
27 (CASTLE Move p⊃SQUARES Alsoto Move p)∧((CASTLE Move p⊃SQUARES Alsofrom
Move p)∧((ENPASSANT Move p⊃SQUARES Takenon Move p)∧CHESSPIECES x))
```

It therefore tautologically follows, that in the move that created p, x and sq must have performed the desired roles.

```
*****tauteq ((PREDEGAME(p p)∨p=p)∧((Mover Move p=x∧To Move p=sq)∨(CASTLE
*Move p∧(Alsomover Move p=x∧Alsoto Move p=sq)))) L6:↑,L2:L5;
28 (PREDEGAME(p,p)∨p=p)∧((Mover Move p=x∧To Move p=sq)∨(CASTLE Move p∧(
Alsomover Move p=x∧Alsoto Move p=sq)))  (4 5 15)
```

p is thus seen to be the position whose existence we were trying to prove.

```
*****∃I↑p←q occ 1 3 5 6 7 8 9;
29 ∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5 15)
```

All this was, of course, based on the assumption that the piece was on a different square.

```
*****⊃I L6⊃↑;
30 ¬(Pos(p,sq)=Pos(r,sq))⊃∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move
q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5)
```

We have obtained the desired WFF in both cases; when x had changed squares, and when x had not. It is therefore always true.

```
*****taut ↑:#2 ↑,L4;
31 ∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(
Alsomover Move q=x∧Alsoto Move q=sq))))  (4 5)
```

We insert the assumptions back in the correct order, so as to obtain the premises for the chess induction form.

```
*****⊃I L2+1⊃↑;
32 (Pos(p,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧
To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))))  (4)

*****∀I↑sq x;
33 ∀sq x.((Pos(p,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,p)∨q=p)∧((Mover
Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))
)))  (4)

*****⊃I L2⊃↑;
34 (∀sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((Mover
Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))
)))∧SUCCESSOR(r,p))⊃∀sq x.((Pos(p,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,p)
∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧
Alsoto Move q=sq)))))

*****∀I↑r p;
35 ∀r p.((∀sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((
Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move
q=sq)))))∧SUCCESSOR(r,p))⊃∀sq x.((Pos(p,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((
PREDEGAME(q,p)∨q=p)∧((Mover Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover
 Move q=x∧Alsoto Move q=sq))))))
```

Having satisfied both requirements, the theorem is now ours. We call it *Mobility*.

```
*****label Mobility;
*****taut L1:#2 ↑,L2-1,L1;
36 ∀r sq x.((Pos(r,sq)=x∧¬(Pos(P0,sq)=x))⊃∃q.((PREDEGAME(q,r)∨q=r)∧((Mover
Move q=x∧To Move q=sq)∨(CASTLE Move q∧(Alsomover Move q=x∧Alsoto Move q=sq))
)))
```

Section 3.3.3 Segregate

For our last example of a chess inductive proof, we prove the well known chess fact that bishops stay on squares of the same color. The key predicate for this proof is WHITESQUARES, a sort on squares, which is true, of course, on the white squares.

In proving this theorem we employ the lemma *BishopStaysOnSameColor*.[50] *BishopStaysOnSameColor* states that between any position, r, and a successor, p, if a bishop ybi is on square sq1 in r, and sq in p, then sq1 and sq are of the same color. Expressed as a WFF in our axiomatization, this is:

```
∀r p ybi sq1 sq.((SUCCESSOR(r,p)∧(Pos(p,sq)=ybi∧Pos(r,sq1)=ybi))⊃
                 (WHITESQUARES(sq)=WHITESQUARES(sq1)));;
```

This theorem also employs the lemma *WasHere*, which states that for any piece x, in a position p, if x is on some square in p, then x was on some square in the position previous to p.[51]

The proof of *BishopsIsOnSameColor* follows the form of our other chess inductive proofs. First the simplified form of chess induction, *ChsInd*, is instantiated with the theorem to be proven.

```
*****label L1;
******∧I ChsInd[α←λp.(∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(p,sq2)=ybi)⊃ (
*WHITESQUARES(sq1)=WHITESQUARES(sq2))))];
1 (∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(P0,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))∧∀r p.((∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(
WHITESQUARES sq1=WHITESQUARES sq2))∧SUCCESSOR(r,p))⊃∀sq1 sq2 ybi.((Pos(P0,
sq1)=ybi∧Pos(p,sq2)=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2))))⊃∀r sq1 sq2
ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2))
```

Proving the proposition for the initial position is trivial. No piece can be on more than one square in any position. So, of course, our bishop ybi is on the same color square in P0 as in P0.

```
*****∀E Unique P0,sq1,sq2,ybi;
2 Pos(P0,sq1)=ybi⊃(Pos(P0,sq2)=ybi=sq1=sq2)

*****tauteq (Pos(P0,sq1)=ybi∧Pos(P0,sq2)=ybi)⊃(WHITESQUARES(sq1)=
*WHITESQUARES(sq2)) ↑;
3 (Pos(P0,sq1)=ybi∧Pos(P0,sq2)=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2)

*****label L2;
***** ∀I ↑ sq1 sq2 ybi;
4 ∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(P0,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))
```

We make the two usual assumptions for chess inductive proofs.

```
*****label L3;
*****assume L1:#1#2#1#1#1;
5 ∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))∧SUCCESSOR(r,p)  (5)

*****assume Pos(P0,sq1)=ybi∧Pos(p,sq2)=ybi;
6 Pos(P0,sq1)=ybi∧Pos(p,sq2)=ybi  (6)

******∧E L3:#1;
7 ∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))  (5)
```

---

50. The proof of BishopStaysOnSameColor is in section A.9.3.2.

51. The proof of WasHere is in section A.11.1.2.1.

As ybi is on square sq2 in p, it must have been on some square in r. Let us call that square sq.

```
*****VE WasHere r,p,sq2,ybi;
8 (SUCCESSOR(r,p)∧Pos(p,sq2)=ybi)⊃∃sq.Pos(r,sq)=ybi

*****taut ∃sq.Pos(r,sq)=ybi L3:↑↑↑,↑;
9 ∃sq.Pos(r,sq)=ybi  (5 6)

*****∃E ↑ sqx;
10 Pos(r,sqx)=ybi  (10)
```

This square sq is, by our assumption, the same color as the square ybi started on (sq1)

```
*****VE ↑↑↑↑ sq1,sqx,ybi;
11 (Pos(P0,sq1)=ybi∧Pos(r,sqx)=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sqx)  (5)
```

And by the lemma *BishopStaysOnSameColor*, it is the same color as the square ybi is on in p (sq2).

```
*****VE BishopStaysOnSameColor r,p,ybi,sqx,sq2;
12 (SUCCESSOR(r,p)∧(Pos(p,sq2)=ybi∧Pos(r,sqx)=ybi))⊃(WHITESQUARES sq2=
WHITESQUARES sqx)
```

So it obviously follows that the initial square, sq1 is the same color as the final square, sq2.

```
*****tauteq WHITESQUARES(sq1)=WHITESQUARES(sq2) L3:6,↑↑↑:↑;
13 WHITESQUARES sq1=WHITESQUARES sq2  (5 6)
```

We remove the dependencies and generalize in the appropriate order.

```
*****⊃I L3+1⊃↑;
14 (Pos(P0,sq1)=ybi∧Pos(p,sq2)=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2)  (5)

*****∀I ↑ sq1 sq2 ybi;
15 ∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(p,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))  (5)

*****⊃I L3⊃↑;
16 (∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))∧SUCCESSOR(r,p))⊃∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(p,sq2)
=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2))

*****∀I ↑ r p;
17 ∀r p.((∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))∧SUCCESSOR(r,p))⊃∀sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(p,sq2)
=ybi)⊃(WHITESQUARES sq1=WHITESQUARES sq2)))
```

Having satisfied both chess inductive requirements, we have our theorem.

```
*****label BishopsIsOnSameColor;
*****taut L1:#2 L1,L2,↑;
18 ∀r sq1 sq2 ybi.((Pos(P0,sq1)=ybi∧Pos(r,sq2)=ybi)⊃(WHITESQUARES sq1=
WHITESQUARES sq2))
```

Section 3.4 More Complex Chess Theorems

Section 3.4.1 Proof by Cases: Symmetric Orthogonality

The ORTHO relation, on a board and two squares, is true if the argument squares are on the same orthogonal (row or column), and all squares between the two are empty on that board. It is used in defining the rook and queen moves. There is an attachment to ORTHO that, given a board and squares, will compute the value of the ortho relation. However, much as LISP can not compute *call by name* function evaluations, the simplify mechanism cannot handle simplifications of equally fragmentary information. We will have occasion to conclude the ORTHO relation on sub-boards from that on of total boards, and vice-versa.

One can conclude this equivalence, of course, when none of the squares between the given squares is undefined.

The proof itself is an example of proof by cases. We will have to prove the theorem for both rows and the columns, and in each direction. We will accomplish this by the use of four *parallel* proof threads, which, properly Riemannian, will converge to the our theorem.

We begin by assuming that board a is a sub-board of of board b, that our two squares, sq1 and sq2, are different, and that either the two squares are in the same column, and every square between them on that column is not undefined (UD) on a; or that they share the same row, and every square between them on that row is not UD.

```
*****label L1;
*****assume SUBOARD(a,b)∧(¬(sq1=sq2)∧ ((Column(sq1)=Column(sq2)∧ ∀sq3.((
*BETWEEN(Row(sq1),Row(sq3),Row(sq2))∧Column(sq3)=Column(sq1))⊃ ¬(Valueon(a,
*sq3)=UD))) ∨(Row(sq1)=Row(sq2)∧ ∀sq3.((BETWEEN(Column(sq1),Column(sq3),
*Column(sq2))∧Row(sq3)=Row(sq1))⊃ ¬(Valueon(a,sq3)=UD)))));
1 SUBOARD(a,b)∧(¬(sq1=sq2)∧((Column sq1=Column sq2∧∀sq3.((BETWEEN(Row sq1,
Row sq3,Row sq2)∧Column sq3=Column sq1)⊃¬(Valueon(a,sq3)=UD)))∨(Row sq1=Row
sq2∧∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃¬(
Valueon(a,sq3)=UD))))) (1)
```

If a is a sub-board of b, then they differ only on the squares where a is undefined.

```
*****VE SUB_BOARDS4 a , b ;
2 SUBOARD(a,b)=∀sq.(Valueon(a,sq)=Valueon(b,sq)∨Valueon(a,sq)=UD)

*****taut ↑:#2 ↑,↑↑;
3 ∀sq.(Valueon(a,sq)=Valueon(b,sq)∨Valueon(a,sq)=UD) (1)
```

Let us call the typical square between sq1 and sq2, sq3. Either this square is undefined (UD) on a, or it has the same Valueon it in both a and b.

```
*****label L8;
*****VE ↑ sq3;
4 Valueon(a,sq3)=Valueon(b,sq3)∨Valueon(a,sq3)=UD (1)
```

We invoke the lemma *RowColumnSquareThm*, which states that if two squares have the same row

and column, they are equal.[52]

```
*****VE RowColumnSquareTnm sq1 , sq2 ;
5 Row sq1=Row sq2⊃(Column sq1=Column sq2⊃sq1=sq2)
```

Since sq1 and sq2 are assumed to be unequal, they must differ in row or column.

We consider each possibility. They might be equal by columns, or equal by rows.

```
*****label L2;
*****assume Column(sq1)=Column(sq2);
6 Column sq1=Column sq2  (6)

*****assume Row(sq1)=Row(sq2);
7 Row sq1=Row sq2  (7)
```

The definition of ORTHO, applied to both a and b.

```
*****label L5;
*****VE MOVING2 a , sq1 , sq2;
8 ORTHO(a,sq1,sq2)=(¬(sq1=sq2)∧((Column sq1=Column sq2∧∀sq3.((BETWEEN(Row
sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(a,sq3)=MT))∨(Row sq1=Row
 sq2∧∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃
Valueon(a,sq3)=MT))))

*****VE MOVING2 b , sq1 , sq2;
9 ORTHO(b,sq1,sq2)=(¬(sq1=sq2)∧((Column sq1=Column sq2∧∀sq3.((BETWEEN(Row
sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(b,sq3)=MT))∨(Row sq1=Row
 sq2∧∀sq^ .,BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃
Valueon(b,sq3)=MT))))
```

As we seek to prove equivalence, we assume each of the ortho conditions and try to prove the other.

```
*****label L3;
*****assume ↑↑:#1;
10 ORTHO(a,sq1,sq2)  (10)

*****assume ↑↑:#1;
11 ORTHO(b,sq1,sq2)  (11)
```

There are now four parallel cases through the proof, determined by whether the presumed orthogonality is horizontal or vertical, and on board b, or its sub-board, a. Note the dependencies.

We can conclude, in each case, from our assumptions and the definition of orthogonality, that every square between sq1 and sq2 is MT.

```
*****label L4;
*****tauteq L5:#2#2#1#2 L5,L2,L1,L8+1,L3;
12 Vsq3.((BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(a,
sq3)=MT)  (1 6 10)

*****tauteq L5:#2#2#2#2 L5,L2+1,L1,L8+1,L3;
```

52. The proof of this lemma is in section A.4.1.

```
13 Vsq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon
(a,sq3)=MT)  (1 7 10)

*****tauteq L5+1:#2#2#1#2 L5+1,L2,L1,L8+1,L3+1;
14 Vsq3.((BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(b,
sq3)=MT)  (1 6 11)

*****tauteq L5+1:#2#2#2#2 L5+1,L2+1,L1,L8+1,L3+1;
15 Vsq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon
(b,sq3)=MT)  (1 7 11)
```

We apply this fact to our typical square, sq3.

```
*****label L6;
*****VE ↑↑↑↑ sq3;
16 (BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(a,sq3)=
MT  (1 6 10)

*****VE ↑↑↑↑ sq3;
17 (BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon(a,sq3
)=MT  (1 7 10)

*****VE ↑↑↑↑ sq3;
18 (BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(b,sq3)=
MT  (1 6 11)

*****VE ↑↑↑↑ sq3;
19 (BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon(b,sq3
)=MT  (1 7 11)
```

sq3 must either be the same on both boards, or undefined on a. By our assumption, all squares on a between sq1 and sq2 are not undefined. Therefore, sq3 will have the same Valueon it in both boards.

```
*****tauteq L1:#2#2#1#2 L1,L2,L8+1;
20 Vsq3.((BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃¬(Valueon(
a,sq3)=UD))  (1 6)

*****tauteq L1:#2#2#2#2 L1,L2+1,L8+1;
21 Vsq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃¬(
Valueon(a,sq3)=UD))  (1 7)

*****label L7;
*****VE ↑↑ sq3;
22 (BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃¬(Valueon(a,sq3)
=UD)  (1 6)

*****VE ↑↑ sq3;
23 (BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃¬(Valueon(a,
sq3)=UD)  (1 7)
```

And, in each case, this value will be MT.

```
*****label L9;
*****tauteq L7:#1⊃Valueon(b sq3)=MT L7,L6,L8;
```

```
24 (BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(b,sq3)=
MT  (1 6 10)

*****tauteq L7+1:#1⊃Valueon(b sq3)=MT L7+1,L6+1,L8;
25 (BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon(b,sq3
)=MT  (1 7 10)

*****tauteq L7:#1⊃Valueon(a sq3)=MT L7,L6+2,L8;
26 (BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(a,sq3)=
MT  (1 6 11)

*****tauteq L7+1:#1⊃Valueon(a sq3)=MT L7+1,L6+3,L8;
27 (BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon(a,sq3
)=MT  (1 7 11)
```

We generalize this result to all squares sq3.

```
*****V|↑↑↑↑sq3;
28 ∀sq3.((BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(b,
sq3)=MT)  (1 6 10)

*****V|↑↑↑↑sq3;
29 ∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon
(b,sq3)=MT)  (1 7 10)

*****V|↑↑↑↑sq3;
30 ∀sq3.((BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃Valueon(a,
sq3)=MT)  (1 6 11)

*****V|↑↑↑↑sq3;
31 ∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃Valueon
(a,sq3)=MT)  (1 7 11)
```

But this is the defining condition for ORTHO on the other board.

```
*****tauteq ORTHO(b sq1 sq2) ↑↑↑↑,L5+1,L1,L8+1,L2;
32 ORTHO(b,sq1,sq2)  (1 6 10)

*****tauteq ORTHO(b sq1 sq2) ↑↑↑↑,L5+1,L1,L8+1,L2+1;
33 ORTHO(b,sq1,sq2)  (1 7 10)

*****tauteq ORTHO(a sq1 sq2) ↑↑↑↑,L5,L1,L8+1,L2;
34 ORTHO(a,sq1,sq2)  (1 6 11)

*****tauteq ORTHO(a sq1 sq2) ↑↑↑↑,L5,L1,L8+1,L2+1;
35 ORTHO(a,sq1,sq2)  (1 7 11)
```

We remove the dependencies of each case assumption.

```
*****⊃I L3⊃↑↑↑↑;
36 ORTHO(a,sq1,sq2)⊃ORTHO(b,sq1,sq2)  (1 6)

*****⊃I L3⊃↑↑↑↑;
37 ORTHO(a,sq1,sq2)⊃ORTHO(b,sq1,sq2)  (1 7)
```

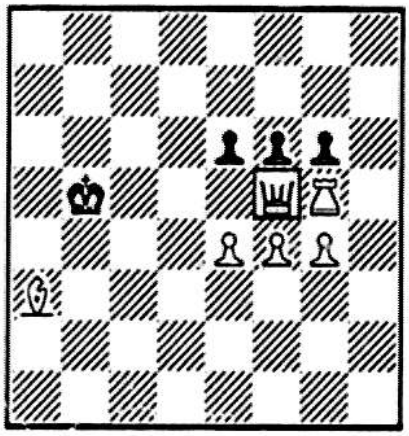
```
******⊃I L3+1⊃↑↑↑↑;
38 ORTHO(b,sq1,sq2)⊃ORTHO(a,sq1,sq2)  (1 6)

******⊃I L3+1⊃↑↑↑↑;
39 ORTHO(b,sq1,sq2)⊃ORTHO(a,sq1,sq2)  (1 7)

******⊃I L2⊃↑↑↑↑;
40 Column sq1=Column sq2⊃(ORTHO(a,sq1,sq2)⊃ORTHO(b,sq1,sq2))  (1)

******⊃I L2+1⊃↑↑↑↑;
41 Row sq1=Row sq2⊃(ORTHO(a,sq1,sq2)⊃ORTHO(b,sq1,sq2))  (1)

******⊃I L2⊃↑↑↑↑;
42 Column sq1=Column sq2⊃(ORTHO(b,sq1,sq2)⊃ORTHO(a,sq1,sq2))  (1)

******⊃I L2+1⊃↑↑↑↑;
43 Row sq1=Row sq2⊃(ORTHO(b,sq1,sq2)⊃ORTHO(a,sq1,sq2))  (1)
```

Having proven the theorem for each case, we can conclude that it is always true.

```
*****tauteq ORTHO(a sq1 sq2)≡ORTHO(b sq1 sq2) ↑↑↑↑;↑,L1;
44 ORTHO(a,sq1,sq2)≡ORTHO(b,sq1,sq2)  (1)

******⊃I L1⊃↑;
45 (SUBOARD(a,b)∧(¬(sq1=sq2)∧((Column sq1=Column sq2∧∀sq3.((BETWEEN(Row sq1,
Row sq3,Row sq2)∧Column sq3=Column sq1)⊃¬(Valueon(a,sq3)=UD)))∨(Row sq1=Row
sq2∧∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3=Row sq1)⊃¬(
Valueon(a,sq3)=UD))))))⊃(ORTHO(a,sq1,sq2)≡ORTHO(b,sq1,sq2))

******label EquiOrthoThm;
******∀1↑a b sq1 sq2;
46 ∀a b sq1 sq2.((SUBOARD(a,b)∧(¬(sq1=sq2)∧((Column sq1=Column sq2∧∀sq3.((
BETWEEN(Row sq1,Row sq3,Row sq2)∧Column sq3=Column sq1)⊃¬(Valueon(a,sq3)=UD)
))∨(Row sq1=Row sq2∧∀sq3.((BETWEEN(Column sq1,Column sq3,Column sq2)∧Row sq3
=Row sq1)⊃¬(Valueon(a,sq3)=UD))))))⊃(ORTHO(a,sq1,sq2)≡ORTHO(b,sq1,sq2)))
```

Section 3.4.2 Cornered Checking Pieces

This is a theorem about checks. It states that if a piece is checking the opposing king on a board, and, if on each of the squares that the piece can move to on that board, the piece still checks the king, then the original check was a discovered check. This situation is illustrated in *figure 28*, where the marked white queen is a cornered checking piece. This check must have been produced by the white bishop moving out from between the black king and the white queen. The theorem excludes certain exceptional conditions, such as pawn promotions, castles, *en passant* captures, and checks by pawns. These restrictions are necessary for these non-reversible moves. If this sounds like a complicated theorem, please be patient; it is the most intricate "general chess theorem" we prove.



The white queen is cornered

figure 28

We start with the assumption of some of the conditions for the theorem. We presume to have a position, q, whose immediate predecessor was r. The transition from r to q was not accomplished by a castle or en passant. q has a board, b, and, on some square of this board, sq, is a white value, vw. On some other square, sqx, is the black value KB (king black), and a piece of vw can move on b from sq to sqx. vw is not a pawn; sq is not in white's last rank. These last two conditions prevent the move from being either a pawn promotion, or a pawn's move.

We label this assumption *L1*.

```
*****label L1;
*****assume SUCCESSOR(r,q)∧(¬EN_PASSANT(r,q)∧(¬CASTLING(r,q)∧(¬WLASTRANK sq∧
*(BOARD(q,b)∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(MOVETO(b,vw,sq,sqx)∧¬
*VALUEP vw))))))));
1 SUCCESSOR(r,q)∧(¬EN_PASSANT(r,q)∧(¬CASTLING(r,q)∧(¬WLASTRANK sq∧(BOARD(q,b
)∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(MOVETO(b,vw,sq,sqx)∧¬VALUEP vw))))))
)  (1)
```

Since there is a white value on sq, it must belong to a chesspiece. Let us call that piece x.

```
*****VE PiecevaluesAreChesspieces q,b,sq;
2 (BOARD(q,b)∧PIECEVALUES Valueon(b,sq))⊃CHESSPIECES Pos(q,sq)

*****simplify PIECEVALUES vw;
3 PIECEVALUES vw

*****tauteq Pos(q,sq)=Pos(q,sq) ;
4 Pos(q,sq)=Pos(q,sq)

*****3I ↑ Pos(q,sq)←x occ 2;
5 ∃x.Pos(q,sq)=x

*****tauteq ∃x.Pos(q,sq)=x L1,2:3,5;
6 ∃x.Pos(q,sq)=x  (1)

*****label CALL_X;
```

```
*****∃E ↑ x;
7 Pos(q,sq)=x  (7)
```

We have presented sufficient conditions to prove the black king in check. We establish this fact, with the help of the lemma *AlternateBlack*[53] *AlternateBlack* also incorporates the knowledge that when black's king is in check, it must be black's turn to move.

```
*****taut Valueon(b,sqx)=KB∧(Valueon(b,sq)=vw∧MOVETO(b,vw,sq,sqx)) L1;
8 Valueon(b,sqx)=KB∧(Valueon(b,sq)=vw∧MOVETO(b,vw,sq,sqx))  (1)

*****∃I ↑ sqx sq vw;
9 ∃vw sq sqx.(Valueon(b,sqx)=KB∧(Valueon(b,sq)=vw∧MOVETO(b,vw,sq,sqx)))  (1)

*****∀E CHECKERS2 b;
10 BLACKINCHECK b=∃vw sq1 sq2.(Valueon(b,sq2)=KB∧(Valueon(b,sq1)=vw∧MOVETO(b
,vw,sq1,sq2)))

*****∀E AlternateBlack q,b;
11 (BOARD(q,b)∧BLACKINCHECK b)⊃(POSITIONINCHECK(q,BLACK)∧¬WHITETURN q)

*****label L2;
*****taut POSITIONINCHECK(q,BLACK)∧¬WHITETURN q L1;↑;
12 POSITIONINCHECK(q,BLACK)∧¬WHITETURN q  (1)
```

Also, if black is checked, then the color of position q must be black.

```
*****∀E POS_COLORS q,BLACK;
13 Color q=BLACK=(WHT BLACK=WHITETURN q)
```

The various type simplifications needed in the rest of the proof.

```
*****simplify ∀vw vb.¬(Valuecolor vw=Valuecolor vb);
14 ∀vw vb.¬(Valuecolor vw=Valuecolor vb)

*****∀E ↑ vw,KB;
15 ¬(Valuecolor vw=Valuecolor KB)

*****simplify ¬WHT BLACK∧VALUEK KB;
16 ¬WHT BLACK∧VALUEK KB
```

The proof will also employ parts of the definition of successor, and various facts about the colors of pieces.

```
*****label L3;
*****∀E MCONSEQA r,q;
17 SUCCESSOR(r,q)⊃((¬WHITETURN r=WHITETURN q)∧(Prevpos q=r∧(¬POSITIONINCHECK
(q,Color r)∧((WHITEPIECE Mover Move q=WHITETURN r)∧(Pos(r,From Move q)=Mover
 Move q∧(Pos(q,To Move q)=Mover Move q∧(Pos(q,From Move q)=EMPTY∧((CAPTURE
Move q⊃Pos(r,To Move q)=Taken Move q)∧(CASTLING(r,q)∨(EN_PASSANT(r,q)∨
SIMPLELEGALMOVE(r,q)))))))))))

*****taut Prevpos q=r L1,L3;
18 Prevpos q=r  (1)
```

---

53. Proven in section A.7.5.1.

By the theorem *__CheckTypes__*, there are four ways a check can occur. The piece that is making the check can have moved into the check, the check could have occurred on a discovery from an *en passant* capture, the rook of a castle could have moved and checked, or the check could have resulted from a piece moving out from between the king and the checking piece, a discovered check.

```
*****VE _CheckTypes_ q,b,sq,sqx,x,vw,KB;
19 (POSITIONINCHECK(q,Color q)∧(BOARD(q,b)∧(MOVETO(b,vw,sq,sqx)∧(Pos(q,sq)=x
∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(VALUEK KB∧¬(Valuecolor vw=Valuecolor
KB)))))))⊃(Mover Move q=x∨((EN_PASSANT(Prevpos q,q)∧(SQUARE_BETWEEN(sq,From
 Move q,sqx)∨SQUARE_BETWEEN(sq,Takenon Move q,sqx)))∨((CASTLING(Prevpos q,q)
∧Alsomover Move q=vw)∨(ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))))
)

*****substr ↑↑ in ↑;
20 (POSITIONINCHECK(q,Color q)∧(BOARD(q,b)∧(MOVETO(b,vw,sq,sqx)∧(Pos(q,sq)=x
∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(VALUEK KB∧¬(Valuecolor vw=Valuecolor
KB)))))))⊃(Mover Move q=x∨((EN_PASSANT(r,q)∧(SQUARE_BETWEEN(sq,From Move q,
sqx)∨SQUARE_BETWEEN(sq,Takenon Move q,sqx)))∨((CASTLING(r,q)∧Alsomover Move
q=vw)∨(ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))))  (1)
```

By our assumption *L1*, we can eliminate the special move (capture *en passant*, castle) possibilities.

```
*****label L4;
*****tauteq Mover Move q=x∨(ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,
*sqx)) L1,CALL_X,L2:13,15:16,↑;
21 Mover Move q=x∨(ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))  (1 7
)
```

Let use assume that the move was not a discovered check, but rather, that the checking piece, x, made the last move, into the checking position. We call this assumption *umption*.

```
*****label umption;
*****assume Mover Move q=x;
22 Mover Move q=x  (22)
```

If the last move was not a pawn promotion, then x has the same value in q as it had in r.

```
*****label sume;
*****assume Vsql.(MOVETO(Tboard q,vw,sq,sql)⊃(¬(Valueon(Tboard q,sql)=MT)∨
*MOVETO(Tboard q,vw,sqx,sql)));
23 Vsql.(MOVETO(Tboard q,vw,sq,sql)⊃(¬(Valueon(Tboard q,sql)=MT)∨MOVETO(
Tboard q,vw,sqx,sql)))  (23)
```

We assume that every square that this piece could have moved from, either is not empty, or also checks the black king.

```
*****VE sume From Move q;
24 MOVETO(Tboard q,vw,sq,From Move q)⊃(¬(Valueon(Tboard q,From Move q)=MT)∨
MOVETO(Tboard q,vw,sqx,From Move q))  (23)

*****VE _MoveBack_ r,q,vw,To Move q,From Move q;
25 (SUCCESSOR(r,q)∧(ORDINARY Move q∧¬VALUEP vw))⊃(MOVETO(Tboard q,vw,To Move
 q,From Move q)=MOVETO(Tboard r,vw,From Move q,To Move q))
```

Now, most piece moves are commutative. We need only show that this (non-pawn valued) piece did not just promote, and this value is also the value of the piece on the square sq in the boards of r and q.

```
*****VE MCONSEQK r,q;
26 SIMPLELEGALMOVE(r,q)≡(¬(From Move q=To Move q)∧(MOVETO(Tboard r,Valueon(
Tboard r,From Move q),From Move q,To Move q)∧((SIMPLE Move q∧Valueon(Tboard
r,To Move q)=MT)∨(CAPTURE Move q∧(PIECEVALUES Valueon(Tboard r,To Move q)∧¬(
Valuecolor Valueon(Tboard r,To Move q)=Color r))))))

*****VE MconseqmX r,q;
27 SUCCESSOR(r,q)⊃((CASTLE Move q≡CASTLING(r,q))∧((ENPASSANT Move q≡EN_
PASSANT(r,q))∧(ORDINARY Move q≡SIMPLELEGALMOVE(r,q))))

*****label L5;
*****VE UnpromotedFrom r,q,b,x,sq;
28 (SUCCESSOR(r,q)∧(¬WLASTRANK sq∧(BOARD(q,b)∧(Valueon(b,sq)=vw∧(Pos(q,sq)=x
∧Mover Move q=x)))))⊃Valueon(Tboard r,From Move q)=vw

*****VE MOVETYPES1 Move q;
29 ENPASSANT Move q∨(CASTLE Move q∨ORDINARY Move q)

*****VE Unique q,To Move q,sq,x;
30 Pos(q,To Move q)=x⊃(Pos(q,sq)=x≡To Move q=sq)

*****tauteq ¬(Valueon(Tboard q,From Move q)=MT)∨MOVETO(Tboard q,vw,sqx,From
*Move q) L1,CALL_X,L3,umption,sume+1:↑;
31 ¬(Valueon(Tboard q,From Move q)=MT)∨MOVETO(Tboard q,vw,sqx,From Move q)
(1 7 22 23)
```

Now, the source square of this move is obviously empty. Hence, the MT squares of our assume sume can be eliminated. The piece must be able to make the indicated move.

```
*****VE ValueTransposition C q,From Move q;
32 Valueon(Tboard q,From Move q)=Val(q,Pos(q,From Move q))

*****VE EmptyFrom q,Pos(q,From Move q),From Move q;
33 CHESSPIECES Pos(q,From Move q)⊃(Pos(q,From Move q)≡Pos(q,From Move q)⊃¬(
From Move q=From Move q))

*****VE EmptyIsMT q,Pos(q,From Move q);
34 Pos(q,From Move q)=EMPTY≡Val(q,Pos(q,From Move q))=MT

*****VE NotChesspieceEmpty_ Pos(q,From Move q);
35 ¬CHESSPIECES Pos(q,From Move q)≡Pos(q,From Move q)=EMPTY

*****tauteq MOVETO(Tboard q,vw,sqx,From Move q) ↑↑↑↑↑:↑;
36 MOVETO(Tboard q,vw,sqx,From Move q)  (1 7 22 23)
```

The movement commutivity rules also hold for this MOVETO. We need to show that the values of the pieces haven't changed by this last move. As the move was not a pawn promotion, this follows.

```
*****VE _MoveBack_ r,q,vw,sqx,From Move q;
37 (SUCCESSOR(r,q)∧(ORDINARY Move q∧¬VALUEP vw))⊃(MOVETO(Tboard q,vw,sqx,
From Move q)≡MOVETO(Tboard r,vw,From Move q,sqx))
```

```
*****tauteq MOVETO(Tboard r,vw,From Move q,sqx) L1,L5-1,L5+1,↑↑:↑;
38 MOVETO(Tboard r,vw,From Move q,sqx)  (1 7 22 23)

*****VE OtherSideStays r,q,sqx,BK;
39 (SUCCESSOR(r,q)∧((WHITEPIECE BK=WHITETURN q)∧Pos(q,sqx)=BK))⊃Pos(r,sqx)=
BK

*****VE KingValueThm r,Tboard r,sqx;
40 (BOARD(r,Tboard r)∧¬(Valueon(Tboard r,sqx)=UD))⊃((Pos(r,sqx)=WK=Valueon(
Tboard r,sqx)=KW)∧(Pos(r,sqx)=BK=Valueon(Tboard r,sqx)=KB))

*****label L6;
*****VE BoardTboard r;
41 BOARD(r,Tboard r)

*****VE SUB_BOARDS2 Tboard r,sqx;
42 ¬(Valueon(Tboard r,sqx)=UD)

*****simplify ¬WHITEPIECE BK∧¬(KB=UD);
43 ¬WHITEPIECE BK∧¬(KB=UD)

*****VE KingValueThm q,b,sqx;
44 (BOARD(q,b)∧¬(Valueon(b,sqx)=UD))⊃((Pos(q,sqx)=WK=Valueon(b,sqx)=KW)∧(Pos
(q,sqx)=BK=Valueon(b,sqx)=KB))
```

Hence, black must also have been in check in the previous position.

```
*****tauteq Valueon(Tboard r,sqx)=KB∧(Valueon(Tboard r,From Move q)=vw∧
*MOVETO(Tboard r,vw,From Move q,sqx)) L1,CALL_X,L2,umption,L5,↑↑↑↑↑↑↑:↑;
45 Valueon(Tboard r,sqx)=KB∧(Valueon(Tboard r,From Move q)=vw∧MOVETO(Tboard
r,vw,From Move q,sqx))  (1 7 22 23)

*****3( ↑ sqx sq2 From Move q sq1 vw ;
46 ∃vw sq1 sq2.(Valueon(Tboard r,sq2)=KB∧(Valueon(Tboard r,sq1)=vw∧MOVETO(
Tboard r,vw,sq1,sq2)))  (1 7 22 23)

*****VE CHECKERS2 Tboard r;
47 BLACKINCHECK Tboard r=∃vw sq1 sq2.(Valueon(Tboard r,sq2)=KB∧(Valueon(
Tboard r,sq1)=vw∧MOVETO(Tboard r,vw,sq1,sq2)))
```

This is clearly impossible.

```
*****VE AlternateBlack r,Tboard r;
48 (BOARD(r,Tboard r)∧BLACKINCHECK Tboard r)⊃(POSITIONINCHECK(r,BLACK)∧¬
WHITETURN r)

*****tauteq FALSE L1,L2,L3,L6,↑↑↑:↑;
49 FALSE  (1 7 22 23)
```

Therefore, we must negate our assumption that this cornered piece made the last move.

```
*****¬I ↑,Mover Move q=x;
50 ¬(Mover Move q=x)  (1 7 23)
```

We arrange this conclusion in a more useful form.

```
*****taut (ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))∧¬(Mover Move
*q=x) L4,↑;
51 (ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))∧¬(Mover Move q=x)  (
1 7 23)

*****subst CALL_X in ↑;
52 (ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))∧¬(Mover Move q=Pos(q
,sq))  (1 23)
```

And, after removing the dependencies, we generalize.

```
*****⊃I sume⊃↑;
53 ∀sq1.(MOVETO(Tboard q,vw,sq,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(
Tboard q,vw,sqx,sq1)))⊃((ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))
∧¬(Mover Move q=Pos(q,sq)))  (1)

*****⊃I L1⊃↑;
54 (SUCCESSOR(r,q)∧(¬EN_PASSANT(r,q)∧(¬CASTLING(r,q)∧(¬WLASTRANK sq∧(BOARD(q
,b)∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(MOVETO(b,vw,sq,sqx)∧¬VALUEP vw))))
))))⊃(∀sq1.(MOVETO(Tboard q,vw,sq,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(
Tboard q,vw,sqx,sq1)))⊃((ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))
∧¬(Mover Move q=Pos(q,sq))))
```

We call this theorem *WhiteCornered*. *BlackCornered* is the same theorem for black checking white. We forego the repetition required for its proof.

```
*****label WhiteCornered;
*****∀I ↑ r q b vw sq sqx;
55 ∀r q b vw sq sqx.((SUCCESSOR(r,q)∧(¬EN_PASSANT(r,q)∧(¬CASTLING(r,q)∧(¬
WLASTRANK sq∧(BOARD(q,b)∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧(MOVETO(b,vw,
sq,sqx)∧¬VALUEP vw)))))))⊃(∀sq1.(MOVETO(Tboard q,vw,sq,sq1)⊃(¬(Valueon(
Tboard q,sq1)=MT)∨MOVETO(Tboard q,vw,sqx,sq1)))⊃((ORDINARY Move q∧SQUARE_
BETWEEN(sq,From Move q,sqx))∧¬(Mover Move q=Pos(q,sq)))))
```

The corresponding result for checking the white king is:

```
define BlackCornered:
∀r q b vw sq sqx.((SUCCESSOR(r,q)∧(¬EN_PASSANT(r,q)∧(¬CASTLING(r,q)
∧(¬WLASTRANK(sq)∧((BOARD(q,b)∧(Valueon(b,sq)=vw∧(Valueon(b,sqx)=KB∧
MOVETO(b,vw,sq,sqx))))∧¬VALUEP vw)))))⊃(∀sq1.(MOVETO(Tboard q,vw,
sq,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,vw,sqx,sq1)))⊃
((ORDINARY Move q∧SQUARE_BETWEEN(sq,From Move q,sqx))∧¬(Mover Move
q=Pos(q,sq)))))));;
```

Section 3.4.3 No Black Pawns on the First Row

A final annotated chess lemma. We prove the theorem *NoBlackPawnsOn1Row*, which states that no piece whose value is PB (black pawn) is ever in on any square of the board's first row. This is of course true, as all the black pawns start on the second row, and, while they still have the value of pawn, never move backwards.

An elenctic proof. We assume that such a condition exists. In some position p, a chesspiece x is to have value PB. In p, x is on square sq1. The row of sq1 is 1.

```
*****label L1;
***** assume Val(p,x)=PB∧Pos(p,sq1)=x;
1 Val(p,x)=PB∧Pos(p,sq1)=x  (1)

*****assume Row(sq1)=1;
2 Row sq1=1  (2)
```

x must, of course, be a black piece.

```
*****label L2;
***** VE ColorChoices p,x;
3 (BVALUES Val(p,x)=BLACKPIECE x)∧(WVALUES Val(p,x)=WHITEPIECE x)

******simplify BVALUES PB∧VALUEP PB ;
4 BVALUES PB∧VALUEP PB

*****VE PieceChoices_ x;
5 (WHITEPIECE x=Piececolor x=WHITE)∧(BLACKPIECE x=Piececolor x=BLACK)

*****label L3;
***** tauteq Piececolor x=BLACK L1,L2;↑;
6 Piececolor x=BLACK  (1)
```

Every piece started on some square. Let us call the square that x was on in the initial position, sq2.

```
*****VE AllStart_ x;
7 ∃sq.Pos(P0,sq)=x

*****∃E ↑ sq2;
8 Pos(P0,sq2)=x  (8)
```

If x has pawn value, it must be a pawn. Since x is a blackpiece, it must be a black pawn.

```
*****VE PawnValuedPawnsThm p,x;
9 VALUEP Val(p,x)⊃PAWNS x

*****VE BlackpiecePawnsAre_ x;
10 (BLACKPIECE x∧PAWNS x)=BPAWNS x
```

Simplification tells us that all black pawns start in the second row.

```
*****label L4;
***** VE BlackPawnsOn2Start_ sq2;
11 BPAWNS Pos(P0,sq2)=Row sq2=2

*****tauteq Row(sq2)=2 L1,L2;L2+2,L3+2;L4;
12 Row sq2=2  (1 8)
```

Each of sq1 and sq2 is the composite of its row and column.

```
*****VE SQUARED1 sq2;
13 sq2=Makesquare(Row sq2,Column sq2)

*****VE SQUARED1 sq1;
```

```
14 sq1=Makesquare(Row sq1,Column sq1)

*****substr ↑↑↑ in ↑↑;
15 sq2=Makesquare(2,Column sq2)  (1 8)

*****substr L1+1 in ↑↑;
16 sq1=Makesquare(1,Column sq1)  (2)
```

By the theorem *PawnStructure_1*, every path that a pawn takes must satisfy the predicate MAY_PAWN_CAPTURES. We substitute the Makesquare value for sq1 and sq2 in this WFF.

```
*****VE _PawnStructure_1 P0,p,x,sq1,sq2;
17 (Pos(p,sq1)=x∧(Pos(P0,sq2)=x∧(PREDEGAME(P0,p)∧VALUEP Val(Prevpos p,x))))⊃
MAY_PAWN_CAPTURES(sq2,sq1,Piececolor x)

*****substr ↑↑↑ in ↑ occ 2;
18 (Pos(p,sq1)=x∧(Pos(P0,sq2)=x∧(PREDEGAME(P0,p)∧VALUEP Val(Prevpos p,x))))⊃
MAY_PAWN_CAPTURES(Makesquare(2,Column sq2),sq1,Piececolor x)  (1 8)

*****substr ↑↑↑ in ↑ occ 2;
19 (Pos(p,sq1)=x∧(Pos(P0,sq2)=x∧(PREDEGAME(P0,p)∧VALUEP Val(Prevpos p,x))))⊃
MAY_PAWN_CAPTURES(Makesquare(2,Column sq2),Makesquare(1,Column sq1),
Piececolor x)  (1 2 8)

*****label L5;
***** substr L3 in ↑;
20 (Pos(p,sq1)=x∧(Pos(P0,sq2)=x∧(PREDEGAME(P0,p)∧VALUEP Val(Prevpos p,x))))⊃
MAY_PAWN_CAPTURES(Makesquare(2,Column sq2),Makesquare(1,Column sq1),BLACK)
(1 2 8)
```

We know P0 to have occurred in the game of p.

```
*****VE POSITION_RULES p;
21 SUCCESSOR(Prevpos p,p)∧PREDEGAME(P0,p)
```

And that if a piece has pawn value, it has always had pawn value.

```
*****VE PreviousPawnValue Prevpos p,p,x;
22 Prevpos p=Prevpos p⊃(VALUEP Val(p,x)⊃VALUEP Val(Prevpos p,x))
```

Simplification reveals that there are no two squares satisfying the MAY_PAWN_CAPTUURES predicate for black, such that the transition goes from the second row to the first. Thus, we have a contradiction.

```
*****VE NotMPC_Black2to1_ Column(sq2),Column(sq1);
23 ¬MAY_PAWN_CAPTURES(Makesquare(2,Column sq2),Makesquare(1,Column sq1),
BLACK)

*****tauteq FALSE L1,L2+1,L3+2,L5:↑;
24 FALSE  (1 2)
```

Our original assumption must be wrong. No piece with value black pawn can be on a square whose row is one in any GAMEPOSITION.

```
*****¬I ↑,Row(sq1)=1;
25 ¬(Row sq1=1)  (1)

*****⊃I L1⊃↑;
26 (Val(p,x)=PB∧Pos(p,sq1)=x)⊃¬(Row sq1=1)

*****label L6;
***** VI ↑ p sq1;
27 ∀p sq1.((Val(p,x)=PB∧Pos(p,sq1)=x)⊃¬(Row sq1=1))
```

However, we wish to prove our theorem for all POSITIONS, not just GAMEPOSITIONs. Hence, we must establish it for the initial position. This is trivial, as all black pawns are on the second row at the beginning of the game, not the first. We first establish that all things with value of pawn black in the initial position are the black pawns; we then instantiate our just concluded lemma to any position, r, show by simplification that that if r is P0,the theorem is still true. As all positions are either GAMEPOSITIONs or P0, we have our theorem.

```
*****simplify ∀x.(Val(P0,x)=PB=BPAWNS x);
28 ∀x.(Val(P0,x)=PB=BPAWNS x)

*****∀E ↑ x;
29 Val(P0,x)=PB=BPAWNS x

*****simplify ¬(2=1);
30 ¬(2=1)

*****tauteq (Val(P0,x)=PB∧Pos(P0,sq2)=x)⊃¬(Row(sq2)=1) L4,↑↑;↑;
31 (Val(P0,x)=PB∧Pos(P0,sq2)=x)⊃¬(Row sq2=1)

*****assume r=P0;
32 r=P0  (32)

*****subst ↑ in ↑↑;
33 (Val(r,x)=PB∧Pos(r,sq2)=x)⊃¬(Row sq2=1)  (32)

*****⊃I ↑↑⊃↑;
34 r=P0⊃((Val(r,x)=PB∧Pos(r,sq2)=x)⊃¬(Row sq2=1))

*****∀E L6 r,sq2;
35 GAMEPOSITION r⊃((Val(r,x)=PB∧Pos(r,sq2)=x)⊃¬(Row sq2=1))

*****∀E POSITION_TYPES r;
36 ¬(r=P0=GAMEPOSITION r)

*****taut (Val(r,x)=PB∧Pos(r,sq2)=x)⊃¬(Row(sq2)=1) ↑↑↑;↑;
37 (Val(r,x)=PB∧Pos(r,sq2)=x)⊃¬(Row sq2=1)

*****label NoBlackPawnsOn1Row;
***** VI ↑ r x sq2←sq;
38 ∀r x sq.((Val(r,x)=PB∧Pos(r,sq)=x)⊃¬(Row sq=1))
```

Chapter 4 A FOL Solution to the Chess Puzzle

> ***Systems of natural deduction . . . constitute a form for the development of logic that is natural in many respects. In the first place, there is a similarity between natural deduction and intuitive, informal reasoning. The inference rules of the systems of natural deduction correspond closely to procedures common in intuitive reasoning, and when informal proofs -- such as are encountered in mathematics for example -- are formalized within these systems, the main structure of the informal proofs can often be preserved. This in itself gives the systems of natural deduction an interest as an explication of the informal concept of logical deduction.***
>
> ***Dag Prawitz***[54]

This chapter details our proof, in FOL, of the solution to the chess puzzle presented in section 1.6. This proof follows closely with the solution presented in that section.

Section 4.1 Declarations for this Proof

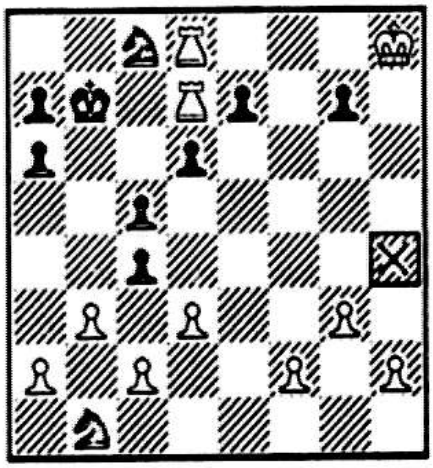
First order logic is somewhat distinguished by the proliferation of constants. If one needs a new entity, one creates a new constant; if a particular formula is a frequent referent, one defines a new predicate to abbreviate that formula. This particular proof shall not spawn any new predicates for the chess world. However, perhaps obviously, we shall need names for the individuals mentioned in the problem and solution. More particularly, we define INDCONSTs for some of the more important boards of section 1.6.

Most obviously, we need a constant to represent the puzzle board, the board illustrated in *figure 29*. Let us call this individual GIVEN.

```
declare INDCONST GIVEN ∈ BOARDS;
```

---

54. [Prawitz65], page 7.



The board GIVEN

figure 29

The attachment to GIVEN is therefore:

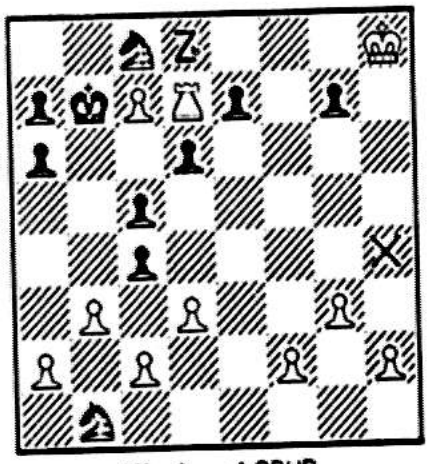
```
attach GIVEN ↔[CHESS] ((MT MT NB RW MT MT MT KW)  (PB KB MT RW PB MT PB MT)
                       (PB MT MT PB MT MT MT MT)  (MT MT PB MT MT MT MT MT)
                       (MT MT PB MT MT MT MT UD)  (MT PW MT PW MT MT PW MT)
                       (PW MT PW MT MT PW MT PW)  (MT NB MT MT MT MT MT MT));
```

Our proof also dealt at length with the position of the board just prior to the last move. We concluded that a white pawn on WQB7 had captured some black piece on BQ1. We shall need to refer to several of the possible identities of that piece. In the base situation, that piece is undefined, and we get the board QBUD.[55] When we wish to consider that piece a rook or a queen, we will use the boards QBR and QBQ. Recalling the definitions of section 2.1.5, we see that QBUD is a sub-board of both QBR and QBQ. QBUD is illustrated in *figure 30*.

```
declare INDCONST QBUD QBR QBQ ∈ BOARDS;
```

---

55. In this proof, we will refer to the position presented in the problem as px, its previous position, qx. Thus, the name QBUD signifies that this in position Qx, this Board is UnDefined on the interesting square (BQ1). Similarly, QBR has a black rook on that square; QBQ, a black queen.



The board QBUD

figure 30

```
attach QBR ↔ [CHESS] ((MT MT NB RB MT MT MT KW) (PB KB PW RW PB MT PB MT)
                      (PB MT MT PB MT MT MT MT) (MT MT PB MT MT MT MT MT)
                      (MT MT PB MT MT MT MT UD) (MT PW MT PW MT MT PW MT)
                      (PW MT PW MT MT PW MT PW) (MT NB MT MT MT MT MT MT));
attach QBQ ↔ [CHESS] ((MT MT NB QB MT MT MT KW) (PB KB PW RW PB MT PB MT)
                      (PB MT MT PB MT MT MT MT) (MT MT PB MT MT MT MT MT)
                      (MT MT PB MT MT MT MT UD) (MT PW MT PW MT MT PW MT)
                      (PW MT PW MT MT PW MT PW) (MT NB MT MT MT MT MT MT));
attach QBUD↔ [CHESS] ((MT MT NB UD MT MT MT KW) (PB KB PW RW PB MT PB MT)
                      (PB MT MT PB MT MT MT MT) (MT MT PB MT MT MT MT MT)
                      (MT MT PB MT MT MT MT UD) (MT PW MT PW MT MT PW MT)
                      (PW MT PW MT MT PW MT PW) (MT NB MT MT MT MT MT MT));
```

Section 4.2 The Proof

Declarations completed, we plunge forward into our proof. One of the major propositions of this paper is the existence of a correspondence between the *human* solution to our chess puzzle (presented in 1.6.2), and our FOL encoding of that proof. In support of this hypothesis, this chapter is organized like section 1.6.2; we number the description of our FOL proof to illustrate the relationship.

Section 4.2.1 Black is in Check

We seek to prove that, if the given board (GIVEN) is the board of some legal position, and there is a chesspiece on the square WKR4, then that piece must be the white queen's bishop (WQB). Expressed as a FOL WFF, this becomes:[56]

$$\forall p.\ ((\text{BOARD}(p\ \text{GIVEN}) \wedge \text{CHESSPIECES Pos}(p\ \text{WKR4})) \supset \text{Pos}(p\ \text{WKR4}) = \text{WQB})$$

It is therefore reasonable to begin our proof with the assumption of the antecedent of this WFF. Rather than p, we select the distinctive parameter px to symbolize this original position. For future reference, we label this line CALL_PX

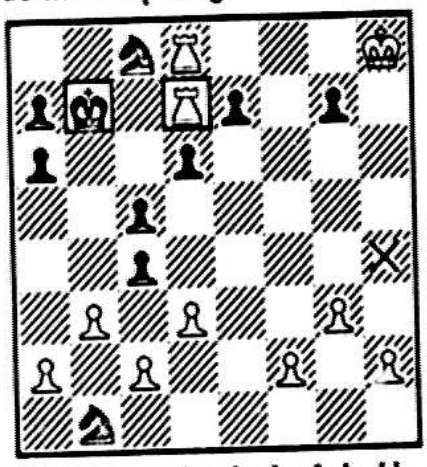
```
*****label CALL_PX;
*****assume BOARD(px,GIVEN)∧CHESSPIECES Pos(px,WKR4);
```

---

56. Here we have begun by presuming that the given board is a position of a GAMEPOSITION, not just any POSITIONS. This is, of course, trivially established. The only non GAMEPOSITION POSITIONS is P0, the initial (game starting) position. A quick consultation to the simplification mechanism will show they differ on many squares. Hence, we take the liberty of using px, rather than some POSITIONS variable.

```
1 BOARD(px,GIVEN)∧CHESSPIECES Pos(px,WKR4)  (1)
```

1. We see that, on the given board, the black king is in check (*figure 31*). We obtain this through a single invocation of the simplification mechanism. Notice how we have transformed this *observation* into a simple *computation*. We will use this computational ability in this proof whenever possible; more particularly, when we have the ground instances (constants) to compute about, and have the appropriate functions to do the computing.[57]



Simplification sees the check of the black king.

figure 31

```
*****label BINCHECK;
*****simplify BLACKINCHECK GIVEN;
2 BLACKINCHECK GIVEN
```

1.1. One of the more trivial chess lemmas, ***AlternateBlack***, informs us that any position which has a BLACKINCHECK board must have black on move. Additionally, the lemma fills in the POSITIONINCHECK predicate for us. Let us call this line BLACK_GOES.

```
*****label BLACK GOES;
*****VE AlternateBlack px,GIVEN;
3 (BOARD(px,GIVEN)∧BLACKINCHECK GIVEN)⊃(POSITIONINCHECK(px,BLACK)∧¬WHITETURN
  px)
```

1.2. If this position is black's turn, then white must have made the previous move. We want a name for this position, too. We will call it qx. Implicit in using a name from the sort of LEGALPOSITION, rather from POSITIONS is the obligation to show that the stated position is not the initial position. It is obvious to us that the board GIVEN was not achieved one move from the start of the game. But to convince our proof checker, we invoke the lemma ***PREVLEGAL***, which demands

---

57. We have tried to have all of the appropriate functions defined in our axiomatization (chapter 2). Occasionally, computing something with the chess eye, and thereby considering each case of a quantified WFF, would be too time consuming. In those instances, we may attempt the proof through the usual deductive means.

the display of a black piece not on its original square. What piece to use? Kings are the easiest commodity; from the lemma *KingValueThm* we know that any king valued piece must be the king, and we can see (can simplify) the black king on BQN2.

```
*****VE KingValueThm px,GIVEN,BQN2;
4 (BOARD(px,GIVEN)∧¬(Valueon(GIVEN,BQN2)=UD))⊃((Pos(px,BQN2)=WK=Valueon(
GIVEN,BQN2)=KW)∧(Pos(px,BQN2)=BK=Valueon(GIVEN,BQN2)=KB))

*****simplify ↑;
5 BOARD(px,GIVEN)⊃(¬(Pos(px,BQN2)=WK)∧Pos(px,BQN2)=BK)

*****VE PrevGameposition px,BQN2,BK;
6 (((WHITEPIECE BK=WHITETURN px)∧Pos(px,BQN2)=BK)∧¬(Pos(P0,BQN2)=BK))⊃∃q.
Prevpos px=q

*****simplify ↑;
7 (¬WHITETURN px∧Pos(px,BQN2)=BK)⊃∃q.Prevpos px=q

*****taut ∃q.Prevpos px=q CALL_PX:BLACK_GOES,↑↑↑,↑;
8 ∃q.Prevpos px=q  (1)

*****label CALL_QX;
*****∃E ↑ qx;
9 Prevpos px=qx  (9)
```

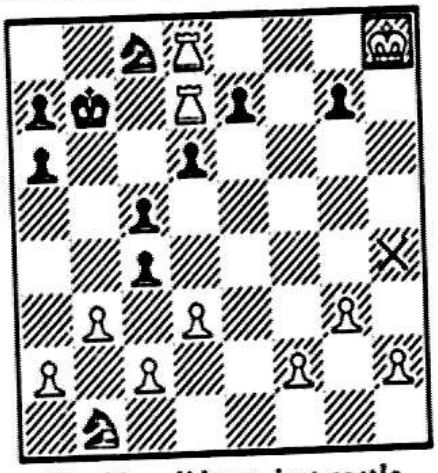
It is also useful to have around (for the conditional parts of various theorems) facts about the ancestry and relationships of px and qx. We create and label these auxiliaries.

```
*****label PXIS;
*****VE POSITION_RULES px;
10 SUCCESSOR(Prevpos px,px)∧PREDEGAME(P0,px)

*****label QXIS;
*****VE POSITION_RULES qx;
11 SUCCESSOR(Prevpos qx,qx)∧PREDEGAME(P0,qx)

*****label PXSUC;
*****tauteq SUCCESSOR(qx,px) CALL_QX:PXIS;
12 SUCCESSOR(qx,px)  (9)
```

Section 4.2.2 White's Last Move

2. Our attention turns to discovering white's last move, the one that put black into check. We consider each of the possible checking maneuvers (castling rook makes the check, a pawn captured *en passant* leaves a discovered check, the checking piece made the last move, and the piece that move last discovered check) and discard the first three.

2.1. We wish to prove that white did not castle to reach this position. This is easy. We observe that on GIVEN, a board of px, white's king is on BKR1. This is not, of course, a square a castle can leave the king upon, (as the lemma *WhiteCastleThm* informs us) (*figure 32*).



The king did not just castle.

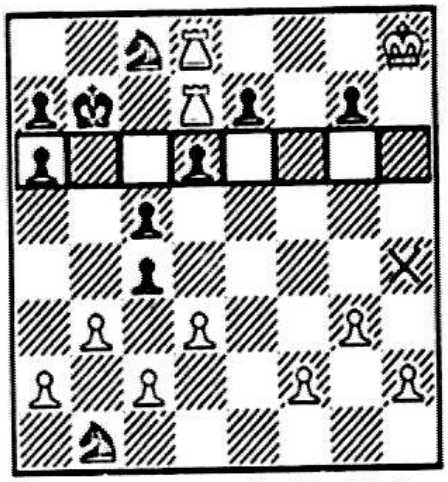
figure 32

```
*****VE KingValueThm px,GIVEN,BKR1;
13 (BOARD(px,GIVEN)∧¬(Valueon(GIVEN,BKR1)=UD))⊃((Pos(px,BKR1)=WK=Valueon(
GIVEN,BKR1)=KW)∧(Pos(px,BKR1)=BK=Valueon(GIVEN,BKR1)=KB))

*****VE WhiteCastleThm qx,px,BKR1;
14 (SUCCESSOR(qx,px)∧(CASTLING(qx,px)∧¬WHITETURN px))⊃(Pos(px,BKR1)=WK⊃(BKR1
=WKN1∨BKR1=WQB1))

*****simplify ↑↑;
15 BOARD(px,GIVEN)⊃(Pos(px,BKR1)=WK∧¬(Pos(px,BKR1)=BK))

*****simplify ↑↑;
16 (SUCCESSOR(qx,px)∧(CASTLING(qx,px)∧¬WHITETURN px))⊃¬(Pos(px,BKR1)=WK)
```

We can conclude, tautologically, that white did not just castle.

```
*****label NOTPXCASTLE;
*****tauteq ¬CASTLING(qx,px) CALL_PX:BLACK_GOES,PXSUC,↑↑:↑;
17 ¬CASTLING(qx,px)  (1 9)
```

2.2. Similarly, if white has just captured *en passant*, then he would have a pawn on black's third row (from the theorem *WhiteEnPassantThm2*). Since GIVEN is a board of px, and inspection reveals neither an undefined piece on the third row, nor a white pawn, we can quickly dismiss *en passant* capture as a possibility (*figure 33*).



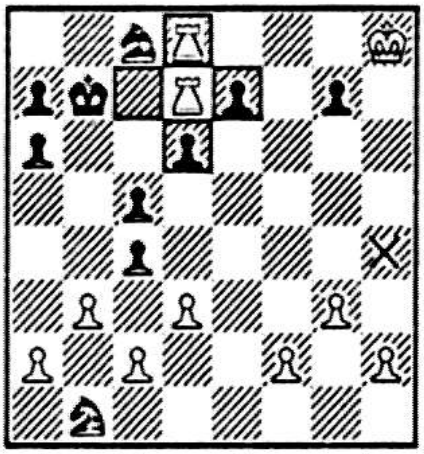
No white pawns on black's third row.

figure 33

```
*****VE WhiteEnPassantThm2 qx px GIVEN;
18 (SUCCESSOR(qx,px)∧(EN_PASSANT(qx,px)∧¬WHITETURN px))⊃(Vdcx.¬(Valueon(
GIVEN,Makesquare(3,dcx))=PW∨Valueon(GIVEN,Makesquare(3,dcx))=UD)⊃¬BOARD(px,
GIVEN))

*****simplify ↑;
19 (SUCCESSOR(qx,px)∧(EN_PASSANT(qx,px)∧¬WHITETURN px))⊃¬BOARD(px,GIVEN)

*****label NOTPXEP;
*****tauteq ¬EN_PASSANT(qx,px) CALL_PX↓BLACK_GOES,PXSUC,↑↓
20 ¬EN_PASSANT(qx,px)  (1 9)
```

Section 4.2.2.1 The Check Must Have Been Discovered

2.3. Proving that the piece that moved last generated the check is more difficult. Knowing chess, and with broad pattern recognition abilities, we can see that the checking white rook is blocked on ever side except the king's, and that only moves that started with the king in check could have lead him to that square (*figure 34*):



The white rook is cornered.

figure 34

We have a theorem, ***WhiteCornered***, with that effect: if a checking piece is trapped on all sides, then that piece did not make the last move, but, rather, the check was a discovered check. (This is only true with assumptions which eliminate the special moves.) The validity of this condition is obvious on the problem board (GIVEN). However, some deduction is needed to show that it still holds on the *total* board of px (which is not undefined on WKR4, the x-ed square.) We have shoved this deduction to the background; it is presented in the lemma ***BlockedGivenThm*** in section B.2. That derivation is a good example of both the problems accruing to different representations of the same object, and the difficulties involved in proving predicates true on *similar* objects.

We invoke our theorem about cornered checking pieces.

```
*****VE WhiteCornered qx px GIVEN RW BQ2 BQN2;
21 (SUCCESSOR(qx,px)∧(¬EN_PASSANT(qx,px)∧(¬CASTLING(qx,px)∧(¬WLASTRANK BQ2∧(
(BOARD(px,GIVEN)∧(Valueon(GIVEN,BQ2)=RW∧(Valueon(GIVEN,BQN2)=KB∧MOVETO(GIVEN
,RW,BQ2,BQN2))))∧¬VALUEP RW)))))⊃(∀sq1.(MOVETO(Tboard px,RW,BQ2,sq1)⊃(¬(
Valueon(Tboard px,sq1)=MT)∨MOVETO(Tboard px,RW,BQN2,sq1)))⊃((ORDINARY Move
px∧SQUARE_BETWEEN(BQ2,From Move px,BQN2))∧¬(Mover Move px=Pos(px,BQ2))))
```

Some of the antecedents of this WFF have been established earlier in this proof (such as the successor relationship between qx and px, and the non castling nature of the last move). Others we can see by observation. We need here observe that the checking piece is not on the last rank, it can capture the king, and pieces on that board are where we claim them to be.

```
*****simplify ¬WLASTRANK BQ2∧Valueon(GIVEN,BQ2)=RW∧Valueon(GIVEN,BQN2)=KB
*   ∧MOVETO(GIVEN,RW,BQ2,BQN2)∧¬VALUEP RW;
22 ¬WLASTRANK BQ2∧(Valueon(GIVEN,BQ2)=RW∧(Valueon(GIVEN,BQN2)=KB∧(MOVETO(
```

```
GIVEN,RW,BQ2,BQN2)∧¬VALUEP RW)))
```

The quantified part of the conditional is obtained through the use of our lemma.

```
*****VE BlockedGivenThm px;
23 BOARD(px,GIVEN)⊃∀sq1.(MOVETO(Tboard px,RW,BQ2,sq1)⊃(¬(Valueon(Tboard px,
sq1)=MT)∨MOVETO(Tboard px,RW,BQN2,sq1)))
```

2.4. Hence, the check was a discovered check; the piece that made the last move moved out from between the king and rook.

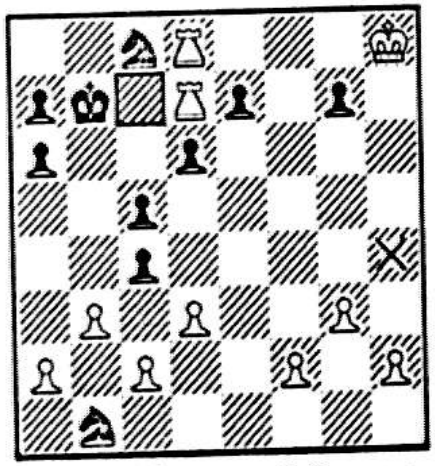
```
*****label ORDPX;
*****tauteq ↑↑↑:#2#2 ↑↑↑:↑,PXSUC,NOTPXEP,NOTPXCASTLE,CALL_PX;
24 (ORDINARY Move px∧SQUARE_BETWEEN(BQ2,From Move px,BQN2))∧¬(Mover Move px=
Pos(px,BQ2))  (1)
```

Section 4.2.3 Which Piece Discovered the Check

Section 4.2.3.1 Where the Last Move Originated

3. We seek the identity of the piece that moved last.

3.1. We observe that there is only one square between the rook and the king, BQB2. If the piece that moved last moved out from between them, if must have come from this square.



The FROM square of the move.

figure 35

```
*****simplify ∀sq.(SQUARE_BETWEEN(BQ2 sq BQN2)⊃sq=BQB2);
25 ∀sq.(SQUARE_BETWEEN(BQ2,sq,BQN2)⊃sq=BQB2)

*****VE ↑ From Move px;
26 SQUARE_BETWEEN(BQ2,From Move px,BQN2)⊃From Move px=BQB2
```

```
*****label FROMPX;
*****tauteq From Move px=BQB2 ↑↑↑,↑;
27 From Move px=BQB2  (1)
```

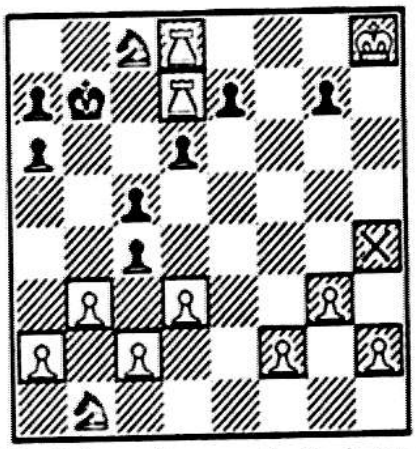
Section 4.2.3.2 The Last Move was a Pawn Promotion

3.2. Perhaps the last move was not special. We have already eliminated the possibility that the move was a capture en passant or a castle. Let us assume that the last move was not a pawn promotion.

We know several facts about all moves. In particular, moves are either castles, captures en passant, or satisfy the SIMPLELEGALMOVE predicate.

```
*****label MCONAPX;
*****VE MCONSEQA qx px;
28 SUCCESSOR(qx,px)⊃((¬WHITETURN qx=WHITETURN px)∧(Prevpos px=qx∧(¬
POSITIONINCHECK(px,Color qx)∧((WHITEPIECE Mover Move px=WHITETURN qx)∧(Pos(
qx,From Move px)=Mover Move px∧(Pos(px,To Move px)=Mover Move px∧(Pos(px,
From Move px)=EMPTY∧((CAPTURE Move px⊃Pos(qx,To Move px)=Taken Move px)∧(
CASTLING(qx,px)∨(EN_PASSANT(qx,px)∨SIMPLELEGALMOVE(qx,px))))))))))
```

We have a lemma ***MovedValues*** (section A.72.1) applicable to this situation. It states that for all ordinary, non-pawn promoting moves, the moving piece, with its current value, could MOVETO, on the total board of the previous position, from the From square of that move, to the To square. Additionally, when the ensuing position has black on move, then a white piece occupies the To square of that move (and similarly for white). We consider each of the white and undefined pieces on GIVEN in turn (*figure 36*).



Which of these pieces made the last move?

figure 36

```
*****VE MovedValues qx px GIVEN BQB2 To Move px;
29 ((SUCCESSOR(qx,px)∧(¬EN_PASSANT(qx,px)∧(¬CASTLING(qx,px)∧(¬PAWNPROM Move
px∧BOARD(px,GIVEN)))))∧(From Move px=BQB2∧To Move px=To Move px))⊃(MOVETO(
Tboard qx,Val(px,Mover Move px),BQB2,To Move px)∧(¬(Valueon(GIVEN,To Move px
)=UD)⊃(MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,To Move px)∧((
WHITETURN px⊃BVALUES Valueon(GIVEN,To Move px))∧(¬WHITETURN px⊃WVALUES
Valueon(GIVEN,To Move px))))))
```

3.3. There are ten white pieces on the board GIVEN. Could any of them have made the last move, out from between the rook and the king?

```
*****VE GivenWV To Move px;
30 WVALUES Valueon(GIVEN,To Move px)⊃(To Move px=BKR1∨(To Move px=BQ1∨(To
Move px=BQ2∨(To Move px=WQR2∨(To Move px=WQN3∨(To Move px=WQB2∨(To Move px=
WQ3∨(To Move px=WKB2∨(To Move px=WKN3∨To Move px=WKR2)))))))))
```

3.3.1. Obviously, the king on BKR1 could not have made that jump.

```
*****VE MayMove Tboard qx Valueon(GIVEN,To Move px) BQB2 BKR1;
31 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,BKR1)⊃(Column BQB2=Column
 BKR1∨(KNIGHTMOVE(BQB2,BKR1)∨(Row BQB2=Row BKR1∨(SAMEDIAG(BQB2,BKR1)∨(
KINGMOVE(BQB2,BKR1)∨(TWOTOUCHING(Column BQB2,Column BKR1)∧(WSUC(Row BQB2,Row
 BKR1)∨BSUC(Row BQB2,Row BKR1)))))))

*****simplify ↑;
32 ¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,BKR1)
```

3.3.2. We check each of the white pawns on GIVEN, and observe (using our *Chess Eye*, the simplification mechanism) that none of them could have just moved from BQB2.

```
*****VE MayMove Tboard qx Valueon(GIVEN,To Move px) BQB2 WKB2;
33 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WKB2)⊃(Column BQB2=Column
 WKB2∨(KNIGHTMOVE(BQB2,WKB2)∨(Row BQB2=Row WKB2∨(SAMEDIAG(BQB2,WKB2)∨(
KINGMOVE(BQB2,WKB2)∨(TWOTOUCHING(Column BQB2,Column WKB2)∧(WSUC(Row BQB2,Row
 WKB2)∨BSUC(Row BQB2,Row WKB2)))))))

*****simplify ↑;
34 ¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WKB2)

*****VE MayMove Tboard qx Valueon(GIVEN,To Move px) BQB2 WQ3;
35 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQ3)⊃(Column BQB2=Column
WQ3∨(KNIGHTMOVE(BQB2,WQ3)∨(Row BQB2=Row WQ3∨(SAMEDIAG(BQB2,WQ3)∨(KINGMOVE(
BQB2,WQ3)∨(TWOTOUCHING(Column BQB2,Column WQ3)∧(WSUC(Row BQB2,Row WQ3)∨BSUC(
Row BQB2,Row WQ3)))))))

*****simplify ↑;
36 ¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQ3)

*****VE MayMove Tboard qx Valueon(GIVEN,To Move px) BQB2 WQN3;
37 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQN3)⊃(Column BQB2=Column
 WQN3∨(KNIGHTMOVE(BQB2,WQN3)∨(Row BQB2=Row WQN3∨(SAMEDIAG(BQB2,WQN3)∨(
KINGMOVE(BQB2,WQN3)∨(TWOTOUCHING(Column BQB2,Column WQN3)∧(WSUC(Row BQB2,Row
 WQN3)∨BSUC(Row BQB2,Row WQN3)))))))

*****simplify ↑;
```

```
38 ¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQN3)

*****VE MayMove Tboard qx Valueon(GIVEN,To Move px) BQB2 WQR2;
39 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQR2)⊃(Column BQB2=Column
 WQR2∨(KNIGHTMOVE(BQB2,WQR2)∨(Row BQB2=Row WQR2∨(SAMEDIAG(BQB2,WQR2)∨(
KINGMOVE(BQB2,WQR2)∨(TWOTOUCHING(Column BQB2,Column WQR2)∧(WSUC(Row BQB2,Row
 WQR2)∨BSUC(Row BQB2,Row WQR2))))))))

*****simplify ↑;
40 ¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQR2)
```

Note that we are invoking two different lemmas here. ***MayMove*** is useful for showing that, between a pair of squares, *no* piece can ever move. ***WhitePawnMovement*** (white pawn motion) is more specific: it applies only to white pawns, and is basically a telescoping of the conditions on white pawn movement (as defined in the axioms ***MOVING1*** and ***PAWNMOVING***) so that they can be checked in a single simplification.

```
*****VE WhitePawnMovement Tboard qx GIVEN To Move px BQB2 WKR2;
41 To Move px=WKR2⊃(Valueon(GIVEN,WKR2)=PW⊃(MOVETO(Tboard qx,Valueon(GIVEN,
To Move px),BQB2,WKR2)≡((Column BQB2=Column WKR2∧(WSUC(Row BQB2,Row WKR2)∧
Valueon(Tboard qx,WKR2)=MT))∨((Column BQB2=Column WKR2∧(Row BQB2=7∧(Valueon(
Tboard qx,Makesquare(6,Column BQB2))=MT∧Row WKR2=5)))∨(Valueon(Tboard qx,
WKR2)=MT∧(TWOTOUCHING(Column BQB2,Column WKR2)∧(WSUC(Row BQB2,Row WKR2)∧
BVALUES Valueon(Tboard qx,WKR2))))))))

*****simplify ↑;
42 To Move px=WKR2⊃¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WKR2)

*****VE WhitePawnMovement Tboard qx GIVEN To Move px BQB2 WKN3;
43 To Move px=WKN3⊃(Valueon(GIVEN,WKN3)=PW⊃(MOVETO(Tboard qx,Valueon(GIVEN,
To Move px),BQB2,WKN3)≡((Column BQB2=Column WKN3∧(WSUC(Row BQB2,Row WKN3)∧
Valueon(Tboard qx,WKN3)=MT))∨((Column BQB2=Column WKN3∧(Row BQB2=7∧(Valueon(
Tboard qx,Makesquare(6,Column BQB2))=MT∧Row WKN3=5)))∨(Valueon(Tboard qx,
WKN3)=MT∧(TWOTOUCHING(Column BQB2,Column WKN3)∧(WSUC(Row BQB2,Row WKN3)∧
BVALUES Valueon(Tboard qx,WKN3))))))))

*****simplify ↑;
44 To Move px=WKN3⊃¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WKN3)

*****VE WhitePawnMovement Tboard qx GIVEN To Move px BQB2 WQB2;
45 To Move px=WQB2⊃(Valueon(GIVEN,WQB2)=PW⊃(MOVETO(Tboard qx,Valueon(GIVEN,
To Move px),BQB2,WQB2)≡((Column BQB2=Column WQB2∧(WSUC(Row BQB2,Row WQB2)∧
Valueon(Tboard qx,WQB2)=MT))∨((Column BQB2=Column WQB2∧(Row BQB2=7∧(Valueon(
Tboard qx,Makesquare(6,Column BQB2))=MT∧Row WQB2=5)))∨(Valueon(Tboard qx,
WQB2)=MT∧(TWOTOUCHING(Column BQB2,Column WQB2)∧(WSUC(Row BQB2,Row WQB2)∧
BVALUES Valueon(Tboard qx,WQB2))))))))

*****simplify ↑;
46 To Move px=WQB2⊃¬MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,WQB2)
```

3.3.3. We have already eliminated the checking rook as the moving piece in the last move.

```
*****assume To Move px=BQ2;
47 To Move px=BQ2  (47)
```

```
*****subst ↑ IN ORDPX;
48 (ORDINARY Move px∧SQUARE_BETWEEN(To Move px,From Move px,BQN2))∧¬(Mover
Move px=Pos(px,To Move px))  (1 47)

*****⊃I ↑↑⊃↑;
49 To Move px=RQ2⊃((ORDINARY Move px∧SQUARE_BETWEEN(To Move px,From Move px,
BQN2))∧¬(Mover Move px=Pos(px,To Move px)))  (1)
```

3.3.4. Nor could the rook on BQ1, if it was a rook in the last position, have moved on the diagonal.

```
*****VE MOVING2 Tboard qx BQB2 BQ1;
50 ORTHO(Tboard qx,BQB2,BQ1)=(¬(BQB2=BQ1)∧((Column BQB2=Column BQ1∧∀sq3.((
BETWEEN(Row BQB2,Row sq3,Row BQ1)∧Column sq3=Column BQB2)⊃Valueon(Tboard qx,
sq3)=MT))∨(Row BQB2=Row BQ1∧∀sq3.((BETWEEN(Column BQB2,Column sq3,Column BQ1
)∧Row sq3=Row BQB2)⊃Valueon(Tboard qx,sq3)=MT))))

*****simplify ¬Column BQB2=Column BQ1∧¬Row BQB2=Row BQ1∧
*  Valueon(GIVEN,BQ1)=RW∧VALUER RW∧¬VALUEB RW∧¬VALUEN RW∧¬VALUEK RW∧
*  ¬VALUEP RW∧¬VALUEQ RW;
51 ¬(Column BQB2=Column BQ1)∧(¬(Row BQB2=Row BQ1)∧(Valueon(GIVEN,BQ1)=RW∧(
VALUER RW∧(¬VALUEB RW∧(¬VALUEN RW∧(¬VALUEK RW∧(¬VALUEP RW∧¬VALUEQ RW)))))))

*****VE MOVING1 Tboard qx Valueon(GIVEN BQ1) BQB2 BQ1;
52 MOVETO(Tboard qx,Valueon(GIVEN,BQ1),BQB2,BQ1)=((VALUER Valueon(GIVEN,BQ1)
∧ORTHO(Tboard qx,BQB2,BQ1))∨((VALUEB Valueon(GIVEN,BQ1)∧DIAG(Tboard qx,BQB2,
BQ1))∨((VALUEQ Valueon(GIVEN,BQ1)∧ORTHO(Tboard qx,BQB2,BQ1))∨((VALUEQ
Valueon(GIVEN,BQ1)∧DIAG(Tboard qx,BQB2,BQ1))∨((VALUEK Valueon(GIVEN,BQ1)∧
KINGMOVE(BQB2,BQ1))∨((VALUEN Valueon(GIVEN,BQ1)∧KNIGHTMOVE(BQB2,BQ1))∨(
VALUEP Valueon(GIVEN,BQ1)∧PAWNMOVE(Tboard qx,Valueon(GIVEN,BQ1),BQB2,BQ1))))
))))

*****assume To Move px=BQ1;
53 To Move px=BQ1  (53)

*****subst ↑ IN ↑↑ OCC 1,2;
54 MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,To Move px)=((VALUER
Valueon(GIVEN,BQ1)∧ORTHO(Tboard qx,BQB2,BQ1))∨((VALUEB Valueon(GIVEN,BQ1)∧
DIAG(Tboard qx,BQB2,BQ1))∨((VALUEQ Valueon(GIVEN,BQ1)∧ORTHO(Tboard qx,BQB2,
BQ1))∨((VALUEQ Valueon(GIVEN,BQ1)∧DIAG(Tboard qx,BQB2,BQ1))∨((VALUEK Valueon
(GIVEN,BQ1)∧KINGMOVE(BQB2,BQ1))∨((VALUEN Valueon(GIVEN,BQ1)∧KNIGHTMOVE(BQB2,
BQ1))∨(VALUEP Valueon(GIVEN,BQ1)∧PAWNMOVE(Tboard qx,Valueon(GIVEN,BQ1),BQB2,
BQ1)))))))))  (53)

*****⊃I ↑↑⊃↑;
55 To Move px=BQ1⊃(MOVETO(Tboard qx,Valueon(GIVEN,To Move px),BQB2,To Move
px)=((VALUER Valueon(GIVEN,BQ1)∧ORTHO(Tboard qx,BQB2,BQ1))∨((VALUEB Valueon(
GIVEN,BQ1)∧DIAG(Tboard qx,BQB2,BQ1))∨((VALUEQ Valueon(GIVEN,BQ1)∧ORTHO(
Tboard qx,BQB2,BQ1))∨((VALUEQ Valueon(GIVEN,BQ1)∧DIAG(Tboard qx,BQB2,BQ1))∨(
(VALUEK Valueon(GIVEN,BQ1)∧KINGMOVE(BQB2,BQ1))∨((VALUEN Valueon(GIVEN,BQ1)∧
KNIGHTMOVE(BQB2,BQ1))∨(VALUEP Valueon(GIVEN,BQ1)∧PAWNMOVE(Tboard qx,Valueon(
GIVEN,BQ1),BQB2,BQ1)))))))))
```

3.3.5. We have shown that (if the last move was not a pawn promotion), none of the white pieces on the board could have moved out from between the rook and king, discovering the check. We

must also consider each of the undefined pieces on GIVEN. We can see (simplify) that there is only one such piece, and can show that, (if it was white) it was also incapable (no matter what value it might have had) of discovering the check. Once again we turn to the lemma ***MovedValues***, and observe that no piece can make that giant knight's move.

```
*****VE GivenUD To Move px;
56 Valueon(GIVEN,To Move px)=UD∧To Move px=WKR4

*****VE MayMove Tboard qx Valueon(Tboard qx From Move px) BQB2 WKR4;
57 MOVETO(Tboard qx,Valueon(Tboard qx,From Move px).BQB2,WKR4)⊃(Column BQB2=
Column WKR4∨(KNIGHTMOVE(BQB2.WKR4)∨(Row BQB2=Row WKR4∨(SAMEDIAG(BQB2,WKR4)∨(
KINGMOVE(BQB2,WKR4)∨(TWOTOUCHING(Column BQB2,Column WKR4)∧(WSUC(Row BQB2,Row
 WKR4)∨BSUC(Row BQB2,Row WKR4)))))))

*****simplify ↑;
58 ¬MOVETO(Tboard qx,Valueon(Tboard qx,From Move px),BQB2,WKR4)

*****VE MCONSEQK qx px;
59 SIMPLELEGALMOVE(qx,px)=(¬(From Move px=To Move px)∧(MOVETO(Tboard qx,
Valueon(Tboard qx,From Move px),From Move px,To Move px)∧((SIMPLE Move px∧
Valueon(Tboard qx,To Move px)=MT)∨(CAPTURE Move px∧(PIECEVALUES Valueon(
Tboard qx,To Move px)∧¬(Valuecolor Valueon(Tboard qx,To Move px)=Color qx)))
)))
```

3.4. It therefore tautologically follows (all other alternatives having been disposed), that the last move must have been a pawn promotion.

```
*****label PROMPX;
*****tauteq PAWNPROM Move px
*     PXSUC,NOTPXCASTLE,NOTPXEP,FROMPX,CALL_PX,BLACK_GOES,BINCHECK,
*     MCONAPX,29,30,32,34,36,38,40,42,44,46,49,50,51,55,56,58,59;
60 PAWNPROM Move px  (1)
```

Section 4.2.4 How the Pawn Promoted

4. The promoting pawn could, of course, have moved to only one of three squares. In any case, the square he moved to must now have a white piece on it. We prove a lemma (section B.3) to condense this computation. This lemma states that, for any position just reached by a pawn promotion, a promoting white pawn on BQB2 could have moved to one of three squares, BQB1 by a simple move, or BQN1 or BQ1 by a capture. In either case, there is now a white piece on any board of that position (that isn't undefined on those squares). In the latter case, there must have been a black piece on the capture square of the previous position's board.

```
*****VE PXPawnTo qx,px,GIVEN;
61 (SUCCESSOR(qx,px)∧(¬CASTLING(qx,px)∧(¬EN_PASSANT(qx,px)∧(PAWNPROM Move px
∧(¬WHITETURN px∧(From Move px=BQB2∧BOARD(px,GIVEN)))))))⊃((To Move px=BQN1∧(
(WVALUES Valueon(GIVEN,BQN1)∨Valueon(GIVEN,BQN1)=UD)∧BVALUES Valueon(Tboard
qx,To Move px)))∨((To Move px=BQ1∧((WVALUES Valueon(GIVEN,BQ1)∨Valueon(GIVEN
,BQ1)=UD)∧BVALUES Valueon(Tboard qx,To Move px)))∨(To Move px=BQB1∧(WVALUES
Valueon(GIVEN,BQB1)∨Valueon(GIVEN,BQB1)=UD))))
```

4.1. We observe that only one of these three squares has a white piece on it on the board GIVEN.

```
*****simplify ↑;
62 (SUCCESSOR(qx,px)∧(¬CASTLING(qx,px)∧(¬EN_PASSANT(qx,px)∧(PAWNPROM Move px
∧(¬WHITETURN px∧(From Move px=BQB2∧BOARD(px,GIVEN)))))))⊃(To Move px=BQ1∧
BVALUES Valueon(Tboard qx,To Move px))
```

5. Hence, the destination (To) square of the last move must have been BQ1. Additionally, this pawn promotion resulted in the capture of some black piece.

```
*****VE ValueTranspositionC qx,To Move px;
63 Valueon(Tboard qx,To Move px)=Val(qx,Pos(qx,To Move px))

*****VE ColorChoices qx,Pos(qx,To Move px);
64 (BVALUES Val(qx,Pos(qx,To Move px))=BLACKPIECE Pos(qx,To Move px))∧(
WVALUES Val(qx,Pos(qx,To Move px))=WHITEPIECE Pos(qx,To Move px))

*****tauteq To Move px=BQ1∧BLACKPIECE Pos(qx,To Move px) ∧
*     BVALUES Valueon(Tboard qx To Move px)
*  CALL_PX:BLACK_GOES,PXSUC,NOTPXCASTLE,NOTPXEP,FROMPX,PROMPX,↑↑↑:↑;
65 To Move px=BQ1∧(BLACKPIECE Pos(qx,To Move px)∧BVALUES Valueon(Tboard qx,
To Move px))  (1 9)

*****label TOPX;
***** taut ↑:#1 ↑;
66 To Move px=BQ1  (1)
```

Let us call that black piece zb.

```
*****tauteq Pos(qx To Move px)=Pos(qx To Move px)  ;
67 Pos(qx,To Move px)=Pos(qx,To Move px)

*****∃I ↑, ↑:#2 ← zb OCC 2;
68 BLACKPIECE Pos(qx,To Move px)⊃∃zb.Pos(qx,To Move px)=zb

*****taut ∃zb.Pos(qx To Move px)=zb ↑,↑↑↑↑;
69 ∃zb.Pos(qx,To Move px)=zb  (1 9)

*****label CALL_ZB;
*****∃E ↑ zb;
70 Pos(qx,To Move px)=zb  (70)
```

We proceed to seek the identity of the captured black piece.

5.1. Black's king is on BQN2. As white moved last, and didn't capture this king, we know that he was on BQN2 in qx. A black king that just castled would not be on this square. Hence, we can conclude that black has not just finished a castling move.

```
*****label PX_BK;
*****VE KingValueThm px,GIVEN,BQN2;
71 (BOARD(px,GIVEN)∧¬(Valueon(GIVEN,BQN2)=UD))⊃((Pos(px,BQN2)=WK=Valueon(
GIVEN,BQN2)=KW)∧(Pos(px,BQN2)=BK=Valueon(GIVEN,BQN2)=KB))

*****VE OtherSideStays qx,px,BQN2,BK;
72 (SUCCESSOR(qx,px)∧((WHITEPIECE BK=WHITETURN px)∧Pos(px,BQN2)=BK))⊃Pos(qx,
BQN2)=BK
```

```
*****VE BlackCastleThm Prevpos qx,qx,BQN2;
73 (SUCCESSOR(Prevpos qx,qx)∧(CASTLING(Prevpos qx,qx)∧WHITETURN qx))⊃(Pos(qx
,BQN2)=BK⊃(BQN2=BKN1∨BQN2=BQB1))

*****simplify Valueon(GIVEN,BQN2)=KB∧¬KB=UD∧¬WHITEPIECE BK∧
*     ¬BQN2=BKN1∧¬BQN2=BQB1;
74 Valueon(GIVEN,BQN2)=KB∧(¬(KB=UD)∧(¬WHITEPIECE BK∧(¬(BQN2=BKN1)∧¬(BQN2=
BQB1))))

*****label NOTQXCASTLE;
***** tauteq ¬CASTLING(Prevpos qx qx)
*     PXSUC,CALL_PX,BINCHECK,BLACK_GOES,MCONAPX,QXIS,↑↑↑↑;↑;
75 ¬CASTLING(Prevpos qx,qx)  (1 9)
```

Proving that black's last move was not an *en passant* capture is slightly more difficult. More particularly, we must account for either each of his pawns, each of the squares that a black pawn, capturing *en passant*, would land in, or demonstrate the existence of all of the white pawns. However, it is sufficient for our purposes to show that if QBUD is a board of qx, then a capture *en passant* was not just completed.

```
*****VE BlackEnPassantThm2 Prevpos qx qx QBUD;
76 (SUCCESSOR(Prevpos qx,qx)∧(EN_PASSANT(Prevpos qx,qx)∧WHITETURN qx))⊃(∀dcx
.¬(Valueon(QBUD,Makesquare(6,dcx))=PB∨Valueon(QBUD,Makesquare(6,dcx))=UD)⊃¬
BOARD(qx,QBUD))

*****simplify ↑;
77 (SUCCESSOR(Prevpos qx,qx)∧(EN_PASSANT(Prevpos qx,qx)∧WHITETURN qx))⊃¬
BOARD(qx,QBUD)

*****label NOTQBUDEP;
***** tauteq BOARD(qx QBUD)⊃¬EN_PASSANT(Prevpos qx qx)
*     ↑,PXSUC,QXIS,MCONAPX,BLACK_GOES,CALL_PX,BINCHECK;
78 BOARD(qx,QBUD)⊃¬EN_PASSANT(Prevpos qx,qx)  (1 9)
```

Section 4.2.4.1 The Pawn Did Not Capture a Rook or Queen

5.2. We proceed by assuming the promoting white pawn captured a black rook or queen (valued) piece on BQ1. This part of the proof is the first time we employ any of the *move undoing* functions, UNMK___MOVE. The axiom delimiting their use requires we establish the sort of the last move.

We know that all pawn promotions are ordinary moves, and that any move (by white) to a square occupied by a black piece is a capture.

```
*****simplify ∀mpp.ORDINARY mpp;
79 ∀mpp.ORDINARY mpp

*****VE ↑ Move px;
80 PAWNPROM Move px⊃ORDINARY Move px

*****VE BlackCapturedThm px To Move px;
81 To Move px=To Move px⊃((ORDINARY Move px∧BVALUES Valueon(Tboard Prevpos
px,To Move px))⊃CAPTURE Move px)
```

```
*****label CAPTURE_PX;
*****substr CALL_QX IN ↑;
82 To Move px=To Move px⊃((ORDINARY Move px∧BVALUES Valueon(Tboard qx,To
Move px))⊃CAPTURE Move px)  (9)
```

We can therefore conclude that the last move was a capturing pawn promotion.

```
*****VE CAPPP_SortThm Move px;
83 (PAWNPROM Move px∧CAPTURE Move px)⊃CAPPP Move px

*****label CAPPPPX;
***** tauteq CAPPP Move px ↑,↑↑,↑↑↑↑,TOPX-1,PROMPX;
84 CAPPP Move px  (1)
```

If the last move (the capturing pawn promotion) captured a black rook, then the board QBR was a board of that position. If it captured a queen, then QBQ.

```
*****VE UNDO4 qx px GIVEN BQB2 BQ1 RB;
85 (SUCCESSOR(qx,px)∧(BOARD(px,GIVEN)∧(CAPPP Move px∧(Val(qx,Taken Move px)=
RB∧(From Move px=BQB2∧To Move px=BQ1)))))⊃BOARD(qx,Unmkcapppmove(GIVEN,BQB2,
BQ1,RB))

*****VE UNDO4 qx px GIVEN BQB2 BQ1 QB;
86 (SUCCESSOR(qx,px)∧(BOARD(px,GIVEN)∧(CAPPP Move px∧(Val(qx,Taken Move px)=
QB∧(From Move px=BQB2∧To Move px=BQ1)))))⊃BOARD(qx,Unmkcapppmove(GIVEN,BQB2,
BQ1,QB))
```

The board QBUD is a sub-board of both.

```
*****simplify ↑↑:#2#2=QBR∧↑:#2#2=QBQ∧SUBOARD(QBUD,QBQ)∧SUBOARD(QBUD,QBR);
87 Unmkcapppmove(GIVEN,BQB2,BQ1,RB)=QBR∧(Unmkcapppmove(GIVEN,BQB2,BQ1,QB)=
QBQ∧(SUBOARD(QBUD,QBQ)∧SUBOARD(QBUD,QBR)))

*****VE SubboardTransitivityX QBUD QBQ qx;
88 (SUBOARD(QBUD,QBQ)∧BOARD(qx,QBQ))⊃BOARD(qx,QBUD)

*****VE SubboardTransitivityX QBUD QBR qx;
89 (SUBOARD(QBUD,QBR)∧BOARD(qx,QBR))⊃BOARD(qx,QBUD)
```

Therefore, if the captured piece (zb) was rook valued, QBR is a board of qx; if queen valued, QBQ. In either case, QBUD is a board of qx.

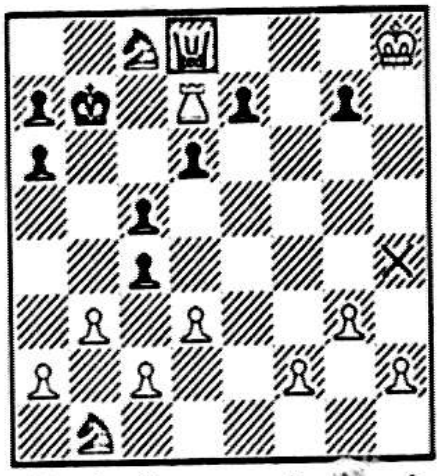
```
*****tauteq     (Val(qx Taken Move px)=RB ⊃ BOARD(qx QBR)) ∧
*      (Val(qx Taken Move px)=QB ⊃ BOARD(qx QBQ)) ∧
*      ((Val(qx Taken Move px)=RB ∨Val(qx Taken Move px)=QB ) ⊃
*               BOARD(qx QBUD))
*      ↑↑↑↑↑↑:↑,PXSUC,CALL_PX,FROMPX,TOPX;
90 (Val(qx,Taken Move px)=RB⊃BOARD(qx,QBR))∧((Val(qx,Taken Move px)=QB⊃BOARD
(qx,QBQ))∧((Val(qx,Taken Move px)=RB∨Val(qx,Taken Move px)=QB)⊃BOARD(qx,QBUD
)))  (1 9)
```

We know that the captured piece of position px was zb. We substitute that equivalence into the previous conclusion.

```
*****tauteq Taken Move px = zb
*     MCONAPX PXSUC,CAPTURE_PX,CAPTURE_PX-2, PROMPX,TOPX-1,CALL_ZB;
91 Taken Move px=zb  (1 9 70)

*****labe' QBUDLBL;
***** substr ↑ IN ↑↑;
92 (Val(qx,zb)=RB⊃BOARD(qx,QBR))∧((Val(qx,zb)=QB⊃BOARD(qx,QBQ))∧((Val(qx,zb)
=RB∨Val(qx,zb)=QB)⊃BOARD(qx,QBUD)))  (1 9 70)
```

Section 4.2.4.1.1 The Cornered Rook or Queen

5.2 1. Just as the white rook on BQ2 was *cornered*, unable to have moved into its check, the (presumed) black rook or queen on BQ1 is cornered. We use the same theorem to show its last move was a discovered check.



The checking queen is trapped.

figure 37

```
*****VE BlackCornered Prevpos qx qx QBQ QB BQ1 BKR1;
93 (SUCCESSOR(Prevpos qx,qx)∧(¬EN_PASSANT(Prevpos qx,qx)∧(¬CASTLING(Prevpos
qx,qx)∧(¬BLASTRANK BQ1∧((BOARD(qx,QBQ)∧(Valueon(QBQ,BQ1)=QB∧(Valueon(QBQ,
BKR1)=KW∧MOVETO(QBQ,QB,BQ1,BKR1))))∧¬VALUEP QB)))))⊃(∀sq1.(MOVETO(Tboard qx,
QB,BQ1,sq1)⊃(¬(Valueon(Tboard qx,sq1)=MT)∨MOVETO(Tboard qx,QB,BKR1,sq1)))⊃((
ORDINARY Move qx∧SQUARE_BETWEEN(BQ1,From Move qx,BKR1))∧¬(Mover Move qx=Pos(
qx,BQ1))))

*****VE BlackCornered Prevpos qx qx QBR RB BQ1 BKR1;
94 (SUCCESSOR(Prevpos qx,qx)∧(¬EN_PASSANT(Prevpos qx,qx)∧(¬CASTLING(Prevpos
qx,qx)∧(¬BLASTRANK BQ1∧((BOARD(qx,QBR)∧(Valueon(QBR,BQ1)=RB∧(Valueon(QBR,
BKR1)=KW∧MOVETO(QBR,RB,BQ1,BKR1))))∧¬VALUEP RB)))))⊃(∀sq1.(MOVETO(Tboard qx,
RB,BQ1,sq1)⊃(¬(Valueon(Tboard qx,sq1)=MT)∨MOVETO(Tboard qx,RB,BKR1,sq1)))⊃((
ORDINARY Move qx∧SQUARE_BETWEEN(BQ1,From Move qx,BKR1))∧¬(Mover Move qx=Pos(
qx,BQ1))))
```

The quantified portion of the premise of this WFF is somewhat more complex. We need to prove theorems about the movements of the pieces on the total boards of qx, when we have only partial boards. Once again, we retreat to the security of a lemma. The theorem *TRAPPED_QX_BQ1_THM*, proven in section B.4, shows that, for any position which has QBUD as a board, a rook or queen valued piece is *cornered* on BQ1, in just the form we need for steps 93 and 94.

```
*****VE Trapped_QX_QB1_Thm qx QB;
95 BOARD(qx,QBUD)⊃((QB=RB∨QB=QB)⊃∀sq1.(MOVETO(Tboard qx,QB,BQ1,sq1)⊃(¬(
Valueon(Tboard qx,sq1)=MT)∨MOVETO(Tboard qx,QB,BKR1,sq1))))

*****VE Trapped_QX_QB1_Thm qx RB;
96 BOARD(qx,QBUD)⊃((RB=RB∨RB=QB)⊃∀sq1.(MOVETO(Tboard qx,RB,BQ1,sq1)⊃(¬(
Valueon(Tboard qx,sq1)=MT)∨MOVETO(Tboard qx,RB,BKR1,sq1))))
```

Other conditions for this theorem are more easily established. For example, we can observe that, on both BQR and BQB, the white king on BKR1 is checked by a black officer on BQ1. We imply that this officer did not just complete a promotion move.

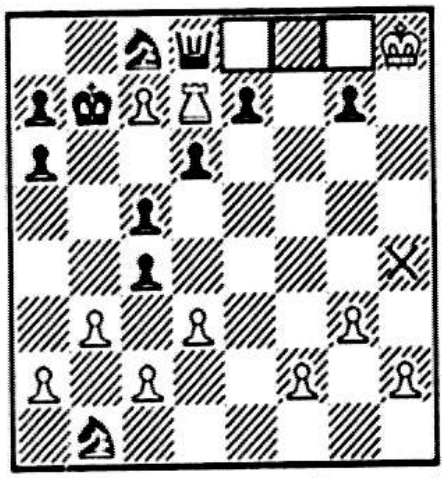
```
*****simplify ¬BLASTRANK BQ1∧¬VALUEP QB∧¬VALUEP RB∧MOVETO(QBR,RB,BQ1,BKR1)
*     ∧MOVETO(QBQ,QB,BQ1,BKR1)∧Valueon(QBQ,BQ1)=QB∧Valueon(QBR,BQ1)=RB
*     ∧Valueon(QBQ,BKR1)=KW∧Valueon(QBR,BKR1)=KW;
97 ¬BLASTRANK BQ1∧(¬VALUEP QB∧(¬VALUEP RB∧(MOVETO(QBR,RB,BQ1,BKR1)∧(MOVETO(
QBQ,QB,BQ1,BKR1)∧(Valueon(QBQ,BQ1)=QB∧(Valueon(QBR,BQ1)=RB∧(Valueon(QBQ,BKR1
)=KW∧Valueon(QBR,BKR1)=KW)))))))
```

5.3. It therefore tautologically follows that, if zb was a black rook or queen, the check must have been a discovered check.

```
*****label DISQX;
***** tauteq (Val(qx,zb)=RB∨Val(qx,zb)=QB)⊃((ORDINARY Move qx∧
*  SQUARE_BETWEEN(BQ1,From Move qx,BKR1))∧¬Mover Move qx=Pos(qx,BQ1))
*        QXIS,NOTQXCASTLE,NOTQBUDEP,QBUDLBL:↑;
98 (Val(qx,zb)=RB∨Val(qx,zb)=QB)⊃((ORDINARY Move qx∧SQUARE_BETWEEN(BQ1,From
Move qx,BKR1))∧¬(Mover Move qx=Pos(qx,BQ1)))  (1 9 70)
```

Section 4.2.4.1.2 Which Piece Discovered the Check

We have concluded that the discovering move must have started upon a square between the (presumed) queen (or rook), and the white king. We consult the simplification mechanism, which informs us that the only squares between these two are BK1, BKB1, and BKN1. Hence, (if the captured piece had rook or queen value), one of these squares must have been the From square of the last move (*figure 38*).



If a black piece moved to discover check, then it moved from one of these squares.

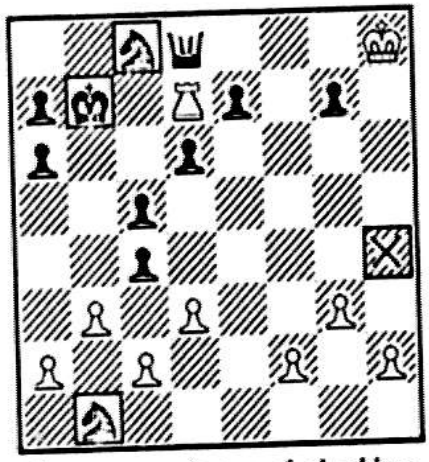
figure 38

```
*****simplify Vsq.(SQUARE_BETWEEN(BQ1,sq,BKR1)⊃(sq=BK1v(sq=BKB1vsq=BKN1)));
99 Vsq.(SQUARE_BETWEEN(BQ1,sq,BKR1)⊃(sq=BK1v(sq=BKB1vsq=BKN1)))

*****VE ↑ From Move qx:
100 SQUARE_BETWEEN(BQ1,From Move qx,BKR1)⊃(From Move qx=BK1v(From Move qx=
BKB1vFrom Move qx=BKN1))
```

5.3.1. So we must consider each of the black (possibly) pieces, to determine if any of them could have moved from one of these three squares on the last move. The theorem *NotBPFrom1Thm* is useful in this respect. From several suitable premises (black's move, the source square of the move is on the first row, and this isn't a *special* move) it permits various useful conclusions. Most relevantly, it asserts that the destination of the last move is now either occupied by a non-pawn, black value, or by the undefined value, and that the last move needed to satisfy several MOVETO conditions.

```
*****VE NotBPFrom1Thm qx,QBUD:
101 (¬CASTLING(Prevpos qx,qx)∧(BOARD(qx,QBUD)∧(¬EN_PASSANT(Prevpos qx,qx)∧(
WHITETURN qx∧Row From Move qx=1))))⊃((¬(Valueon(QBUD,To Move qx)=UD)⊃MOVETO(
Tboard Prevpos qx,Valueon(QBUD,To Move qx),From Move qx,To Move qx))∧(¬
PAWNPROM Move qx∧(MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),
From Move qx,To Move qx)∧(¬(Valueon(QBUD,To Move qx)=UD)⊃(¬VALUEP Valueon(
QBUD,To Move qx)∧BVALUES Valueon(QBUD,To Move qx))))))
```

This simplifies our task enormously. There are now only four possible destination squares for the last move, BQN2, occupied by the black king, BQB1 and WQN1, occupied by black knights, and, of course, WKR4, whose occupant is still unclear. The other undefined square (of the partial board we compute upon), BQ1, has already been dismissed as a possible destination.



The squares that need checking.

figure 39

```
*****simplify ∀sq.((BVALUES Valueon(QBUD,sq)∧¬VALUEP Valueon(QBUD,sq))⊃
*       (sq=BQN2∨(sq=BQB1∨sq=WQN1)));
102 ∀sq.((BVALUES Valueon(QBUD,sq)∧¬VALUEP Valueon(QBUD,sq))⊃(sq=BQN2∨(sq=
BQB1∨sq=WQN1)))

*****simplify ∀sq.(Valueon(QBUD,sq)=UD⊃(sq=WKR4∨sq=BQ1));
103 ∀sq.(Valueon(QBUD,sq)=UD⊃(sq=WKR4∨sq=BQ1))

*****VE ↑↑ To Move qx;
104 (BVALUES Valueon(QBUD,To Move qx)∧¬VALUEP Valueon(QBUD,To Move qx))⊃(To
Move qx=BQN2∨(To Move qx=BQB1∨To Move qx=WQN1))

*****VE ↑↑ To Move qx;
105 Valueon(QBUD,To Move qx)=UD⊃(To Move qx=WKR4∨To Move qx=BQ1)
```

We consider each of the possible pieces (and its associated square) in turn, showing how a piece with that value (on QBUD) could not have moved to any of the possible From squares. Note that six steps are required for each piece: three to instantiate the axiom, and three for simplification.

The knight on white's first row:

```
*****VE MayMove Tboard Prevpos qx,NB,BK1,WQN1;
106 MOVETO(Tboard Prevpos qx,NB,BK1,WQN1)⊃(Column BK1=Column WQN1∨(
KNIGHTMOVE(BK1,WQN1)∨(Row BK1=Row WQN1∨(SAMEDIAG(BK1,WQN1)∨(KINGMOVE(BK1,
WQN1)∨(TWOTOUCHING(Column BK1,Column WQN1)∧(WSUC(Row BK1,Row WQN1)∨BSUC(Row
BK1,Row WQN1)))))))))

*****VE MayMove Tboard Prevpos qx,NB,BKB1,WQN1;
107 MOVETO(Tboard Prevpos qx,NB,BKB1,WQN1)⊃(Column BKB1=Column WQN1∨(
KNIGHTMOVE(BKB1,WQN1)∨(Row BKB1=Row WQN1∨(SAMEDIAG(BKB1,WQN1)∨(KINGMOVE(BKB1
,WQN1)∨(TWOTOUCHING(Column BKB1,Column WQN1)∧(WSUC(Row BKB1,Row WQN1)∨BSUC(
Row BKB1,Row WQN1)))))))))

*****VE MayMove Tboard Prevpos qx,NB,BKN1,WQN1;
108 MOVETO(Tboard Prevpos qx,NB,BKN1,WQN1)⊃(Column BKN1=Column WQN1∨(
KNIGHTMOVE(BKN1,WQN1)∨(Row BKN1=Row WQN1∨(SAMEDIAG(BKN1,WQN1)∨(KINGMOVE(BKN1
```

```
,WQN1)∨(TWOTOUCHING(Column BKN1,Column WQN1)∧(WSUC(Row BKN1,Row WQN1)∨BSUC(
Row BKN1,Row WQN1)))))))

*****simplify ↑↑↑;
109 ¬MOVETO(Tboard Prevpos qx,NB,BK1,WQN1)

*****simplify ↑↑↑;
110 ¬MOVETO(Tboard Prevpos qx,NB,BKB1,WQN1)

*****simplify ↑↑↑;
111 ¬MOVETO(Tboard Prevpos qx,NB,BKN1,WQN1)
```

The knight on black's first row:

```
*****VE MOVING1 Tboard Prevpos qx,NB,BK1,BQB1;
112 MOVETO(Tboard Prevpos qx,NB,BK1,BQB1)≡((VALUER NB∧ORTHO(Tboard Prevpos
qx,BK1,BQB1))∨((VALUEB NB∧DIAG(Tboard Prevpos qx,BK1,BQB1))∨((VALUEQ NB∧
ORTHO(Tboard Prevpos qx,BK1,BQB1))∨((VALUEQ NB∧DIAG(Tboard Prevpos qx,BK1,
BQB1))∨((VALUEK NB∧KINGMOVE(BK1,BQB1))∨((VALUEN NB∧KNIGHTMOVE(BK1,BQB1))∨(
VALUEP NB∧PAWNMOVE(Tboard Prevpos qx,NB,BK1,BQB1)))))))

*****VE MOVING1 Tboard Prevpos qx,NB,BKB1,BQB1;
113 MOVETO(Tboard Prevpos qx,NB,BKB1,BQB1)≡((VALUER NB∧ORTHO(Tboard Prevpos
qx,BKB1,BQB1))∨((VALUEB NB∧DIAG(Tboard Prevpos qx,BKB1,BQB1))∨((VALUEQ NB∧
ORTHO(Tboard Prevpos qx,BKB1,BQB1))∨((VALUEQ NB∧DIAG(Tboard Prevpos qx,BKB1,
BQB1))∨((VALUEK NB∧KINGMOVE(BKB1,BQB1))∨((VALUEN NB∧KNIGHTMOVE(BKB1,BQB1))∨(
VALUEP NB∧PAWNMOVE(Tboard Prevpos qx,NB,BKB1,BQB1)))))))

*****VE MOVING1 Tboard Prevpos qx,NB,BKN1,BQB1;
114 MOVETO(Tboard Prevpos qx,NB,BKN1,BQB1)≡((VALUER NB∧ORTHO(Tboard Prevpos
qx,BKN1,BQB1))∨((VALUEB NB∧DIAG(Tboard Prevpos qx,BKN1,BQB1))∨((VALUEQ NB∧
ORTHO(Tboard Prevpos qx,BKN1,BQB1))∨((VALUEQ NB∧DIAG(Tboard Prevpos qx,BKN1,
BQB1))∨((VALUEK NB∧KINGMOVE(BKN1,BQB1))∨((VALUEN NB∧KNIGHTMOVE(BKN1,BQB1))∨(
VALUEP NB∧PAWNMOVE(Tboard Prevpos qx,NB,BKN1,BQB1)))))))

*****simplify ↑↑↑;
115 ¬MOVETO(Tboard Prevpos qx,NB,BK1,BQB1)

*****simplify ↑↑↑;
116 ¬MOVETO(Tboard Prevpos qx,NB,BKB1,BQB1)

*****simplify ↑↑↑;
117 ¬MOVETO(Tboard Prevpos qx,NB,BKN1,BQB1)
```

And, of course, the black king is too far away to have discovered the check.

```
*****VE MOVING1 Tboard Prevpos qx,KB,BK1,BQN2;
118 MOVETO(Tboard Prevpos qx,KB,BK1,BQN2)≡((VALUER KB∧ORTHO(Tboard Prevpos
qx,BK1,BQN2))∨((VALUEB KB∧DIAG(Tboard Prevpos qx,BK1,BQN2))∨((VALUEQ KB∧
ORTHO(Tboard Prevpos qx,BK1,BQN2))∨((VALUEQ KB∧DIAG(Tboard Prevpos qx,BK1,
BQN2))∨((VALUEK KB∧KINGMOVE(BK1,BQN2))∨((VALUEN KB∧KNIGHTMOVE(BK1,BQN2))∨(
VALUEP KB∧PAWNMOVE(Tboard Prevpos qx,KB,BK1,BQN2)))))))

*****VE MOVING1 Tboard Prevpos qx,KB,BKB1,BQN2;
119 MOVETO(Tboard Prevpos qx,KB,BKB1,BQN2)≡((VALUER KB∧ORTHO(Tboard Prevpos
qx,BKB1,BQN2))∨((VALUEB KB∧DIAG(Tboard Prevpos qx,BKB1,BQN2))∨((VALUEQ KB∧
```

```
ORTHO(Tboard Prevpos qx,BKB1,BQN2))∨((VALUEQ KB∧DIAG(Tboard Prevpos qx,BKB1,
BQN2))∨((VALUEK KB∧KINGMOVE(BKB1,BQN2))∨((VALUEN KB∧KNIGHTMOVE(BKB1,BQN2))∨(
VALUEP KB∧PAWNMOVE(Tboard Prevpos qx,KB,BKB1,BQN2))))))))

*****VE MOVING1 Tboard Prevpos qx,KB,BKN1,BQN2;
120 MOVETO(Tboard Prevpos qx,KB,BKN1,BQN2)≡((VALUER KB∧ORTHO(Tboard Prevpos
qx,BKN1,BQN2))∨((VALUEB KB∧DIAG(Tboard Prevpos qx,BKN1,BQN2))∨((VALUEQ KB∧
ORTHO(Tboard Prevpos qx,BKN1,BQN2))∨((VALUEQ KB∧DIAG(Tboard Prevpos qx,BKN1,
BQN2))∨((VALUEK KB∧KINGMOVE(BKN1,BQN2))∨((VALUEN KB∧KNIGHTMOVE(BKN1,BQN2))∨(
VALUEP KB∧PAWNMOVE(Tboard Prevpos qx,KB,BKN1,BQN2))))))))

*****simplify ↑↑↑;
121 ¬MOVETO(Tboard Prevpos qx,KB,BK1,BQN2)

*****simplify ↑↑↑;
122 ¬MOVETO(Tboard Prevpos qx,KB,BKB1,BQN2)

*****simplify ↑↑↑;
123 ¬MOVETO(Tboard Prevpos qx,KB,BKN1,BQN2)
```

A little substitution for the tautology decider.

```
******∧I Substitution[β←λx.Valueon(QBUD,x)];
124 ∀j k.(j=k⊃Valueon(QBUD,j)=Valueon(QBUD,k))

*****VE ↑ To Move qx,BQN2;
125 To Move qx=BQN2⊃Valueon(QBUD,To Move qx)=Valueon(QBUD,BQN2)

*****VE ↑↑ To Move qx,BQB1;
126 To Move qx=BQB1⊃Valueon(QBUD,To Move qx)=Valueon(QBUD,BQB1)

*****VE ↑↑↑ To Move qx,WQN1;
127 To Move qx=WQN1⊃Valueon(QBUD,To Move qx)=Valueon(QBUD,WQN1)
```

And we appeal to the chess eye, to confirm that various squares of QBUD have the values we asserted:

```
*****simplify Valueon(QBUD,BQN2)=KB∧Valueon(QBUD,BQB1)=NB∧
*      Valueon(QBUD,WQN1)=NB;
128 Valueon(QBUD,BQN2)=KB∧(Valueon(QBUD,BQB1)=NB∧Valueon(QBUD,WQN1)=NB)
```

Our attention turns to proving the undefined squares of QBUD do not harbor the last move mover. This piece must, of course, be on the To square of the last move. And we have already determined (step 98) that this is not the square BQ1.

```
*****label MCONAQX;
*****VE MCONSEQA Prevpos qx,qx;
129 SUCCESSOR(Prevpos qx,qx)⊃((¬WHITETURN Prevpos qx≡WHITETURN qx)∧(Prevpos
qx=Prevpos qx∧(¬POSITIONINCHECK(qx,Color Prevpos qx)∧((WHITEPIECE Mover Move
 qx≡WHITETURN Prevpos qx)∧(Pos(Prevpos qx,From Move qx)=Mover Move qx∧(Pos(
qx,To Move qx)=Mover Move qx∧(Pos(qx,From Move qx)=EMPTY∧((CAPTURE Move qx⊃
Pos(Prevpos qx,To Move qx)=Taken Move qx)∧(CASTLING(Prevpos qx,qx)∨(EN_
PASSANT(Prevpos qx,qx)∨SIMPLELEGALMOVE(Prevpos qx,qx))))))))))))

******∧I Substitution[β ← λ x.Pos(qx x)];
130 ∀j k.(j=k⊃Pos(qx,j)=Pos(qx,k))
```

```
*****∀E ↑ To Move qx,BQ1;
131 To Move qx=BQ1⊃Pos(qx,To Move qx)=Pos(qx,BQ1)
```

All of the candidate source squares for this move are in the first row.

```
*****∧I Substitution[β ← λ x.Row x];
132 ∀j k.(j=k⊃Row j=Row k)

*****∀E ↑ From Move qx,BK1;
133 From Move qx=BK1⊃Row From Move qx=Row BK1

*****∀E ↑↑ From Move qx,BKB1;
134 From Move qx=BKB1⊃Row From Move qx=Row BKB1

*****∀E ↑↑↑ From Move qx,BKN1;
135 From Move qx=BKN1⊃Row From Move qx=Row BKN1

*****simplify Row BK1=1∧(Row BKB1=1∧Row BKN1=1);
136 Row BK1=1∧(Row BKB1=1∧Row BKN1=1)
```

And, the fallen piece, no matter what value it might have had could not have moved to one of these first row squares.

```
*****∀E MayMove Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BK1,WKR4;
137 MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BK1,WKR4)⊃(Column
 BK1=Column WKR4∨(KNIGHTMOVE(BK1,WKR4)∨(Row BK1=Row WKR4∨(SAMEDIAG(BK1,WKR4)
∨(KINGMOVE(BK1,WKR4)∨(TWOTOUCHING(Column BK1,Column WKR4)∧(WSUC(Row BK1,Row
WKR4)∨BSUC(Row BK1,Row WKR4))))))))

*****∀E MayMove Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKB1,WKR4;
138 MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKB1,WKR4)⊃(
Column BKB1=Column WKR4∨(KNIGHTMOVE(BKB1,WKR4)∨(Row BKB1=Row WKR4∨(SAMEDIAG(
BKB1,WKR4)∨(KINGMOVE(BKB1,WKR4)∨(TWOTOUCHING(Column BKB1,Column WKR4)∧(WSUC(
Row BKB1,Row WKR4)∨BSUC(Row BKB1,Row WKR4))))))))

*****∀E MayMove Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKN1,WKR4;
139 MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKN1,WKR4)⊃(
Column BKN1=Column WKR4∨(KNIGHTMOVE(BKN1,WKR4)∨(Row BKN1=Row WKR4∨(SAMEDIAG(
BKN1,WKR4)∨(KINGMOVE(BKN1,WKR4)∨(TWOTOUCHING(Column BKN1,Column WKR4)∧(WSUC(
Row BKN1,Row WKR4)∨BSUC(Row BKN1,Row WKR4))))))))

*****simplify ↑↑↑;
140 ¬MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BK1,WKR4)

*****simplify ↑↑↑;
141 ¬MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKB1,WKR4)

*****simplify ↑↑↑;
142 ¬MOVETO(Tboard Prevpos qx,Val(Prevpos qx,Mover Move qx),BKN1,WKR4)
```

It then tautologically follows that the value of the captured piece in qx, zb, was neither a rook nor a queen.

```
*****label NOT QB OR RB;
***** tauteq ¬(Val(qx,zb)=RB∨Val(qx,zb)=QB)
```

```
*     CALL_PX:BLACK_GOES,QXIS,PXSUC,MCONAPX,NOTQXCASTLE,NOTQBUDEP,
*     QBUDLBL,DISQX,MCONAQX,100:101,104:105,109:111,115:117,121:123,
*     125:129,131,133:136,140:142;
143 ¬(Val(qx,zb)=RB∨Val(qx,zb)=QB)  (1 9 70)
```

Section 4.2.4.2 The Pawn Did Not Capture a King or Pawn

There are six varieties of black pieces that the promoting pawn could have captured. We have already eliminated a black rook or queen as a possible victim. What about the others?

5.4. Perhaps the captured piece had pawn value?
5.4.1. But that black pawn would have been in black's first row. We have a theorem that prohibits black pawn (valued) pieces from black first row. Hence, the captured piece (while it might have been a pawn) did not have pawn value.

```
*****VE NoBlackPawnsOn1Row qx,zb,BQ1;
144 (Val(qx,zb)=PB∧Pos(qx,BQ1)=zb)⊃¬(Row BQ1=1)

*****label ON_ZB;
*****substr TOPX IN CALL_ZB;
145 Pos(qx,BQ1)=zb  (1 70)

*****simplify Row BQ1;
146 Row BQ1=1

*****label NOT_ZB_PB;
*****taut ¬(Val(qx,zb)=PB) ↑↑↑:↑;
147 ¬(Val(qx,zb)=PB)  (1 70)
```

5.5. The captured piece was certainly not the black king. We have already shown the black king to be on BQN2 on GIVEN (steps 71 through 74). As the king did not just move, he must still be there.

```
*****VE BlackKingThm qx,BQ1;
148 Val(qx,Pos(qx,BQ1))=KB≡Pos(qx,BQ1)=BK

*****VE Unique qx,BQN2,BQ1,BK;
149 Pos(qx,BQN2)=BK⊃(Pos(qx,BQ1)=BK≡BQN2=BQ1)

*****simplify ¬(Valueon(GIVEN,BQN2)=UD)∧ (Valueon(GIVEN,BQN2)=KB∧
*(¬WHITEPIECE BK∧ ¬(BQN2=BQ1)));
150 ¬(Valueon(GIVEN,BQN2)=UD)∧(Valueon(GIVEN,BQN2)=KB∧(¬WHITEPIECE BK∧¬(BQN2
=BQ1)))

*****tauteq ¬Val(qx,Pos(qx BQ1))=KB
*     ↑↑↑:↑,PXSUC,CALL_PX,BINCHECK,BLACK_GOES,PX_BK,PX_BK+1;
151 ¬(Val(qx,Pos(qx,BQ1))=KB)  (1 9)

*****label NOT_ZB_KB;
*****substr ON_ZB IN ↑;
152 ¬(Val(qx,zb)=KB)  (1 9 70)
```

Hence, the captured piece must have had, just before being captured, either bishop or knight value.

```
*****simplify ∀vb.(vb=KBv∨vb=QBv∨vb=RBv∨vb=PBv∨vb=NBv∨vb=BB);
153 ∀vb.(vb=KBv(vb=QBv(vb=RBv(vb=PBv(vb=NBv∨vb=BB)))))

*****VE ↑ Val(qx zb);
154 BVALUES Val(qx,zb)⊃(Val(qx,zb)=KBv(Val(qx,zb)=QBv(Val(qx,zb)=RBv(Val(qx,
zb)=PBv(Val(qx,zb)=NBv∨Val(qx,zb)=BB)))))

*****VE ValueTranspositionA qx zb To Move px;
155 Pos(qx,To Move px)=zb⊃Valueon(Tboard qx,To Move px)=Val(qx,zb)

*****label NB_OR_BB;
*****tauteq Val(qx zb)=NBv∨Val(qx zb)=BB
*      ↑↑:↑,TOPX-1,CALL_ZB,NOT_QB_OR_RB,NOT_ZB_PB,NOT_ZB_KB;
156 Val(qx,zb)=NBv∨Val(qx,zb)=BB  (1 9 70)
```

And we can also deduce, from this limited selection, that QBUD was, in either case, a board of the position qx.

```
*****VE TransitiveUNMKCAPPP qx,GIVEN,QBUD,BQB2,BQ1,Val(qx,zb),BB;
157 (BOARD(qx,Unmkcapppmove(GIVEN,BQB2,BQ1,Val(qx,zb)))∧(SUBOARD(QBUD,
Unmkcapppmove(GIVEN,BQB2,BQ1,BB))∧Val(qx,zb)=BB))⊃BOARD(qx,QBUD)

*****VE TransitiveUNMKCAPPP qx,GIVEN,QBUD,BQB2,BQ1,Val(qx,zb),NB;
158 (BOARD(qx,Unmkcapppmove(GIVEN,BQB2,BQ1,Val(qx,zb)))∧(SUBOARD(QBUD,
Unmkcapppmove(GIVEN,BQB2,BQ1,NB))∧Val(qx,zb)=NB))⊃BOARD(qx,QBUD)

*****simplify ↑↑;
159 (BOARD(qx,Unmkcapppmove(GIVEN,BQB2,BQ1,Val(qx,zb)))∧Val(qx,zb)=BB)⊃BOARD
(qx,QBUD)

*****simplify ↑↑;
160 (BOARD(qx,Unmkcapppmove(GIVEN,BQB2,BQ1,Val(qx,zb)))∧Val(qx,zb)=NB)⊃BOARD
(qx,QBUD)

*****VE UNDO4 qx,px,GIVEN,BQB2,BQ1,Val(qx,Taken Move px);
161 (SUCCESSOR(qx,px)∧(BOARD(px,GIVEN)∧(CAPPP Move px∧(Val(qx,Taken Move px)
=Val(qx,Taken Move px)∧(From Move px=BQB2∧To Move px=BQ1)))))⊃BOARD(qx,
Unmkcapppmove(GIVEN,BQB2,BQ1,Val(qx,Taken Move px)))

*****subst QBUDLBL-1 IN ↑;
162 (SUCCESSOR(qx,px)∧(BOARD(px,GIVEN)∧(CAPPP Move px∧(Val(qx,zb)=Val(qx,zb)
∧(From Move px=BQB2∧To Move px=BQ1)))))⊃BOARD(qx,Unmkcapppmove(GIVEN,BQB2,
BQ1,Val(qx,zb)))  (1 9 70)

*****label QX_QBUD;
*****tauteq BOARD(qx,QBUD)
*      CALL_PX,PXSUC,FROMPX,TOPX,CAPPPPX,NB_OR_BB,↑↑↑↑,↑↑↑,↑;
163 BOARD(qx,QBUD)  (1 9)
```

Section 4.2.4.3 The Fate of the Black Bishops

5.6. We have already determined that the captured piece was either a black bishop, or a black knight valued. However, we can infer other results from this fact.

5.6.1. We know that one of black's bishops, the BQB, is the black *on white* bishop. That is, that bishop never moves to a black square. But the capture square, BQ1, is a black square. Hence, the captured piece could not have been the BQB.

```
*****VE BishopsIsOnSameColor qx,BQB1,BQ1,BQB;
164 (Pos(P0,BQB1)=BQB∧Pos(qx,BQ1)=BQB)⊃(WHITESQUARES BQB1=WHITESQUARES BQ1)

*****simplify ↑;
165 ¬(Pos(qx,BQ1)=BQB)
```

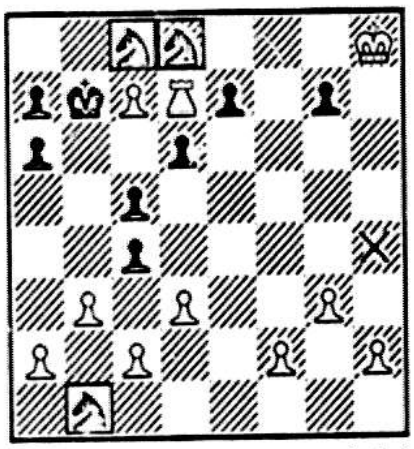
5.6.2. Nor could the captured piece have been the BKB. The black pawns on BK1 and BKN1 trap this bishop, preventing his moving until they have moved, and freed one of his exit squares. But we can see that these pawns are still on their original squares. Hence, the BKB can be on no square except *his* original square. (He is not, of course, on that square; rather, that bishop has been captured earlier in this game.)

```
*****VE Blocked_BKB qx,QBUD,BQ1;
166 (BOARD(qx,QBUD)∧(Valueon(QBUD,BK2)=PB∧(Valueon(QBUD,BKN2)=PB∧Pos(qx,BQ1)
=BKB)))⊃BQ1=BKB1

*****simplify ↑;
167 ¬(BOARD(qx,QBUD)∧Pos(qx,BQ1)=BKB)
```

5.7. However, any black bishop valued piece must be either the BQB, the BKB, or a promoted black pawn. Hence, if the captured piece was a black bishop, it must have been a promoted pawn.

```
*****VE MightBeBB qx,zb;
168 Val(qx,zb)=BB⊃((zb=BKB∨zb=BQB)∨(BPAWNS zb∧PROMOTEDPAWN(qx,zb)))

*****label IF_BISH;
*****tauteq Val(qx,zb)=BB⊃(BPAWNS zb∧PROMOTEDPAWN(qx,zb))
*      ON_ZB,QX_QBUD,↑↑↑↑,↑↑,↑;
169 Val(qx,zb)=BB⊃(BPAWNS zb∧PROMOTEDPAWN(qx,zb))  (1 9 70)
```

5.8. We know that QBUD is a board of qx. Hence, there must be black knights on both BQ1 and WQN1 on the total board of qx (or, equivalently, knight valued pieces on these squares in the position qx). If the captured piece was a black knight, then black had three knights on the board in qx. Having three black knights is proof of having a promoted pawn on one of those three squares. However, the conditions (and conclusions) of the lemma we really invoke are stronger. It states that if at most one black pawn has promoted, and the three black knight valued squares situation exists, then that pawn is on one of the three squares, and no other square (in particular, its not on the fallen square.) In this step, we're looking forward to phrase our conclusion in the form that will be most useful in the future.



If the captured piece were a knight, then black had three knights in QX.

figure 40

```
*****VE ThreeNB qx QBUD BKRP BQ1 BQB1 WQN1 WKR4;
170 Vt.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP)⊃(((¬(BQ1=BQB1)∧(¬(BQ1=WQN1)∧¬(
BQB1=WQN1)))∧((Val(qx,Pos(qx,BQ1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,BQ1)=NB))
∧((Val(qx,Pos(qx,BQB1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,BQB1)=NB))∧(Val(qx,
Pos(qx,WQN1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,WQN1)=NB)))))⊃(PROMOTEDPAWN(qx
,BKRP)∧((¬(BQ1=WKR4)∧(¬(BQB1=WKR4)∧¬(WQN1=WKR4)))⊃(¬(Pos(qx,WKR4)=BKRP)∧(¬(
Pos(qx,WKR4)=BKN)∧¬(Pos(qx,WKR4)=BQN))))))
```

We consult the simplification mechanism for several useful equalities and inequalities.

```
*****simplify ¬BQ1=BQB1∧¬BQ1=WQN1∧¬BQB1=WQN1∧Valueon(QBUD BQB1)=NB∧
*      Valueon(QBUD WQN1)=NB;
171 ¬(BQ1=BQB1)∧(¬(BQ1=WQN1)∧(¬(BQB1=WQN1)∧(Valueon(QBUD,BQB1)=NB∧Valueon(
QBUD,WQN1)=NB)))
```

A little renaming, and we get a useful result from our tautology decider.

```
*****substr ON_ZB IN ↑↑;
172 Vt.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP)⊃(((¬(BQ1=BQB1)∧(¬(BQ1=WQN1)∧¬(
BQB1=WQN1)))∧((Val(qx,zb)=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,BQ1)=NB))∧((Val(qx
,Pos(qx,BQB1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,BQB1)=NB))∧(Val(qx,Pos(qx,
WQN1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,WQN1)=NB)))))⊃(PROMOTEDPAWN(qx,BKRP)∧
((¬(BQ1=WKR4)∧(¬(BQB1=WKR4)∧¬(WQN1=WKR4)))⊃(¬(Pos(qx,WKR4)=BKRP)∧(¬(Pos(qx,
WKR4)=BKN)∧¬(Pos(qx,WKR4)=BQN))))))  (1 70)

*****label PROM_KNIGHT;
*****tauteq (Val(qx,zb)=NB∧Vt.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP))⊃
*(PROMOTEDPAWN(qx,BKRP)∧((¬BQ1=WKR4∧¬BQB1=WKR4∧¬WQN1=WKR4)⊃
*(¬Pos(qx,WKR4)=BKRP∧¬Pos(qx,WKR4)=BKN∧¬Pos(qx,WKR4)=BQN))) QX_QBUD,↑,↑↑;
173 (Val(qx,zb)=NB∧Vt.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP))⊃(PROMOTEDPAWN(
qx,BKRP)∧((¬(BQ1=WKR4)∧(¬(BQB1=WKR4)∧¬(WQN1=WKR4)))⊃(¬(Pos(qx,WKR4)=BKRP)∧(¬
(Pos(qx,WKR4)=BKN)∧¬(Pos(qx,WKR4)=BQN)))))  (1 9 70)
```

5.9. From the fact that the captured piece had either bishop or knight value (step 156), we

could now conclude that black has promoted one of his pawns. However, we defer that deduction for a few steps, until we can prove which black pawn it was that promoted. To do this, we need to examine the black pawn structure of QBUD.

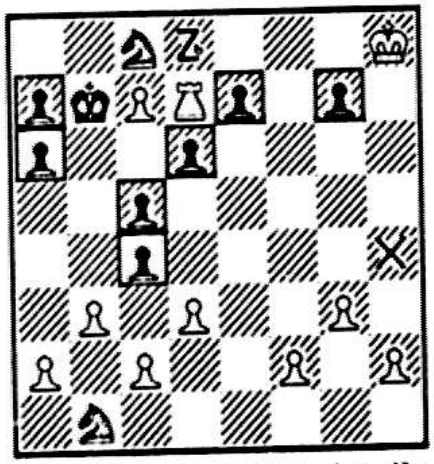
Section 4.2.5 The Black Pawns

6. Our attention turns towards identifying the black pawns on QBUD. We will be (almost) able to identify each of the pawn value pieces on that board.

6.1. We consider first the pawns in black's second row. These pawns have not moved, and are obviously the pawns that started on those squares. Of course, we have a lemma for this situation. It states that if a black pawn *value* is upon some square, and there was also a black pawn value upon that square in P0, the initial position, then it is the same *piece* is on that square as was upon it in P0. More concisely, certain black pawns have obviously not moved.

```
*****VE UnmovedBlackPawnThm qx,QBUD,BQRP,BQR2;
174 (Pos(P0,BQR2)=BQRP∧(Valueon(QBUD,BQR2)=PB∧BOARD(qx,QBUD)))⊃(Pos(P0,BQR2)
=Pos(qx,BQR2)∧Pospcf(qx,BQRP)=BQR2)

*****VE UnmovedBlackPawnThm qx,QBUD,BKP,BK2;
175 (Pos(P0,BK2)=BKP∧(Valueon(QBUD,BK2)=PB∧BOARD(qx,QBUD)))⊃(Pos(P0,BK2)=Pos
(qx,BK2)∧Pospcf(qx,BKP)=BK2)

*****VE UnmovedBlackPawnThm qx,QBUD,BKNP,BKN2;
176 (Pos(P0,BKN2)=BKNP∧(Valueon(QBUD,BKN2)=PB∧BOARD(qx,QBUD)))⊃(Pos(P0,BKN2)
=Pos(qx,BKN2)∧Pospcf(qx,BKNP)=BKN2)
```

We consult the simplification mechanism to verify our asserted arrangement.



Simplify can quickly and easily find the value on a square.

figure 41

```
******simplify Pos(P0,BQR2)=BQRP∧Valueon(QBUD,BQR2)=PB∧Pos(P0,BK2)=BKP∧
*       Valueon(QBUD,BK2)=PB∧Pos(P0,BKN2)=BKNP∧Valueon(QBUD,BKN2)=PB;
177 Pos(P0,BQR2)=BQRP∧(Valueon(QBUD,BQR2)=PB∧(Pos(P0,BK2)=BKP∧(Valueon(QBUD,
BK2)=PB∧(Pos(P0,BKN2)=BKNP∧Valueon(QBUD,BKN2)=PB))))
```

Hence, the black pawns on the second row squares are the BQRP, BKP and the BKNP.

```
*****label ROW2_BP;
***** tauteq ↑↑↑↑:#2∧↑↑↑:#2∧↑↑:#2 QX_QBUD,↑↑↑↑:↑;
178 (Pos(P0,BQR2)=Pos(qx,BQR2)∧Pospcf(qx,BQRP)=BQR2)∧((Pos(P0,BK2)=Pos(qx,
BK2)∧Pospcf(qx,BKP)=BK2)∧(Pos(P0,BKN2)=Pos(qx,BKN2)∧Pospcf(qx,BKNP)=BKN2))
(1 9)
```

6.2. The remaining deductions on the pawn structure are produced with the lemma *WhichBlackPawn*. This lemma employs the fact that, if a pawn is to move between two squares, the MAY_PAWN_CAPTURES predicate must be satisfied between those squares. Each of the eight black pawns is considered, resulting in a WFF which, when simplified, eliminates from consideration those pawns that could not be on the requested square. There are only two black pawns which can reach BQR3. We have shown that BQRP is on BQR2 in qx. Hence, the pawn on BQR3 must be the BQNP.

```
*****VE WhichBlackPawn qx,QBUD,BQR3;
179 (BOARD(qx,QBUD)∧Valueon(QBUD,BQR3)=PB)⊃((Pos(qx,BQR3)=BQRP∧(Pospcf(qx,
BQRP)=BQR3∧MAY_PAWN_CAPTURES(BQR2,BQR3,BLACK)))∨((Pos(qx,BQR3)=BQNP∧(Pospcf(
qx,BQNP)=BQR3∧MAY_PAWN_CAPTURES(BQN2,BQR3,BLACK)))∨((Pos(qx,BQR3)=BQBP∧(
Pospcf(qx,BQBP)=BQR3∧MAY_PAWN_CAPTURES(BQB2,BQR3,BLACK)))∨((Pos(qx,BQR3)=BQP
∧(Pospcf(qx,BQP)=BQR3∧MAY_PAWN_CAPTURES(BQ2,BQR3,BLACK)))∨((Pos(qx,BQR3)=BKP
∧(Pospcf(qx,BKP)=BQR3∧MAY_PAWN_CAPTURES(BK2,BQR3,BLACK)))∨((Pos(qx,BQR3)=
BKBP∧(Pospcf(qx,BKBP)=BQR3∧MAY_PAWN_CAPTURES(BKB2,BQR3,BLACK)))∨((Pos(qx,
BQR3)=BKNP∧(Pospcf(qx,BKNP)=BQR3∧MAY_PAWN_CAPTURES(BKN2,BQR3,BLACK)))∨(Pos(
qx,BQR3)=BKRP∧(Pospcf(qx,BKRP)=BQR3∧MAY_PAWN_CAPTURES(BKR2,BQR3,BLACK)))))))
)))

*****simplify ↑;
180 BOARD(qx,QBUD)⊃((Pos(qx,BQR3)=BQRP∧Pospcf(qx,BQRP)=BQR3)∨(Pos(qx,BQR3)=
BQNP∧Pospcf(qx,BQNP)=BQR3))

******simplify ¬(BQR2=BQR3);
181 ¬(BQR2=BQR3)

*****label ROW3R_BP;
*****tauteq Pos(qx,BQR3)=BQNP∧Pospcf(qx,BQNP)=BQR3 QX_QBUD,ROW2_BP,↑↑:↑;
182 Pos(qx,BQR3)=BQNP∧Pospcf(qx,BQNP)=BQR3  (1 9)
```

6.3. Of the remaining pawns, only the BQBP and BQP could reach BQB4 and BQ3. We have not established which of these pawns is on which square, but we can show that, between them, they fill these two locations.

```
*****VE WhichBlackPawn qx,QBUD,BQB4;
183 (BOARD(qx,QBUD)∧Valueon(QBUD,BQB4)=PB)⊃((Pos(qx,BQB4)=BQRP∧(Pospcf(qx,
BQRP)=BQB4∧MAY_PAWN_CAPTURES(BQR2,BQB4,BLACK)))∨((Pos(qx,BQB4)=BQNP∧(Pospcf(
qx,BQNP)=BQB4∧MAY_PAWN_CAPTURES(BQN2,BQB4,BLACK)))∨((Pos(qx,BQB4)=BQBP∧(
Pospcf(qx,BQBP)=BQB4∧MAY_PAWN_CAPTURES(BQB2,BQB4,BLACK)))∨((Pos(qx,BQB4)=BQP
∧(Pospcf(qx,BQP)=BQB4∧MAY_PAWN_CAPTURES(BQ2,BQB4,BLACK)))∨((Pos(qx,BQB4)=BKP
∧(Pospcf(qx,BKP)=BQB4∧MAY_PAWN_CAPTURES(BK2,BQB4,BLACK)))∨((Pos(qx,BQB4)=
BKBP∧(Pospcf(qx,BKBP)=BQB4∧MAY_PAWN_CAPTURES(BKB2,BQB4,BLACK)))∨((Pos(qx,
```

```
BQB4)=BKNP∧(Pospcf(qx,BKNP)=BQB4∧MAY_PAWN_CAPTURES(BKN2,BQB4,BLACK)))∨(Pos(
qx,BQB4)=BKRP∧(Pospcf(qx,BKRP)=BQB4∧MAY_PAWN_CAPTURES(BKR2,BQB4,BLACK)))))))
)))

*****VE WhichBlackPawn qx,QBUD,BQ3;
184 (BOARD(qx,QBUD)∧Valueon(QBUD,BQ3)=PB)⊃((Pos(qx,BQ3)=BQRP∧(Pospcf(qx,BQRP
)=BQ3∧MAY_PAWN_CAPTURES(BQR2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BQNP∧(Pospcf(qx,BQNP
)=BQ3∧MAY_PAWN_CAPTURES(BQN2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BQBP∧(Pospcf(qx,BQBP
)=BQ3∧MAY_PAWN_CAPTURES(BQB2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BQP∧(Pospcf(qx,BQP)=
BQ3∧MAY_PAWN_CAPTURES(BQ2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BKP∧(Pospcf(qx,BKP)=BQ3
∧MAY_PAWN_CAPTURES(BK2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BKBP∧(Pospcf(qx,BKBP)=BQ3∧
MAY_PAWN_CAPTURES(BKB2,BQ3,BLACK)))∨((Pos(qx,BQ3)=BKNP∧(Pospcf(qx,BKNP)=BQ3∧
MAY_PAWN_CAPTURES(BKN2,BQ3,BLACK)))∨(Pos(qx,BQ3)=BKRP∧(Pospcf(qx,BKRP)=BQ3∧
MAY_PAWN_CAPTURES(BKR2,BQ3,BLACK)))))))))

*****simplify ↑↑;
185 BOARD(qx,QBUD)⊃((Pos(qx,BQB4)=BQRP∧Pospcf(qx,BQRP)=BQB4)∨((Pos(qx,BQB4)=
BQNP∧Pospcf(qx,BQNP)=BQB4)∨((Pos(qx,BQB4)=BQBP∧Pospcf(qx,BQBP)=BQB4)∨((Pos(
qx,BQB4)=BQP∧Pospcf(qx,BQP)=BQB4)∨(Pos(qx,BQB4)=BKP∧Pospcf(qx,BKP)=BQB4)))))

*****simplify ↑↑;
186 BOARD(qx,QBUD)⊃((Pos(qx,BQ3)=BQBP∧Pospcf(qx,BQBP)=BQ3)∨((Pos(qx,BQ3)=BQP
∧Pospcf(qx,BQP)=BQ3)∨(Pos(qx,BQ3)=BKP∧Pospcf(qx,BKP)=BQ3)))

*****simplify ¬BQR2=BQB4∧¬BQR3=BQB4∧¬BK2=BQB4∧¬BQ3=BK2∧¬BQB4=BQ3;
187 ¬(BQR2=BQB4)∧(¬(BQR3=BQB4)∧(¬(BK2=BQB4)∧(¬(BQ3=BK2)∧¬(BQB4=BQ3))))

*****label QB_BP;
*****tauteq (Pos(qx,BQB4)=BQBP∧Pospcf(qx,BQBP)=BQB4∧
*      Pos(qx,BQ3)=BQP∧Pospcf(qx,BQP)=BQ3)∨
*        (Pos(qx,BQB4)=BQP∧Pospcf(qx,BQP)=BQB4∧
*      Pos(qx,BQ3)=BQBP∧Pospcf(qx,BQBP)=BQ3)
*     QX_QBUD,ROW2_BP,ROW3R_BP,↑↑↑;↑;
188 (Pos(qx,BQB4)=BQBP∧(Pospcf(qx,BQBP)=BQB4∧(Pos(qx,BQ3)=BQP∧Pospcf(qx,BQP)
=BQ3)))∨(Pos(qx,BQB4)=BQP∧(Pospcf(qx,BQP)=BQB4∧(Pos(qx,BQ3)=BQBP∧Pospcf(qx,
BQBP)=BQ3)))  (1 9)
```

6.4. This implies that the black pawn on BQB5 must be the BKBP.

```
*****VE WhichBlackPawn qx,QBUD,WQB4;
189 (BOARD(qx,QBUD)∧Valueon(QBUD,WQB4)=PB)⊃((Pos(qx,WQB4)=BQRP∧(Pospcf(qx,
BQRP)=WQB4∧MAY_PAWN_CAPTURES(BQR2,WQB4,BLACK)))∨((Pos(qx,WQB4)=BQNP∧(Pospcf(
qx,BQNP)=WQB4∧MAY_PAWN_CAPTURES(BQN2,WQB4,BLACK)))∨((Pos(qx,WQB4)=BQBP∧(
Pospcf(qx,BQBP)=WQB4∧MAY_PAWN_CAPTURES(BQB2,WQB4,BLACK)))∨((Pos(qx,WQB4)=BQP
∧(Pospcf(qx,BQP)=WQB4∧MAY_PAWN_CAPTURES(BQ2,WQB4,BLACK)))∨((Pos(qx,WQB4)=BKP
∧(Pospcf(qx,BKP)=WQB4∧MAY_PAWN_CAPTURES(BK2,WQB4,BLACK)))∨((Pos(qx,WQB4)=
BKBP∧(Pospcf(qx,BKBP)=WQB4∧MAY_PAWN_CAPTURES(BKB2,WQB4,BLACK)))∨((Pos(qx,
WQB4)=BKNP∧(Pospcf(qx,BKNP)=WQB4∧MAY_PAWN_CAPTURES(BKN2,WQB4,BLACK)))∨(Pos(
qx,WQB4)=BKRP∧(Pospcf(qx,BKRP)=WQB4∧MAY_PAWN_CAPTURES(BKR2,WQB4,BLACK)))))))
)))

*****simplify ↑;
190 BOARD(qx,QBUD)⊃((Pos(qx,WQB4)=BQRP∧Pospcf(qx,BQRP)=WQB4)∨((Pos(qx,WQB4)=
BQNP∧Pospcf(qx,BQNP)=WQB4)∨((Pos(qx,WQB4)=BQBP∧Pospcf(qx,BQBP)=WQB4)∨((Pos(
qx,WQB4)=BQP∧Pospcf(qx,BQP)=WQB4)∨((Pos(qx,WQB4)=BKP∧Pospcf(qx,BKP)=WQB4)∨(
Pos(qx,WQB4)=BKBP∧Pospcf(qx,BKBP)=WQB4))))))
```

```
*****simplify ¬BQR2=WQB4∧¬BQR3=WQB4∧¬BK2=WQB4∧¬BQ3=WQB4∧¬BQB4=WQB4;
191 ¬(BQR2=WQB4)∧(¬(BQR3=WQB4)∧(¬(BK2=WQB4)∧(¬(BQ3=WQB4)∧¬(BQB4=WQB4))))

*****label B5_BP;
*****tauteq Pos(qx,WQB4)=BKBP∧Pospcf(qx,BKBP)=WQB4
*      QX_QBUD,ROW2_BP,ROW3R_BP,QB_BP,↑↑;↑;
192 Pos(qx,WQB4)=BKBP∧Pospcf(qx,BKBP)=WQB4  (1 9)
```

Section 4.2.5.1 Which Pawn Promoted

6.5. Which pawn was the promoting pawn? A pawn that has promoted, no longer has pawn value on a board of that position (theorem ***BlackPawnValueSquares***) We consider each black pawn in turn as a possible candidate for having promoted.

The BQRP is, unpromoted, on BQR2.

```
*****label WHEREPROM;
*****VE BlackPawnValueSquares qx,QBUD,BQRP,BQR2;
193 ¬(PROMOTEDPAWN(qx,BQRP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQR2)=PB∧Pos(qx,
BQR2)=BQRP)))
```

Similarly the BQNP is on BQR3.

```
*****VE BlackPawnValueSquares qx,QBUD,BQNP,BQR3;
194 ¬(PROMOTEDPAWN(qx,BQNP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQR3)=PB∧Pos(qx,
BQR3)=BQNP)))
```

The BQBP and BQP split the squares BQB4 and BQ3 between them.

```
*****VE BlackPawnValueSquares qx,QBUD,BQBP,BQB4;
195 ¬(PROMOTEDPAWN(qx,BQBP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQB4)=PB∧Pos(qx,
BQB4)=BQBP)))

*****VE BlackPawnValueSquares qx,QBUD,BQBP,BQ3;
196 ¬(PROMOTEDPAWN(qx,BQBP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQ3)=PB∧Pos(qx,BQ3
)=BQBP)))

*****VE BlackPawnValueSquares qx,QBUD,BQP,BQB4;
197 ¬(PROMOTEDPAWN(qx,BQP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQB4)=PB∧Pos(qx,
BQB4)=BQP)))

*****VE BlackPawnValueSquares qx,QBUD,BQP,BQ3;
198 ¬(PROMOTEDPAWN(qx,BQP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BQ3)=PB∧Pos(qx,BQ3)
=BQP)))
```

The BKP occupies its original square, unpromoted.

```
*****VE BlackPawnValueSquares qx,QBUD,BKP,BK2;
199 ¬(PROMOTEDPAWN(qx,BKP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BK2)=PB∧Pos(qx,BK2)
=BKP)))
```

WQB4 is occupied by a pawn valued BKB.

```
*****VE BlackPawnValueSquares qx,QBUD,BKBP,WQB4;
```

```
200 ¬(PROMOTEDPAWN(qx,BKBP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,WQB4)=PB∧Pos(qx,
WQB4)=BKBP)))
```

And, similarly, the BKNP pawn sits on its original squares.

```
*****VE BlackPawnValueSquares qx,QBUD,BKNP,BKN2;
201 ¬(PROMOTEDPAWN(qx,BKNP)∧(BOARD(qx,QBUD)∧(Valueon(QBUD,BKN2)=PB∧Pos(qx,
BKN2)=BKNP)))
```

We confirm our expectations about the value of the occupants of these squares.

```
*****simplify Valueon(QBUD,BQR2)=PB∧ Valueon(QBUD,BQR3)=PB∧
*     Valueon(QBUD,BQB4)=PB∧ Valueon(QBUD,BQ3)=PB∧ Valueon(QBUD,BQB4)=PB∧
*     Valueon(QBUD,BQ3)=PB∧ Valueon(QBUD,BK2)=PB∧ Valueon(QBUD,WQB4)=PB∧
*     Valueon(QBUD,BKN2)=PB∧Pos(P0 BQR2)=BQRP∧Pos(P0 BK2)=BKP∧
*     Pos(P0 BKN2)=BKNP;
202 Valueon(QBUD,BQR2)=P3∧(Valueon(QBUD,BQR3)=PB∧(Valueon(QBUD,BQB4)=PB∧(
Valueon(QBUD,BQ3)=PB∧(Valueon(QBUD,BQB4)=PB∧(Valueon(QBUD,BQ3)=PB∧(Valueon(
QBUD,BK2)=PB∧(Valueon(QBUD,WQB4)=PB∧(Valueon(QBUD,BKN2)=PB∧(Pos(P0,BQR2)=
BQRP∧(Pos(P0,BK2)=BKP∧Pos(P0,BKN2)=BKNP)))))))))
```

Now, there are eight black pawns.

```
*****VE BlackPawnsAre_ t;
203 (t=BKP∨(t=BQP∨(t=BKNP∨(t=BKBP∨(t=BKRP∨(t=BQBP∨(t=BQNP∨t=BQRP)))))))=
BPAWNS t
```

Hence, if one of them has promoted, it must be the BKRP. We generalize this WFF to all possible black promotions in qx.

```
*****tauteq(BPAWNS t∧PROMOTEDPAWN(qx t))⊃t=BKRP
*WHEREPROM:↑,QX_QBUD,ROW2_BP,ROW3R_BP,QB_BP,B5_BP;
204 (BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP  (1 9)

*****label THE_ONLY_ONE;
*****VI↑t;
205 ∀t.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP)  (1 9)
```

So it zb was a promoted pawn, it must have been the BKRP.

```
*****VE THE_ONLY_ONE zb;
206 (BPAWNS zb∧PROMOTEDPAWN(qx,zb))⊃zb=BKRP  (1 9)
```

But, as we pointed out before, we have established sufficient conditions to prove that a black pawn has promoted. Since a black pawn has promoted, and the only black pawn that could have promoted is the BKRP (line 205), then BKRP has promoted.

```
*****label PROM_BKRP;
*****tauteq PROMOTEDPAWN(qx BKRP)
*     NB_OR_BB,IF_BISH,↑,PROM_KNIGHT,THE_ONLY_ONE;
207 PROMOTEDPAWN(qx,BKRP)  (1 9)
```

Section 4.2.6 Did a Black Piece Fall?

7. Did a black piece fall from the board? We consider each black piece, in turn, to show that it could not have been the fallen piece. But first, we pause to point out that, as the square WKR4 was not involved in the last move, its occupant was identical in both px and qx. Axiomatically, a square not source, destination, or special square of a special move, retains the same contents from position to position.

```
*****VE MCONSEQD qx px WKR4;
208 (SUCCESSOR(qx,px)∧(¬(WKR4=From Move px)∧(¬(WKR4=To Move px)∧(¬(CASTLE
Move px∧(WKR4=Alsofrom Move px∨WKR4=Alsoto Move px))∧¬(ENPASSANT Move px∧
WKR4=Takenon Move px)))))⊃Pos(qx,WKR4)=Pos(px,WKR4)
```

We know the source (From square) and destination (To square) of the Move px. Neither of them is WKR4. We also know that Move px was neither an *en passant* capture, nor a castle.

```
*****simplify ¬WKR4=BQ1 ∧ ¬ WKR4 = BQB2;
209 ¬(WKR4=BQ1)∧¬(WKR4=BQB2)

*****VE MconseqmX qx px;
210 SUCCESSOR(qx,px)⊃((CASTLE Move px=CASTLING(qx,px))∧((ENPASSANT Move px=
EN_PASSANT(qx,px))∧(ORDINARY Move px=SIMPLELEGALMOVE(qx,px))))
```

Hence, the fallen piece was also on WKR4 in qx.

```
*****label SAME_ON_WKR4;
***** tauteq Pos(qx WKR4)=Pos(px WKR4)
*      ↑↑↑;↑,PXSUC,FROMPX,TOPX,NOTPXCASTLE,NOTPXEP;
211 Pos(qx,WKR4)=Pos(px,WKR4)  (1 9)
```

We return to consideration of the each of the black pieces as a candidate fallen piece. We established that the promoted pawn, BKRP, was on one of the squares BQ1, BQB2 or WQN2 in qx. Hence, the fallen piece was not the BKRP.

```
*****label ON_BLACK_SQS;
*****simplify ¬BQ1=WKR4∧ ¬BQB1=WKR4∧ ¬BQR2=WKR4∧ ¬BQN2=WKR4∧
*      ¬BK2=WKR4∧ ¬BKN2=WKR4∧ ¬BQR3=WKR4∧ ¬BQ3=WKR4∧ ¬BQB4=WKR4∧
*      ¬WQB4=WKR4∧ ¬WQN1=WKR4;
212 ¬(BQ1=WKR4)∧(¬(BQB1=WKR4)∧(¬(BQR2=WKR4)∧(¬(BQN2=WKR4)∧(¬(BK2=WKR4)∧(¬(
BKN2=WKR4)∧(¬(BQR3=WKR4)∧(¬(BQ3=WKR4)∧(¬(BQB4=WKR4)∧(¬(WQB4=WKR4)∧¬(WQN1=
WKR4)))))))))

*****VE Unique qx,BQ1,WKR4,BKRP;
213 Pos(qx,BQ1)=BKRP⊃(Pos(qx,WKR4)=BKRP=BQ1=WKR4)

*****tauteq ¬(Pos(qx,WKR4)=BKRP)      ON_ZB,NB_OR_BB,
*      IF_BISH,PROM_KNIGHT,THE_ONLY_ONE:206,ON_BLACK_SQS:213;
214 ¬(Pos(qx,WKR4)=BKRP)  (1 9)
```

We know squares for each of the seven unpromoted black pawns. None of these squares is WKR4.

```
*****VE Unique qx,BQR2,WKR4,BQRP;
215 Pos(qx,BQR2)=BQRP⊃(Pos(qx,WKR4)=BQRP=BQR2=WKR4)
```

```
*****VE Unique qx,BK2,WKR4,BKP;
216 Pos(qx,BK2)=BKP⊃(Pos(qx,WKR4)=BKP∨BK2=WKR4)

*****VE Unique qx,BKN2,WKR4,BKNP;
217 Pos(qx,BKN2)=BKNP⊃(Pos(qx,WKR4)=BKNP∨BKN2=WKR4)

*****VE Unique qx,BQR3,WKR4,BQNP;
218 Pos(qx,BQR3)=BQNP⊃(Pos(qx,WKR4)=BQNP∨BQR3=WKR4)

*****VE Unique qx,BQB4,WKR4,BQBP;
219 Pos(qx,BQB4)=BQBP⊃(Pos(qx,WKR4)=BQBP∨BQB4=WKR4)

*****VE Unique qx,BQ3,WKR4,BQBP;
220 Pos(qx,BQ3)=BQBP⊃(Pos(qx,WKR4)=BQBP∨BQ3=WKR4)

*****VE Unique qx,BQB4,WKR4,BQP;
221 Pos(qx,BQB4)=BQP⊃(Pos(qx,WKR4)=BQP∨BQB4=WKR4)

*****VE Unique qx,BQ3,WKR4,BQP;
222 Pos(qx,BQ3)=BQP⊃(Pos(qx,WKR4)=BQP∨BQ3=WKR4)

*****VE Unique qx,WQB4,WKR4,BKBP;
223 Pos(qx,WQB4)=BKBP⊃(Pos(qx,WKR4)=BKBP∨WQB4=WKR4)
```

7.1. This accounts for all of the black pawns. Hence, the fallen piece was not a black pawn.

```
*****VE BlackPawnsAre_ Pos(qx WKR4);
224 (Pos(qx,WKR4)=BKP∨(Pos(qx,WKR4)=BQP∨(Pos(qx,WKR4)=BKNP∨(Pos(qx,WKR4)=
BKBP∨(Pos(qx,WKR4)=BKRP∨(Pos(qx,WKR4)=BQBP∨(Pos(qx,WKR4)=BQNP∨Pos(qx,WKR4)=
BQRP))))))=BPAWNS Pos(qx,WKR4)

*****label NOT_BP;
*****tauteq ¬↑:#2 212:↑,ROW2_BP,ROW3R_BP,QB_BP,B5_BP,ROW2_BP-1;
225 ¬BPAWNS Pos(qx,WKR4)  (1 9)
```

7.2. If the fallen piece were a black knight, we would suffer from a surfeit of black knights. We use the same lemma as before, *ThreeNB* in this demonstration.

```
*****VE ThreeNB qx,QBUD,BKRP,BQB1,WQN1,WKR4,BQ1;
226 ∀t.((BPAWNS t∧PROMOTEDPAWN(qx,t))⊃t=BKRP)⊃(((¬(BQB1=WQN1)∧(¬(BQB1=WKR4)∧
¬(WQN1=WKR4)))∧((Val(qx,Pos(qx,BQB1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,BQB1)=
NB))∧((Val(qx,Pos(qx,WQN1))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,WQN1)=NB))∧(Val(
qx,Pos(qx,WKR4))=NB∨(BOARD(qx,QBUD)∧Valueon(QBUD,WKR4)=NB)))))⊃(PROMOTEDPAWN
(qx,BKRP)∧((¬(BQB1=BQ1)∧(¬(WQN1=BQ1)∧¬(WKR4=BQ1)))⊃(¬(Pos(qx,BQ1)=BKRP)∧(¬(
Pos(qx,BQ1)=BKN)∧¬(Pos(qx,BQ1)=BQN))))))
```

If a black knight fell, it would have had NB value (knights do not promote).

```
*****VE OfficerValueThmX qx,BKN,Pos(qx,WKR4);
227 (¬PAWNS BKN∧BKN=Pos(qx,WKR4))⊃Val(P0,BKN)=Val(qx,Pos(qx,WKR4))

*****VE OfficerValueThmX qx,BQN,Pos(qx,WKR4);
228 (¬PAWNS BQN∧BQN=Pos(qx,WKR4))⊃Val(P0,BQN)=Val(qx,Pos(qx,WKR4))

******simplify ¬PAWNS BQN∧(Val(P0,BQN)=NB∧(¬PAWNS BKN∧Val(P0,BKN)=NB));
```

```
229 ¬PAWNS BQN∧(Val(P0,BQN)=NB∧(¬PAWNS BKN∧Val(P0,BKN)=NB))
```

Hence, the fallen piece was not a black knight.

```
*****label NOT_NB;
***** tauteq ¬(Pos(qx,WKR4)=BKN)∧¬(Pos(qx,WKR4)=BQN)
*     ON_ZB,NB_OR_BB,QX_QBUD,IF_BISH,PROM_KNIGHT-2,PROM_KNIGHT,
*     THE_ONLY_ONE,THE_ONLY_ONE+1,ON_BLACK_SQS,ON_BLACK_SQS+1,↑↑↑↑;↑;
230 ¬(Pos(qx,WKR4)=BKN)∧¬(Pos(qx,WKR4)=BQN)  (1 9)
```

7.3. The black on white bishop could not have fallen from the black square WKR4.

```
*****VE BishopsIsOnSameColor qx BQB1 WKR4 BQB;
231 (Pos(P0,BQB1)=BQB∧Pos(qx,WKR4)=BQB)⊃(WHITESQUARES BQB1=WHITESQUARES WKR4
)

*****label NOT_BQB;
***** simplify ↑;
232 ¬(Pos(qx,WKR4)=BQB)
```

7.4. The black on black bishop, as we have already asserted, did not escape from his starting square.

```
*****VE Blocked_BKB qx QBUD WKR4;
233 (BOARD(qx,QBUD)∧(Valueon(QBUD,BK2)=PB∧(Valueon(QBUD,BKN2)=PB∧Pos(qx,WKR4
)=BKB)))⊃WKR4=BKB1

*****label NOT_BKB;
***** simplify ↑;
234 ¬(BOARD(qx,QBUD)∧Pos(qx,WKR4)=BKB)
```

7.5. The black king is on BQN2, not WKR4.

```
*****VE Unique px BQN2 WKR4 BK;
235 Pos(px,BQN2)=BK⊃(Pos(px,WKR4)=BK=BQN2=WKR4)

*****label NOT_BK;
***** tauteq ¬Pos(qx WKR4)=BK ↑,212,5,1,211;
236 ¬(Pos(qx,WKR4)=BK)  (1 9)
```

7.6. If a black rook or black queen value were on the square WKR4 in px then white would be in check in px. Note the employment of a single simplification to observe this check on the constructed board.

```
*****label BQ_OR_BR;
*****simplify WHITEINCHECK(Makeboard(GIVEN WKR4 QB))∧
*                    WHITEINCHECK(Makeboard(GIVEN WKR4 RB));
237 WHITEINCHECK Makeboard(GIVEN,WKR4,QB)∧WHITEINCHECK Makeboard(GIVEN,WKR4,
RB)

*****VE SUB_BOARDS1 px GIVEN WKR4 Pos(px WKR4) QB;
238 (Val(px,Pos(px,WKR4))=QB∧(Pos(px,WKR4)=Pos(px,WKR4)∧BOARD(px,GIVEN)))⊃
BOARD(px,Makeboard(GIVEN,WKR4,QB))
```

```
*****VE SUB_BOARDS1 px GIVEN WKR4 Pos(px WKR4) RB;
239 (Val(px,Pos(px,WKR4))=RB∧(Pos(px,WKR4)=Pos(px,WKR4)∧BOARD(px,GIVEN)))⊃
BOARD(px,Makeboard(GIVEN,WKR4,RB))
```

And if white were in check, it would be white's turn in px.

```
*****VE AlternateWhite px ↑↑:#2#2;
240 (BOARD(px,Makeboard(GIVEN,WKR4,QB))∧WHITEINCHECK Makeboard(GIVEN,WKR4,QB
))⊃(POSITIONINCHECK(px,WHITE)∧WHITETURN px)

*****VE AlternateWhite px ↑↑:#2#2;
241 (BOARD(px,Makeboard(GIVEN,WKR4,RB))∧WHITEINCHECK Makeboard(GIVEN,WKR4,RB
))⊃(POSITIONINCHECK(px,WHITE)∧WHITETURN px)
```

If the fallen piece would be rook or queen valued if a rook or queen had fallen.

```
*****VE OfficerValueThmX px BQ Pos(px WKR4);
242 (¬PAWNS BQ∧BQ=Pos(px,WKR4))⊃Val(P0,BQ)=Val(px,Pos(px,WKR4))

*****VE OfficerValueThmX px BQR Pos(px WKR4);
243 (¬PAWNS BQR∧BQR=Pos(px,WKR4))⊃Val(P0,BQR)=Val(px,Pos(px,WKR4))

*****VE OfficerValueThmX px BKR Pos(px WKR4);
244 (¬PAWNS BKR∧BKR=Pos(px,WKR4))⊃Val(P0,BKR)=Val(px,Pos(px,WKR4))

*****simplify ¬PAWNS BQ∧¬PAWNS BQR∧¬PAWNS BKR∧
*      Val(P0 BQ)=QB∧Val(P0 BQR)=RB∧Val(P0 BKR)=RB;
245 ¬PAWNS BQ∧(¬PAWNS BQR∧(¬PAWNS BKR∧(Val(P0,BQ)=QB∧(Val(P0,BQR)=RB∧Val(P0,
BKR)=RB))))
```

But we have already determined (back in steps 1, 2 and 3), that black is in check in px, and its his turn to move. If white were also in check, we would have a contradiction. Hence, the fallen piece could not have been a black rook or queen.

```
*****tauteq ¬Pos(qx WKR4)=BQ∧¬Pos(qx WKR4)=BQR∧¬Pos(qx WKR4)=BKR
*      BQ_OR_BR:↑,SAME_ON_WKR4,CALL_PX,BINCHECK,BLACK_GOES;
246 ¬(Pos(qx,WKR4)=BQ)∧(¬(Pos(qx,WKR4)=BQR)∧¬(Pos(qx,WKR4)=BKR))  (1 9)
```

8. We have considered each of the black pieces. None of them could have fallen from the board. Hence, the fallen piece was not a BLACKPIECE.

```
*****VE BlackpieceArePawnsOr_ Pos(qx WKR4);
247 BLACKPIECE Pos(qx,WKR4)≡(BPAWNS Pos(qx,WKR4)∨(Pos(qx,WKR4)=BK∨(Pos(qx,
WKR4)=BQ∨(Pos(qx,WKR4)=BKN∨(Pos(qx,WKR4)=BKB∨(Pos(qx,WKR4)=BKR∨(Pos(qx,WKR4)
=BQB∨(Pos(qx,WKR4)=BQN∨Pos(qx,WKR4)=BQR)))))))

*****label NOT_B;
*****tauteq ¬↑:#1 ↑↑:↑,QX_QBUD, NOT_BP,NOT_BK,NOT_BN,NOT_BQB,NOT_BKB;
248 ¬BLACKPIECE Pos(qx,WKR4)  (1 9)
```

But all chesspieces are either black or white. We know (from our original assumption) that some chesspiece did fall. Hence, it must have been a white piece.

```
*****VE BorW_Piece_ Pos(qx,WKR4);
```

```
249 CHESSPIECES Pos(qx,WKR4)⊃¬(BLACKPIECE Pos(qx,WKR4)=WHITEPIECE Pos(qx,
WKR4))
```

Let us call that piece yyw.

```
*****∃I SAME_ON_WKR4 Pos(px,WKR4)←yyw;
250 WHITEPIECE Pos(qx,WKR4)⊃∃yyw.Pos(qx,WKR4)=yyw

*****tauteq ∃yyw.Pos(qx,WKR4)=yyw CALL_PX,SAME_ON_WKR4,NOT_B;↑;
251 ∃yyw.Pos(qx,WKR4)=yyw  (1 9)

*****label CALL_YYW;
***** ∃E ↑ yyw;
252 Pos(qx,WKR4)=yyw  (252)
```

Section 4.2.7 The Fallen Piece Wasn't a White Pawn

9. By a process similar to that employed for the black pawns, we can identify the locations of each of the white pawns.

There are four white pawns on white's second row.

```
*****∀E UnmovedWhitePawnThm qx,QBUD,WQRP,WQR2;
253 (Pos(P0,WQR2)=WQRP∧(Valueon(QBUD,WQR2)=PW∧BOARD(qx,QBUD)))⊃(Pos(P0,WQR2)
=Pos(qx,WQR2)∧Pospcf(qx,WQRP)=WQR2)

*****∀E UnmovedWhitePawnThm qx,QBUD,WQBP,WQB2;
254 (Pos(P0,WQB2)=WQBP∧(Valueon(QBUD,WQB2)=PW∧BOARD(qx,QBUD)))⊃(Pos(P0,WQB2)
=Pos(qx,WQB2)∧Pospcf(qx,WQBP)=WQB2)

*****∀E UnmovedWhitePawnThm qx,QBUD,WKBP,WKB2;
255 (Pos(P0,WKB2)=WKBP∧(Valueon(QBUD,WKB2)=PW∧BOARD(qx,QBUD)))⊃(Pos(P0,WKB2)
=Pos(qx,WKB2)∧Pospcf(qx,WKBP)=WKB2)

*****∀E UnmovedWhitePawnThm qx,QBUD,WKRP,WKR2;
256 (Pos(P0,WKR2)=WKRP∧(Valueon(QBUD,WKR2)=PW∧BOARD(qx,QBUD)))⊃(Pos(P0,WKR2)
=Pos(qx,WKR2)∧Pospcf(qx,WKRP)=WKR2)

*****simplify (Pos(P0,WQR2)=WQRP∧Valueon(QBUD,WQR2)=PW)∧
*     (Pos(P0,WQB2)=WQBP∧Valueon(QBUD,WQB2)=PW)∧
*     (Pos(P0,WKB2)=WKBP∧Valueon(QBUD,WKB2)=PW)∧
*     (Pos(P0,WKR2)=WKRP∧Valueon(QBUD,WKR2)=PW);
257 (Pos(P0,WQR2)=WQRP∧Valueon(QBUD,WQR2)=PW)∧((Pos(P0,WQB2)=WQBP∧Valueon(
QBUD,WQB2)=PW)∧((Pos(P0,WKB2)=WKBP∧Valueon(QBUD,WKB2)=PW)∧(Pos(P0,WKR2)=WKRP
∧Valueon(QBUD,WKR2)=PW)))

*****label ROW2_WP;
***** tauteq (Pos(qx WQR2)=WQRP∧Pospcf(qx WQRP)=WQR2)∧
*     (Pos(qx WQB2)=WQBP∧Pospcf(qx WQBP)=WQB2)∧
*     (Pos(qx WKB2)=WKBP∧Pospcf(qx WKBP)=WKB2)∧
*     (Pos(qx WKR2)=WKRP∧Pospcf(qx WKRP)=WKR2) ↑↑↑↑↑;↑,QX_QBUD;
258 (Pos(qx,WQR2)=WQRP∧Pospcf(qx,WQRP)=WQR2)∧((Pos(qx,WQB2)=WQBP∧Pospcf(qx,
WQBP)=WQB2)∧((Pos(qx,WKB2)=WKBP∧Pospcf(qx,WKBP)=WKB2)∧(Pos(qx,WKR2)=WKRP∧
Pospcf(qx,WKRP)=WKR2)))  (1 9)
```

The white pawns on WQN3 and WKN3 are therefore the WQNP and WKNP, respectively.

```
*****VE WhichWhitePawn qx QBUD WQN3;
259 (BOARD(qx,QBUD)∧Valueon(QBUD,WQN3)=PW)⊃((Pos(qx,WQN3)=WQRP∧(Pospcf(qx,
WQRP)=WQN3∧MAY_PAWN_CAPTURES(WQR2,WQN3,WHITE)))∨((Pos(qx,WQN3)=WQNP∧(Pospcf(
qx,WQNP)=WQN3∧MAY_PAWN_CAPTURES(WQN2,WQN3,WHITE)))∨((Pos(qx,WQN3)=WQBP∧(
Pospcf(qx,WQBP)=WQN3∧MAY_PAWN_CAPTURES(WQB2,WQN3,WHITE)))∨((Pos(qx,WQN3)=WQP
∧(Pospcf(qx,WQP)=WQN3∧MAY_PAWN_CAPTURES(WQ2,WQN3,WHITE)))∨((Pos(qx,WQN3)=WKP
∧(Pospcf(qx,WKP)=WQN3∧MAY_PAWN_CAPTURES(WK2,WQN3,WHITE)))∨((Pos(qx,WQN3)=
WKBP∧(Pospcf(qx,WKBP)=WQN3∧MAY_PAWN_CAPTURES(WKB2,WQN3,WHITE)))∨((Pos(qx,
WQN3)=WKNP∧(Pospcf(qx,WKNP)=WQN3∧MAY_PAWN_CAPTURES(WKN2,WQN3,WHITE)))∨(Pos(
qx,WQN3)=WKRP∧(Pospcf(qx,WKRP)=WQN3∧MAY_PAWN_CAPTURES(WKR2,WQN3,WHITE)))))))
)))

*****VE WhichWhitePawn qx QBUD WKN3;
260 (BOARD(qx,QBUD)∧Valueon(QBUD,WKN3)=PW)⊃((Pos(qx,WKN3)=WQRP∧(Pospcf(qx,
WQRP)=WKN3∧MAY_PAWN_CAPTURES(WQR2,WKN3,WHITE)))∨((Pos(qx,WKN3)=WQNP∧(Pospcf(
qx,WQNP)=WKN3∧MAY_PAWN_CAPTURES(WQN2,WKN3,WHITE)))∨((Pos(qx,WKN3)=WQBP∧(
Pospcf(qx,WQBP)=WKN3∧MAY_PAWN_CAPTURES(WQB2,WKN3,WHITE)))∨((Pos(qx,WKN3)=WQP
∧(Pospcf(qx,WQP)=WKN3∧MAY_PAWN_CAPTURES(WQ2,WKN3,WHITE)))∨((Pos(qx,WKN3)=WKP
∧(Pospcf(qx,WKP)=WKN3∧MAY_PAWN_CAPTURES(WK2,WKN3,WHITE)))∨((Pos(qx,WKN3)=
WKBP∧(Pospcf(qx,WKBP)=WKN3∧MAY_PAWN_CAPTURES(WKB2,WKN3,WHITE)))∨((Pos(qx,
WKN3)=WKNP∧(Pospcf(qx,WKNP)=WKN3∧MAY_PAWN_CAPTURES(WKN2,WKN3,WHITE)))∨(Pos(
qx,WKN3)=WKRP∧(Pospcf(qx,WKRP)=WKN3∧MAY_PAWN_CAPTURES(WKR2,WKN3,WHITE)))))))
)))

*****simplify ↑↑;
261 BOARD(qx,QBUD)⊃((Pos(qx,WQN3)=WQRP∧Pospcf(qx,WQRP)=WQN3)∨((Pos(qx,WQN3)=
WQNP∧Pospcf(qx,WQNP)=WQN3)∨(Pos(qx,WQN3)=WQBP∧Pospcf(qx,WQBP)=WQN3)))

*****simplify ↑↑;
262 BOARD(qx,QBUD)⊃((Pos(qx,WKN3)=WKBP∧Pospcf(qx,WKBP)=WKN3)∨((Pos(qx,WKN3)=
WKNP∧Pospcf(qx,WKNP)=WKN3)∨(Pos(qx,WKN3)=WKRP∧Pospcf(qx,WKRP)=WKN3)))

*****simplify ¬WQR2=WQN3∧¬WQB2=WQN3∧¬WKR2=WKN3∧¬WKB2=WKN3;
263 ¬(WQR2=WQN3)∧(¬(WQB2=WQN3)∧(¬(WKR2=WKN3)∧¬(WKB2=WKN3)))

*****label ROW3_WP;
*****tauteq ↑↑↑:#2#2#1∧↑↑:#2#2#1 ↑↑↑:↑,ROW2_WP,QX_QBUD;
264 (Pos(qx,WQN3)=WQNP∧Pospcf(qx,WQNP)=WQN3)∧(Pos(qx,WKN3)=WKNP∧Pospcf(qx,
WKNP)=WKN3)  (1 9)
```

Therefore, the white pawns on BQB2 (in qx) and WQ3 are the WKP and WQP (though we don't know which is which).

```
*****VE WhichWhitePawn qx QBUD WQ3;
265 (BOARD(qx,QBUD)∧Valueon(QBUD,WQ3)=PW)⊃((Pos(qx,WQ3)=WQRP∧(Pospcf(qx,WQRP
)=WQ3∧MAY_PAWN_CAPTURES(WQR2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WQNP∧(Pospcf(qx,WQNP
)=WQ3∧MAY_PAWN_CAPTURES(WQN2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WQBP∧(Pospcf(qx,WQBP
)=WQ3∧MAY_PAWN_CAPTURES(WQB2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WQP∧(Pospcf(qx,WQP)=
WQ3∧MAY_PAWN_CAPTURES(WQ2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WKP∧(Pospcf(qx,WKP)=WQ3
∧MAY_PAWN_CAPTURES(WK2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WKBP∧(Pospcf(qx,WKBP)=WQ3∧
MAY_PAWN_CAPTURES(WKB2,WQ3,WHITE)))∨((Pos(qx,WQ3)=WKNP∧(Pospcf(qx,WKNP)=WQ3∧
MAY_PAWN_CAPTURES(WKN2,WQ3,WHITE)))∨(Pos(qx,WQ3)=WKRP∧(Pospcf(qx,WKRP)=WQ3∧
MAY_PAWN_CAPTURES(WKR2,WQ3,WHITE)))))))))
```

```
*****VE WhichWhitePawn qx QBUD BQB2;
266 (BOARD(qx,QBUD)∧Valueon(QBUD,BQB2)=PW)⊃((Pos(qx,BQB2)=WQRP∧(Pospcf(qx,
WQRP)=BQB2∧MAY_PAWN_CAPTURES(WQR2,BQB2,WHITE)))∨((Pos(qx,BQB2)=WQNP∧(Pospcf(
qx,WQNP)=BQB2∧MAY_PAWN_CAPTURES(WQN2,BQB2,WHITE)))∨((Pos(qx,BQB2)=WQBP∧(
Pospcf(qx,WQBP)=BQB2∧MAY_PAWN_CAPTURES(WQB2,BQB2,WHITE)))∨((Pos(qx,BQB2)=WQP
∧(Pospcf(qx,WQP)=BQB2∧MAY_PAWN_CAPTURES(WQ2,BQB2,WHITE)))∨((Pos(qx,BQB2)=WKP
∧(Pospcf(qx,WKP)=BQB2∧MAY_PAWN_CAPTURES(WK2,BQB2,WHITE)))∨((Pos(qx,BQB2)=
WKBP∧(Pospcf(qx,WKBP)=BQB2∧MAY_PAWN_CAPTURES(WKB2,BQB2,WHITE)))∨((Pos(qx,
BQB2)=WKNP∧(Pospcf(qx,WKNP)=BQB2∧MAY_PAWN_CAPTURES(WKN2,BQB2,WHITE)))∨(Pos(
qx,BQB2)=WKRP∧(Pospcf(qx,WKRP)=BQB2∧MAY_PAWN_CAPTURES(WKR2,BQB2,WHITE))))))))
)))

*****simplify↑↑;
267 BOARD(qx,QBUD)⊃((Pos(qx,WQ3)=WQBP∧Pospcf(qx,WQBP)=WQ3)∨((Pos(qx,WQ3)=WQP
∧Pospcf(qx,WQP)=WQ3)∨(Pos(qx,WQ3)=WKP∧Pospcf(qx,WKP)=WQ3)))

*****simplify ¬WQB2=WQ3∧Valueon(QBUD BQB2)=PW∧ ¬WQR2=BQB2∧ ¬WQB2=BQB2∧
*¬WKB2=BQB2∧ ¬WKR2=BQB2∧ ¬WQN3=BQB2∧ ¬WKN3=BQB2∧ ¬WQ3=BQB2;
268 ¬(WQB2=WQ3)∧(Valueon(QBUD,BQB2)=PW∧(¬(WQR2=BQB2)∧(¬(WQB2=BQB2)∧(¬(WKB2=
BQB2)∧(¬(WKR2=BQB2)∧(¬(WQN3=BQB2)∧(¬(WKN3=BQB2)∧¬(WQ3=BQB2))))))))

*****label ROYAL_WP;
***** tauteq
*     ((Pos(qx WQ3)=WQP∧Pospcf(qx WQP)=WQ3)∧
*      (Pos(qx BQB2)=WKP∧Pospcf(qx WKP)=BQB2))∨
*     ((Pos(qx WQ3)=WKP∧Pospcf(qx WKP)=WQ3)∧
*      (Pos(qx BQB2)=WQP∧Pospcf(qx WQP)=BQB2))
*  ↑↑↑:↑,ROW2_WP,ROW3_WP,QX_QBUD;
269 ((Pos(qx,WQ3)=WQP∧Pospcf(qx,WQP)=WQ3)∧(Pos(qx,BQB2)=WKP∧Pospcf(qx,WKP)=
BQB2))∨((Pos(qx,WQ3)=WKP∧Pospcf(qx,WKP)=WQ3)∧(Pos(qx,BQB2)=WQP∧Pospcf(qx,WQP
)=BQB2))  (1 9)
```

Hence, any square in qx which is not one of these squares, does not have a white pawn on it. Similarly, no white pawn has been captured in the game that reached qx.

```
*****VE WhereWhitePawns p qx x sq WQR2 WQN3 WQB2 WQ3 BQB2 WKB2 WKN3 WKR2;
270 (Pos(qx,WQR2)=WQRP∧(Pos(qx,WQN3)=WQNP∧(Pos(qx,WQB2)=WQBP∧(Pos(qx,WQ3)=
WQP∧(Pos(qx,BQB2)=WKP∧(Pos(qx,WKB2)=WKBP∧(Pos(qx,WKN3)=WKNP∧Pos(qx,WKR2)=
WKRP)))))))⊃(((¬(sq=WQR2)∧(¬(sq=WQN3)∧(¬(sq=WQB2)∧(¬(sq=WQ3)∧(¬(sq=BQB2)∧(¬(
sq=WKB2)∧(¬(sq=WKN3)∧¬(sq=WKR2)))))))⊃¬WPAWNS Pos(qx,sq))∧((x=Taken Move p∧
(PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x))

*****VE WhereWhitePawns p qx x sq WQR2 WQN3 WQB2  BQB2 WQ3 WKB2 WKN3 WKR2;
271 (Pos(qx,WQR2)=WQRP∧(Pos(qx,WQN3)=WQNP∧(Pos(qx,WQB2)=WQBP∧(Pos(qx,BQB2)=
WQP∧(Pos(qx,WQ3)=WKP∧(Pos(qx,WKB2)=WKBP∧(Pos(qx,WKN3)=WKNP∧Pos(qx,WKR2)=WKRP
)))))))⊃(((¬(sq=WQR2)∧(¬(sq=WQN3)∧(¬(sq=WQB2)∧(¬(sq=BQB2)∧(¬(sq=WQ3)∧(¬(sq=
WKB2)∧(¬(sq=WKN3)∧¬(sq=WKR2)))))))⊃¬WPAWNS Pos(qx,sq))∧((x=Taken Move p∧(
PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x))

*****taut ↑:#2 ↑,↑↑,ROW2_WP,ROW3_WP,ROYAL_WP;
272 ((¬(sq=WQR2)∧(¬(sq=WQN3)∧(¬(sq=WQB2)∧(¬(sq=BQB2)∧(¬(sq=WQ3)∧(¬(sq=WKB2)∧
(¬(sq=WKN3)∧¬(sq=WKR2)))))))⊃¬WPAWNS Pos(qx,sq))∧((x=Taken Move p∧(
PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x)  (1 9)

*****label QX_WPAWNS;
*****VI↑ p x sq;
```

```
273 ∀p x sq.(((¬(sq=WQR2)∧(¬(sq=WQN3)∧(¬(sq=WQB2)∧(¬(sq=BQB2)∧(¬(sq=WQ3)∧(¬(
sq=WKB2)∧(¬(sq=WKN3)∧¬(sq=WKR2)))))))⊃¬WPAWNS Pos(qx,sq))∧((x=Taken Move p∧
(PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x))  (1 9)
```

9.1. More particularly, the fallen piece, on WKR4, was not a white pawn.

```
*****VE QX_WPAWNS p,x,WKR4;
274 ((¬(WKR4=WQR2)∧(¬(WKR4=WQN3)∧(¬(WKR4=WQB2)∧(¬(WKR4=BQB2)∧(¬(WKR4=WQ3)∧(¬
(WKR4=WKB2)∧(¬(WKR4=WKN3)∧¬(WKR4=WKR2)))))))⊃¬WPAWNS Pos(qx,WKR4))∧((x=
Taken Move p∧(PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x)  (1 9)

*****simplify ↑;
275 ¬WPAWNS Pos(qx,WKR4)∧((x=Taken Move p∧(PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x)
  (1 9)
```

Section 4.2.8 The White Rook and King

9.2. There are two other white (valued) pieces on the board QBUD. There is a rook value on BQ2. This is either one of the two original white rooks, BQR or BKR, or a promoted white pawn.

```
*****VE MightBeRW qx,Pos(qx,BQ2);
276 Val(qx,Pos(qx,BQ2))=RW⊃((Pos(qx,BQ2)=WKR∨Pos(qx,BQ2)=WQR)∨(WPAWNS Pos(qx
,BQ2)∧PROMOTEDPAWN(qx,Pos(qx,BQ2))))
```

But none of the white pawns is on BQ2.

```
*****VE QX_WPAWNS p,x,BQ2;
277 ((¬(BQ2=WQR2)∧(¬(BQ2=WQN3)∧(¬(BQ2=WQB2)∧(¬(BQ2=BQB2)∧(¬(BQ2=WQ3)∧(¬(BQ2=
WKB2)∧(¬(BQ2=WKN3)∧¬(BQ2=WKR2)))))))⊃¬WPAWNS Pos(qx,BQ2))∧((x=Taken Move p∧
(PREDEGAME(p,qx)∨p=qx))⊃¬WPAWNS x)  (1 9)

*****VE ValueTransposition8 qx,BQ2,QBUD;
278 BOARD(qx,QBUD)⊃(Valueon(QBUD,BQ2)=Val(qx,Pos(qx,BQ2))∨Valueon(QBUD,BQ2)=
UD)

*****simplify ¬BQ2=WQR2∧¬BQ2=WQN3∧¬BQ2=WQB2∧¬BQ2=BQB2∧¬BQ2=WQ3∧¬BQ2=WKB2∧
*     ¬BQ2=WKN3∧¬BQ2=WKR2∧Valueon(QBUD,BQ2)=RW∧¬RW=UD∧
*     WROOKS WKR∧WROOKS WQR∧¬WKR=WQR;
279 ¬(BQ2=WQR2)∧(¬(BQ2=WQN3)∧(¬(BQ2=WQB2)∧(¬(BQ2=BQB2)∧(¬(BQ2=WQ3)∧(¬(BQ2=
WKB2)∧(¬(BQ2=WKN3)∧(¬(BQ2=WKR2)∧(Valueon(QBUD,BQ2)=RW∧(¬(RW=UD)∧(WROOKS WKR∧
(WROOKS WQR∧¬(WKR=WQR))))))))))))
```

Hence, the piece on BQ2 must be either the BQR or the BKR.

```
*****tauteq Pos(qx,BQ2)=WQR∨Pos(qx,BQ2)=WKR QX_QBUD,↑↑↑↑:↑;
280 Pos(qx,BQ2)=WQR∨Pos(qx,BQ2)=WKR  (1 9)
```

However, this is not the most useful formulation of this fact. What we really need is *names* for each of the white rooks. We maneuver to obtain a more pliable WFF.

```
*****tauteq Pos(qx BQ2)=WQR⊃ (Pos(qx BQ2)=WQR∧ (WQR=WQR∨WQR=WKR)∧
*     ¬Pos(qx,BQ2)=WKR∧ (WKR=WQR∨WKR=WKR)∧ ¬WQR=WKR) ↑↑:↑;
281 Pos(qx,BQ2)=WQR⊃(Pos(qx,BQ2)=WQR∧((WQR=WQR∨WQR=WKR)∧(¬(Pos(qx,BQ2)=WKR)∧
((WKR=WQR∨WKR=WKR)∧¬(WQR=WKR)))))  (1 9)
```

```
*****tauteq Pos(qx BQ2)=WKR⊃ (Pos(qx BQ2)=WKR∧ (WKR=WQR∨WKR=WKR)∧
*     ¬Pos(qx,BQ2)=WQR∧ (WQR=WQR∨WQR=WKR)∧ ¬WKR=WQR) ↑↑↑;↑↑;
282 Pos(qx,BQ2)=WKR⊃(Pos(qx,BQ2)=WKR∧((WKR=WQR∨WKR=WKR)∧(¬(Pos(qx,BQ2)=WQR)∧
((WQR=WQR∨WQR=WKR)∧¬(WKR=WQR)))))  (1 9)

*****unify Pos(qx BQ2)=WQR⊃ ∃ywr ywr1.((Pos(qx BQ2)=ywr∧ (ywr=WQR∨ywr=WKR)∧
*     ¬Pos(qx,BQ2)=ywr1∧ (ywr1=WQR∨ywr1=WKR)∧ ¬ywr=ywr1)) ↑↑;
283 Pos(qx,BQ2)=WQR⊃∃ywr ywr1.(Pos(qx,BQ2)=ywr∧((ywr=WQR∨ywr=WKR)∧(¬(Pos(qx,
BQ2)=ywr1)∧((ywr1=WQR∨ywr1=WKR)∧¬(ywr=ywr1)))))  (1 9)
```

More specifically, we want to rename the two white rooks to be ywr and ywr1, where we know that ywr is on the square BQ2, and that ywr1 is not ywr. With the proper manipulations, we obtain:

```
*****unify Pos(qx BQ2)=WKR⊃ ∃ywr ywr1.((Pos(qx BQ2)=ywr∧ (ywr=WQR∨ywr=WKR)∧
*     ¬Pos(qx BQ2)=ywr1∧ (ywr1=WQR∨ywr1=WKR)∧ ¬ywr=ywr1)) ↑↑;
284 Pos(qx,BQ2)=WKR⊃∃ywr ywr1.(Pos(qx,BQ2)=ywr∧((ywr=WQR∨ywr=WKR)∧(¬(Pos(qx,
BQ2)=ywr1)∧((ywr1=WQR∨ywr1=WKR)∧¬(ywr=ywr1)))))  (1 9)

*****taut ↑:#2 ↑,↑↑,↑↑↑↑;
285 ∃ywr ywr1.(Pos(qx,BQ2)=ywr∧((ywr=WQR∨ywr=WKR)∧(¬(Pos(qx,BQ2)=ywr1)∧((
ywr1=WQR∨ywr1=WKR)∧¬(ywr=ywr1)))))  (1 9)
```

permitting the renaming:

```
*****label CALL YWR;
***** ∃E ↑ ywr ywr1;
286 Pos(qx,BQ2)=ywr∧((ywr=WQR∨ywr=WKR)∧(¬(Pos(qx,BQ2)=ywr1)∧((ywr1=WQR∨ywr1=
WKR)∧¬(ywr=ywr1))))  (286)
```

which implies that the rook ywr was not the fallen piece (though the rook ywr1 might have been).

```
*****VE Unique qx,BQ2,WKR4,ywr;
287 Pos(qx,BQ2)=ywr⊃(Pos(qx,WKR4)=ywr=BQ2=WKR4)
```

The white king, on square BKR1, was certainly not the fallen piece.

```
*****VE Unique qx,BKR1,WKR4,WK;
288 Pos(qx,BKR1)=WK⊃(Pos(qx,WKR4)=WK=BKR1=WKR4)

*****label QX WK;
*****VE KingValueThm qx,QBUD,BKR1;
289 (BOARD(qx,QBUD)∧¬(Valueon(QBUD,BKR1)=UD))⊃((Pos(qx,BKR1)=WK=Valueon(QBUD
,BKR1)=KW)∧(Pos(qx,BKR1)=BK=Valueon(QBUD,BKR1)=KB))
```

9.3. The whitepieces include the white pawns, two rooks, knights, and bishops, and a white king and queen. But we have eliminated all but six of these pieces as candidates to be the fallen piece. Hence, it must have been one of them.

```
*****VE WhitepieceArePawnsOr_ yyw;
290 WHITEPIECE yyw=(WPAWNS yyw∨(yyw=WK∨(yyw=WQ∨(yyw=WKN∨(yyw=WKB∨(yyw=WKR∨(
yyw=WQB∨(yyw=WQN∨yyw=WQR)))))))

*****simplify WHITEPIECE yyw∧¬BQ2=WKR4∧¬BKR1=WKR4∧
*     Valueon(QBUD,BKR1)=KW∧¬KW=UD;
291 WHITEPIECE yyw∧(¬(BQ2=WKR4)∧(¬(BKR1=WKR4)∧(Valueon(QBUD,BKR1)=KW∧¬(KW=UD
))))
```

```
*****label WHICH_YYW;
*****tauteq yyw=WQ v yyw=WQB v yyw=WKB v yyw=ywr1 v yyw=WQN v yyw=WKN
*      QX_QBUD,SAME_ON_WKR4,CALL_YYW,QX_WPAWNS+2,CALL_YWR;↑;
292 yyw=WQv(yyw=WQBv(yyw=WKBv(yyw=ywr1v(yyw=WQNvyyw=WKN))))  (1 9 252 286)
```

We set the stage for further deductions.

Similarly, only these six white pieces were ever captured. Furthermore, If the capture occurred on a white square, then the white on black bishop (WQB) was not the captured piece.

```
*****VE MconseqfX qx,p,BKR1,WK;
293 ((p=qxvPREDEGAME(p,qx))∧Taken Move p=WK)⊃¬(Pos(qx,BKR1)=WK)

*****VE MconseqfX qx,p,BQ2,ywr;
294 ((p=qxvPREDEGAME(p,qx))∧Taken Move p=ywr)⊃¬(Pos(qx,BQ2)=ywr)

*****VE WhitepieceArePawnsOr_ x;
295 WHITEPIECE x=(WPAWNS xv(x=WKv(x=WQv(x=WKNv(x=WKBv(x=WKRv(x=WQBv(x=WQNvx=
WQR)))))))

*****VE WhereBishopTaken p,WQB,sq,WQB1;
296 (To Move p=sq∧(Pos(P0,WQB1)=WQB∧¬(WHITESQUARES WQB1=WHITESQUARES sq)))⊃¬
(Taken Move p=WQB)

*****simplify ¬WHITESQUARES WQB1∧Pos(P0,WQB1)=WQB;
297 ¬WHITESQUARES WQB1∧Pos(P0,WQB1)=WQB

*****VE MconseqfX qx p WKR4 yyw;
298 ((p=qxvPREDEGAME(p,qx))∧Taken Move p=yyw)⊃¬(Pos(qx,WKR4)=yyw)

*****tauteq ((PREDEGAME(p qx)vp=qx)∧Taken Move p=x∧WHITEPIECE x)⊃
*      ((x=WQvx=WQNvx=WKNvx=WQBvx=WKBvx=ywr1)∧(¬x=yyw)∧
*      ((To Move p=sq∧WHITESQUARES sq)⊃¬x=WQB))
*    ↑↑↑↑↑↑;↑,QX_WK,QX_WK+2,QX_QBUD,CALL_YWR,QX_WPAWNS+2,CALL_YYW;
299 ((PREDEGAME(p,qx)vp=qx)∧(Taken Move p=x∧WHITEPIECE x))⊃((x=WQv(x=WQNv(x=
WKNv(x=WQBv(x=WKBvx=ywr1)))))∧(¬(x=yyw)∧((To Move p=sq∧WHITESQUARES sq)⊃¬(x=
WQB))))  (1 9 252 286)

*****label WHICH_QX_TAKEN;
*****Vl ↑ p x sq;
300 Vp x sq.(((PREDEGAME(p,qx)vp=qx)∧(Taken Move p=x∧WHITEPIECE x))⊃((x=WQv(
x=WQNv(x=WKNv(x=WQBv(x=WKBvx=ywr1)))))∧(¬(x=yyw)∧((To Move p=sq∧WHITESQUARES
 sq)⊃¬(x=WQB)))))  (1 9 252 286)
```

Section 4.2.9 Black Pawn Captures

10. We see that the BQNP and BKBP have, between them, captured white pieces on the squares BQR3, BK3, BQ4, and WQB4.

```
*****VE BlackPawnCaptureThm qx , BQNP , BQN2 , BQR3 , BQR3 , QBUD ;
301 (Pos(P0,BQN2)=BQNP∧(Pos(qx,BQR3)=BQNP∧(MUST_PAWN_CAPTURES(BQN2,BQR3,
Piececolor BQNP)∧(BOARD(qx,QBUD)∧Valueon(QBUD,BQR3)=PB))))⊃((BQR3=BQR3v(
SAMEDIAG(BQR3,BQR3)∧(SAMEDIAG(BQR3,BQN2)∧BETWEEN(Row BQR3,Row BQR3,Row BQN2)
)))⊃∃q3 x3.((PREDEGAME(q3,qx)vq3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BQNP∧(
To Move q3=BQR3∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=
```

```
BQR3⊃(Mover Move q3=BQNP⊃((Taken Move q3=x3∧¬WHITEPIECE BQNP)⊃(WHITEPIECE x3
∧(¬(Row BQR3=6)⊃Pos(Prevpos q3,BQR3)=x3)))))))))

*****VE BlackPawnCaptureThm qx , BKBP , BKB2 , WQB4 , BK3 , QBUD ;
302 (Pos(P0,BKB2)=BKBP∧(Pos(qx,WQB4)=BKBP∧(MUST_PAWN_CAPTURES(BKB2,WQB4,
Piececolor BKBP)∧(BOARD(qx,QBUD)∧Valueon(QBUD,WQB4)=PB))))⊃((BK3=WQB4∨(
SAMEDIAG(WQB4,BK3)∧(SAMEDIAG(BK3,BKB2)∧BETWEEN(Row WQB4,Row BK3,Row BKB2))))
⊃∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(To
Move q3=BK3∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=BK3⊃(
Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3∧(¬(
Row BK3=6)⊃Pos(Prevpos q3,BK3)=x3)))))))))

*****VE BlackPawnCaptureThm qx , BKBP , BKB2 , WQB4 , BQ4 , QBUD ;
303 (Pos(P0,BKB2)=BKBP∧(Pos(qx,WQB4)=BKBP∧(MUST_PAWN_CAPTURES(BKB2,WQB4,
Piececolor BKBP)∧(BOARD(qx,QBUD)∧Valueon(QBUD,WQB4)=PB))))⊃((BQ4=WQB4∨(
SAMEDIAG(WQB4,BQ4)∧(SAMEDIAG(BQ4,BKB2)∧BETWEEN(Row WQB4,Row BQ4,Row BKB2))))
⊃∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(To
Move q3=BQ4∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=BQ4⊃(
Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3∧(¬(
Row BQ4=6)⊃Pos(Prevpos q3,BQ4)=x3)))))))))

*****VE BlackPawnCaptureThm qx , BKBP , BKB2 , WQB4 , WQB4 , QBUD ;
304 (Pos(P0,BKB2)=BKBP∧(Pos(qx,WQB4)=BKBP∧(MUST_PAWN_CAPTURES(BKB2,WQB4,
Piececolor BKBP)∧(BOARD(qx,QBUD)∧Valueon(QBUD,WQB4)=PB))))⊃((WQB4=WQB4∨(
SAMEDIAG(WQB4,WQB4)∧(SAMEDIAG(WQB4,BKB2)∧BETWEEN(Row WQB4,Row WQB4,Row BKB2)
)))⊃∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(
To Move q3=WQB4∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=
WQB4⊃(Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3
∧(¬(Row WQB4=6)⊃Pos(Prevpos q3,WQB4)=x3)))))))))

*****label PTSIMP;
***** simplify (Pos(P0,BQN2)=BQNP∧
*  MUST_PAWN_CAPTURES(BQN2,BQR3,Piececolor BQNP)∧Valueon(QBUD,BQR3)=PB∧
*  BQR3=BQR3)∧(Pos(P0,BKB2)=BKBP∧
*  MUST_PAWN_CAPTURES(BKB2,WQB4,Piececolor BKBP)∧Valueon(QBUD,WQB4)=PB∧
*  (SAMEDIAG(WQB4,BK3)∧SAMEDIAG(BK3,BKB2)∧
*  BETWEEN(Row WQB4,Row BK3,Row BKB2)))∧(Pos(P0,BKB2)=BKBP∧
*  MUST_PAWN_CAPTURES(BKB2,WQB4,Piececolor BKBP)∧Valueon(QBUD,WQB4)=PB∧
*  (SAMEDIAG(WQB4,BQ4)∧SAMEDIAG(BQ4,BKB2)∧
*  BETWEEN(Row WQB4,Row BQ4,Row BKB2)))∧(Pos(P0,BKB2)=BKBP∧
*  MUST_PAWN_CAPTURES(BKB2,WQB4,Piececolor BKBP)∧
*  Valueon(QBUD,WQB4)=PB∧WQB4=WQB4);
305 (Pos(P0,BQN2)=BQNP∧(MUST_PAWN_CAPTURES(BQN2,BQR3,Piececolor BQNP)∧(
Valueon(QBUD,BQR3)=PB∧BQR3=BQR3)))∧((Pos(P0,BKB2)=BKBP∧(MUST_PAWN_CAPTURES(
BKB2,WQB4,Piececolor BKBP)∧(Valueon(QBUD,WQB4)=PB∧(SAMEDIAG(WQB4,BK3)∧(
SAMEDIAG(BK3,BKB2)∧BETWEEN(Row WQB4,Row BK3,Row BKB2))))))∧((Pos(P0,BKB2)=
BKBP∧(MUST_PAWN_CAPTURES(BKB2,WQB4,Piececolor BKBP)∧(Valueon(QBUD,WQB4)=PB∧(
SAMEDIAG(WQB4,BQ4)∧(SAMEDIAG(BQ4,BKB2)∧BETWEEN(Row WQB4,Row BQ4,Row BKB2))))
))∧(Pos(P0,BKB2)=BKBP∧(MUST_PAWN_CAPTURES(BKB2,WQB4,Piececolor BKBP)∧(
Valueon(QBUD,WQB4)=PB∧WQB4=WQB4)))))
```

Hence, there must have existed four positions in the course of this game where the move that reached that position was one of these captures.

```
*****tauteq ↑↑↑↑↑;#2#2 ↑↑↑↑↑,PTSIMP,ROW3R_BP,QX_QBUD;
306 ∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BQNP∧(
To Move q3=BQR3∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=
```

```
BQR3⊃(Mover Move q3=BQNP⊃((Taken Move q3=x3∧¬WHITEPIECE BQNP)⊃(WHITEPIECE x3
∧(¬(Row BQR3=6)⊃Pos(Prevpos q3,BQR3)=x3)))))))  (1 9)

*****tauteq ↑↑↑↑↑:#2#2 ↑↑↑↑↑,PTSIMP,B5_BP,QX_QBUD;
307 ∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(
To Move q3=BK3∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=BK3
⊃(Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3∧(¬(
Row BK3=6)⊃Pos(Prevpos q3,BK3)=x3)))))))  (1 9)

*****tauteq ↑↑↑↑↑:#2#2 ↑↑↑↑↑,PTSIMP,B5_BP,QX_QBUD;
308 ∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(
To Move q3=BQ4∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=BQ4
⊃(Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3∧(¬(
Row BQ4=6)⊃Pos(Prevpos q3,BQ4)=x3)))))))  (1 9)

*****tauteq ↑↑↑↑↑:#2#2 ↑↑↑↑↑,PTSIMP,B5_BP,QX_QBUD;
309 ∃q3 x3.((PREDEGAME(q3,qx)∨q3=qx)∧((TAKINGS Move q3∧(Mover Move q3=BKBP∧(
To Move q3=WQB4∧Taken Move q3=x3)))∧(PREDEGAME(Prevpos q3,qx)∧(To Move q3=
WQB4⊃(Mover Move q3=BKBP⊃((Taken Move q3=x3∧¬WHITEPIECE BKBP)⊃(WHITEPIECE x3
∧(¬(Row WQB4=6)⊃Pos(Prevpos q3,WQB4)=x3)))))))  (1 9)
```

Let us call these positions p1, p2, p3, and p4, respectively. We will refer to the white pieces captured as xa, xb, xc, and xd.

```
*****label CALL_PN;
*****∃E ↑↑↑↑ p1 xa;
310 (PREDEGAME(p1,qx)∨p1=qx)∧((TAKINGS Move p1∧(Mover Move p1=BQNP∧(To Move
p1=BQR3∧Taken Move p1=xa)))∧(PREDEGAME(Prevpos p1,qx)∧(To Move p1=BQR3⊃(
Mover Move p1=BQNP⊃((Taken Move p1=xa∧¬WHITEPIECE BQNP)⊃(WHITEPIECE xa∧(¬(
Row BQR3=6)⊃Pos(Prevpos p1,BQR3)=xa)))))))  (310)

*****∃E ↑↑↑↑ p2 xb;
311 (PREDEGAME(p2,qx)∨p2=qx)∧((TAKINGS Move p2∧(Mover Move p2=BKBP∧(To Move
p2=BK3∧Taken Move p2=xb)))∧(PREDEGAME(Prevpos p2,qx)∧(To Move p2=BK3⊃(Mover
Move p2=BKBP⊃((Taken Move p2=xb∧¬WHITEPIECE BKBP)⊃(WHITEPIECE xb∧(¬(Row BK3=
6)⊃Pos(Prevpos p2,BK3)=xb)))))))  (311)

*****∃E ↑↑↑↑ p3 xc;
312 (PREDEGAME(p3,qx)∨p3=qx)∧((TAKINGS Move p3∧(Mover Move p3=BKBP∧(To Move
p3=BQ4∧Taken Move p3=xc)))∧(PREDEGAME(Prevpos p3,qx)∧(To Move p3=BQ4⊃(Mover
Move p3=BKBP⊃((Taken Move p3=xc∧¬WHITEPIECE BKBP)⊃(WHITEPIECE xc∧(¬(Row BQ4=
6)⊃Pos(Prevpos p3,BQ4)=xc)))))))  (312)

*****∃E ↑↑↑↑ p4 xd;
313 (PREDEGAME(p4,qx)∨p4=qx)∧((TAKINGS Move p4∧(Mover Move p4=BKBP∧(To Move
p4=WQB4∧Taken Move p4=xd)))∧(PREDEGAME(Prevpos p4,qx)∧(To Move p4=WQB4⊃(
Mover Move p4=BKBP⊃((Taken Move p4=xd∧¬WHITEPIECE BKBP)⊃(WHITEPIECE xd∧(¬(
Row WQB4=6)⊃Pos(Prevpos p4,WQB4)=xd)))))))  (313)
```

Clearly, each of xa through xd must be one of the white pieces that could have been captured.

```
*****label SIMPWS;
***** simplify (¬WHITEPIECE BQNP∧¬WHITEPIECE BKBP)∧
*  WHITESQUARES BQR3∧WHITESQUARES BK3∧WHITESQUARES BQ4∧WHITESQUARES WQB4;
314 (¬WHITEPIECE BQNP∧¬WHITEPIECE BKBP)∧(WHITESQUARES BQR3∧(WHITESQUARES BK3
∧(WHITESQUARES BQ4∧WHITESQUARES WQB4)))
```

```
*****VE WHICH_QX_TAKEN p1,xa,BQR3;
315 ((PREDEGAME(p1,qx)vp1=qx)∧(Taken Move p1=xa∧WHITEPIECE xa))⊃((xa=WQv(xa=
WQNv(xa=WKNv(xa=WQBv(xa=WKBvxa=ywr1)))))∧(¬(xa=yyw)∧((To Move p1=BQR3∧
WHITESQUARES BQR3)⊃¬(xa=WQB))))  (1 9 252 286)

*****VE WHICH_QX_TAKEN p2,xb,BK3;
316 ((PREDEGAME(p2,qx)vp2=qx)∧(Taken Move p2=xb∧WHITEPIECE xb))⊃((xb=WQv(xb=
WQNv(xb=WKNv(xb=WQBv(xb=WKBvxb=ywr1)))))∧(¬(xb=yyw)∧((To Move p2=BK3∧
WHITESQUARES BK3)⊃¬(xb=WQB))))  (1 9 252 286)

*****VE WHICH_QX_TAKEN p3,xc,BQ4;
317 ((PREDEGAME(p3,qx)vp3=qx)∧(Taken Move p3=xc∧WHITEPIECE xc))⊃((xc=WQv(xc=
WQNv(xc=WKNv(xc=WQBv(xc=WKBvxc=ywr1)))))∧(¬(xc=yyw)∧((To Move p3=BQ4∧
WHITESQUARES BQ4)⊃¬(xc=WQB))))  (1 9 252 286)

*****VE WHICH_QX_TAKEN p4,xd,WQB4;
318 ((PREDEGAME(p4,qx)vp4=qx)∧(Taken Move p4=xd∧WHITEPIECE xd))⊃((xd=WQv(xd=
WQNv(xd=WKNv(xd=WQBv(xd=WKBvxd=ywr1)))))∧(¬(xd=yyw)∧((To Move p4=WQB4∧
WHITESQUARES WQB4)⊃¬(xd=WQB))))  (1 9 252 286)
```

Since these are white squares, each of xa through xd was neither the WQB (white on black bishop), nor, of course yyw (the piece that fell from the board).

```
*****label WHO_XA;
***** tauteq (xa=WQvxa=WQNvxa=WKNvxa=WKBvxa=ywr1)∧¬xa=yyw
*     CALL_PN , SIMPWS, ↑↑↑↑;
319 (xa=WQv(xa=WQNv(xa=WKNv(xa=WKBvxa=ywr1))))∧¬(xa=yyw)  (1 9 252 286 310)

*****tauteq (xb=WQvxb=WQNvxb=WKNvxb=WKBvxb=ywr1)∧¬xb=yyw
*     CALL_PN+1 , SIMPWS , ↑↑↑↑;
320 (xb=WQv(xb=WQNv(xb=WKNv(xb=WKBvxb=ywr1))))∧¬(xb=yyw)  (1 9 252 286 311)

*****tauteq (xc=WQvxc=WQNvxc=WKNvxc=WKBvxc=ywr1)∧¬xc=yyw
*     CALL_PN+2 , SIMPWS, ↑↑↑↑;
321 (xc=WQv(xc=WQNv(xc=WKNv(xc=WKBvxc=ywr1))))∧¬(xc=yyw)  (1 9 252 286 312)

*****tauteq (xd=WQvxd=WQNvxd=WKNvxd=WKBvxd=ywr1)∧¬xd=yyw
*     CALL_PN+3 , SIMPWS, ↑↑↑↑;
322 (xd=WQv(xd=WQNv(xd=WKNv(xd=WKBvxd=ywr1))))∧¬(xd=yyw)  (1 9 252 286 313)
```

We need also establish that these moves all captured different pieces. A lemma, ***DifferentTaken***, serves us well here. It states that if a capture took place on differing squares, or by differing pieces, or any other way of proving the capturing positions different, then the captured pieces were not the same piece. As there are six equalities to establish, we invoke the theorem six times.

```
*****VE DifferentTaken p1 p2 qx xa xb;
323 (((p2=qxvPREDEGAME(p2,qx))∧(p1=qxvPREDEGAME(p1,qx)))∧((¬(To Move p1=To
Move p2)v(¬(Mover Move p1=Mover Move p2)v(PREDEGAME(p1,p2)v¬(p1=p2))))∧(
Taken Move p1=xa∧Taken Move p2=xb)))⊃¬(xa=xb)

*****VE DifferentTaken p1 p3 qx xa xc;
324 (((p3=qxvPREDEGAME(p3,qx))∧(p1=qxvPREDEGAME(p1,qx)))∧((¬(To Move p1=To
Move p3)v(¬(Mover Move p1=Mover Move p3)v(PREDEGAME(p1,p3)v¬(p1=p3))))∧(
Taken Move p1=xa∧Taken Move p3=xc)))⊃¬(xa=xc)
```

```
*****VE DifferentTaken p1 p4 qx xa xd;
325 (((p4=qx∨PREDEGAME(p4,qx))∧(p1=qx∨PREDEGAME(p1,qx)))∧((¬(To Move p1=To
Move p4)∨(¬(Mover Move p1=Mover Move p4)∨(PREDEGAME(p1,p4)∨¬(p1=p4))))∧(
Taken Move p1=xa∧Taken Move p4=xd)))⊃¬(xa=xd)

*****VE DifferentTaken p2 p3 qx xb xc;
326 (((p3=qx∨PREDEGAME(p3,qx))∧(p2=qx∨PREDEGAME(p2,qx)))∧((¬(To Move p2=To
Move p3)∨(¬(Mover Move p2=Mover Move p3)∨(PREDEGAME(p2,p3)∨¬(p2=p3))))∧(
Taken Move p2=xb∧Taken Move p3=xc)))⊃¬(xb=xc)

*****VE DifferentTaken p2 p4 qx xb xd;
327 (((p4=qx∨PREDEGAME(p4,qx))∧(p2=qx∨PREDEGAME(p2,qx)))∧((¬(To Move p2=To
Move p4)∨(¬(Mover Move p2=Mover Move p4)∨(PREDEGAME(p2,p4)∨¬(p2=p4))))∧(
Taken Move p2=xb∧Taken Move p4=xd)))⊃¬(xb=xd)

*****VE DifferentTaken p3 p4 qx xc xd;
328 (((p4=qx∨PREDEGAME(p4,qx))∧(p3=qx∨PREDEGAME(p3,qx)))∧((¬(To Move p3=To
Move p4)∨(¬(Mover Move p3=Mover Move p4)∨(PREDEGAME(p3,p4)∨¬(p3=p4))))∧(
Taken Move p3=xc∧Taken Move p4=xd)))⊃¬(xc=xd)
```

And compact its result to a single step.

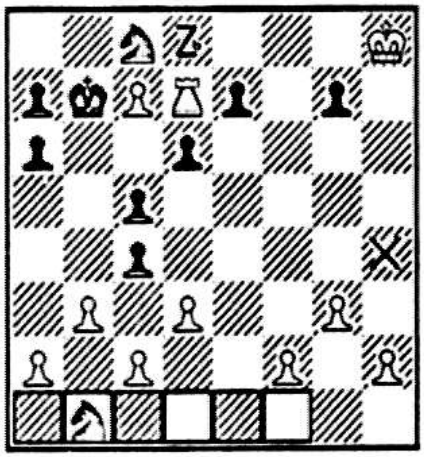
```
*****simplify ¬BQNP=BKBP ∧ ¬WQB4 = BQ4 ∧ ¬ WQB4 = BK3 ∧¬BQ4 = BK3;
329 ¬(BQNP=BKBP)∧(¬(WQB4=BQ4)∧(¬(WQB4=BK3)∧¬(BQ4=BK3)))

*****label NOT_XN_EQ;
*****tauteq ¬xa=xb∧ ¬xa=xc∧ ¬xa=xd∧ ¬xb=xc∧ ¬xb=xd∧ ¬xc=xd
*      ↑↑↑↑↑↑↑↑;↑ , CALL_PN:CALL_PN+3;
330 ¬(xa=xb)∧(¬(xa=xc)∧(¬(xa=xd)∧(¬(xb=xc)∧(¬(xb=xd)∧¬(xc=xd))))) (310 311
312 313)
```

Section 4.2.10 The Black Pawn's Path to Promotion

11. We have proven (back on step 207) that the black king rook's pawn had promoted. Therefore, there must have existed some position (in the course of this game) where he moved onto the eighth row. Let us call this position qy.

```
*****VE BlackPromtesOn8A qx BKRP;
331 PROMOTEDPAWN(qx,BKRP)⊃∃p.((PAWNPROM Move p∧((PREDEGAME(p,qx)∨p=qx)∧Mover
 Move p=BKRP))∧Row To Move p=8)

*****taut ↑:#2 ↑,PROM_BKRP;
332 ∃p.((PAWNPROM Move p∧((PREDEGAME(p,qx)∨p=qx)∧Mover Move p=BKRP))∧Row To
Move p=8)  (1 9)

*****label CALL_QY;
***** ∃E ↑ qy;
333 (PAWNPROM Move qy∧((PREDEGAME(qy,qx)∨qy=qx)∧Mover Move qy=BKRP))∧Row To
Move qy=8  (333)
```

11.1. Our final lemma specifically applicable to this proof, *FarTaken* (section B.4.2) states that if the BKRP promoted on any square to the left of WKN1, *figure 42* then this pawn must have captured two white pieces on the way to his elevation.



We assume that the BKRP promoted on one of these squares.

figure 42

```
*****∀E FarTaken qy;
334 (PAWNPROM Move qy∧(Mover Move qy=BKRP∧(¬(To Move qy=WKR1)∧¬(To Move qy=
WKN1))))⊃∃q1 q2 x1 x2.(((PREDEGAME(q1,qy)∨q1=qy)∧(PREDEGAME(q2,q1)∧(
PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move q1=BKRP∧(
Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2)))))))∧¬(x1=x2))
```

Let us assume that the promotion was on one of these squares

```
*****label TAKE_2_ASSUMPTION;
*****assume ¬(To Move qy=WKN1)∧ ¬(To Move qy=WKR1);
335 ¬(To Move qy=WKN1)∧¬(To Move qy=WKR1)  (335)
```

We call the positions in which the two white pieces were captured q1 and q2, the respective captured pieces, x1 and x2.

```
*****taut ↑↑:#2 ↑↑↑:↑;
336 ∃q1 q2 x1 x2.(((PREDEGAME(q1,qy)∨q1=qy)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,
q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move q1=BKRP∧(Mover Move q2=
BKRP∧(Taken Move q1=x1∧Taken Move q2=x2)))))))∧¬(x1=x2))  (333 335)

*****label CALL_ON;
*****∃E ↑ q1,q2,x1,x2;
337 ((PREDEGAME(q1,qy)∨q1=qy)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS
Move q1∧(TAKINGS Move q2∧(Mover Move q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move
 q1=x1∧Taken Move q2=x2)))))))∧¬(x1=x2)  (337)
```

As q1 and q2 occurred in the game that led to qy, and qy occurred in the game that reach qx, both q1 and q2 are ancestors of qx.

```
*****∀E TransitiveGenealogy q1,qy,qx;
338 (PREDEGAME(q1,qy)∧PREDEGAME(qy,qx))⊃PREDEGAME(q1,qx)

*****∀E TransitiveGenealogy q2,q1,qx;
```

```
339 (PREDEGAME(q2,q1)∧PREDEGAME(q1,qx))⊃PREDEGAME(q2,qx)

*****label PRED_QN;
*****tauteq (PREDEGAME(q1 qx)∨q1=qx)∧PREDEGAME(q2 qx) ↑↑↑:↑,CALL_QY;
340 (PREDEGAME(q1,qx)∨q1=qx)∧PREDEGAME(q2,qx)  (333 337)
```

And ×1 and ×2 must also be in the capture set.

```
*****VE WhiteCapturedOnThm Prevpos q1,q1,BKRP,x1,To Move q1;
341 Prevpos q1=Prevpos q1⊃(To Move q1=To Move q1⊃(Mover Move q1=BKRP⊃((Taken
 Move q1=x1∧¬WHITEPIECE BKRP)⊃(WHITEPIECE x1∧(¬(Row To Move q1=6)⊃Pos(
Prevpos q1,To Move q1)=x1)))))

*****VE WhiteCapturedOnThm Prevpos q2,q2,BKRP,x2,To Move q2;
342 Prevpos q2=Prevpos q2⊃(To Move q2=To Move q2⊃(Mover Move q2=BKRP⊃((Taken
 Move q2=x2∧¬WHITEPIECE BKRP)⊃(WHITEPIECE x2∧(¬(Row To Move q2=6)⊃Pos(
Prevpos q2,To Move q2)=x2)))))

*****VE WHICH_QX_TAKEN q1,x1,To Move q1;
343 ((PREDEGAME(q1,qx)∨q1=qx)∧(Taken Move q1=x1∧WHITEPIECE x1))⊃((x1=WQ∨(x1=
WQN∨(x1=WKN∨(x1=WQB∨(x1=WKB∨x1=ywr1)))))∧(¬(x1=yyw)∧((To Move q1=To Move q1∧
WHITESQUARES To Move q1)⊃¬(x1=WQB))))  (1 9 252 286)

*****VE WHICH_QX_TAKEN q2,x2,To Move q2;
344 ((PREDEGAME(q2,qx)∨q2=qx)∧(Taken Move q2=x2∧WHITEPIECE x2))⊃((x2=WQ∨(x2=
WQN∨(x2=WKN∨(x2=WQB∨(x2=WKB∨x2=ywr1)))))∧(¬(x2=yyw)∧((To Move q2=To Move q2∧
WHITESQUARES To Move q2)⊃¬(x2=WQB))))  (1 9 252 286)

*****simplify ¬WHITEPIECE BKRP;
345 ¬WHITEPIECE BKRP

*****label WHO_X1;
*****tauteq ↑↑↑:#2#1∧↑↑↑:#2#2#1 ↑,↑↑↑,↑↑↑↑↑,PRED_QN,CALL_QN,CALL_QY;
346 (x1=WQ∨(x1=WQN∨(x1=WKN∨(x1=WQB∨(x1=WKB∨x1=ywr1)))))∧¬(x1=yyw)  (1 9 252
286 333 337)

*****taut   ↑↑↑:#2#1∧↑↑↑:#2#2#1 ↑↑,↑↑↑,↑↑↑↑↑,PRED_QN,CALL_QN,CALL_QY;
347 (x2=WQ∨(x2=WQN∨(x2=WKN∨(x2=WQB∨(x2=WKB∨x2=ywr1)))))∧¬(x2=yyw)  (1 9 252
286 333 337)
```

Since ×1 and ×2 were captured by BKRP, and ×a through ×d, by BQNP and BKBP, ×1 and ×2 are not equal to any of ×a through ×d. *DifferentTakenFour* is merely four instantiations of *DifferentTaken*, compressed into one WFF. This is a good illustration of the inaccuracies involved in measuring proof size merely by counting steps.

```
*****VE DifferentTakenFour qx q1 p1 p2 p3 p4 x1 xa xb xc xd;
348 ((q1=qx∨PREDEGAME(q1,qx))∧((p1=qx∨PREDEGAME(p1,qx))∧((p2=qx∨PREDEGAME(p2
,qx))∧((p3=qx∨PREDEGAME(p3,qx))∧((p4=qx∨PREDEGAME(p4,qx))∧(¬(Mover Move p1=
Mover Move q1)∧(¬(Mover Move p2=Mover Move q1)∧(¬(Mover Move p3=Mover Move
q1)∧(¬(Mover Move p4=Mover Move q1)∧(Taken Move q1=x1∧(Taken Move p1=xa∧(
Taken Move p2=xb∧(Taken Move p3=xc∧Taken Move p4=xd))))))))))))⊃(¬(xa=x1)∧(
¬(xb=x1)∧(¬(xc=x1)∧¬(xd=x1))))

*****VE DifferentTakenFour qx q2 p1 p2 p3 p4 x2 xa xb xc xd;
349 ((q2=qx∨PREDEGAME(q2,qx))∧((p1=qx∨PREDEGAME(p1,qx))∧((p2=qx∨PREDEGAME(p2
```

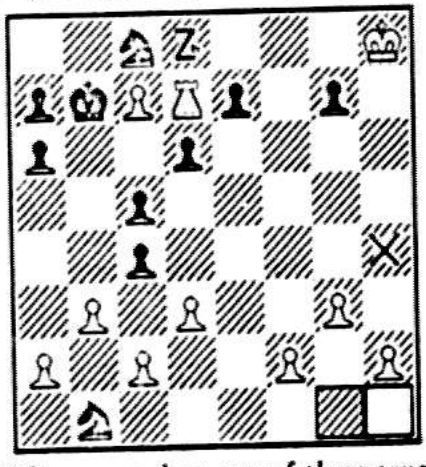
```
,qx))∧((p3=qx∨PREDEGAME(p3,qx))∧((p4=qx∨PREDEGAME(p4,qx))∧(¬(Mover Move p1=
Mover Move q2)∧(¬(Mover Move p2=Mover Move q2)∧(¬(Mover Move p3=Mover Move
q2)∧(¬(Mover Move p4=Mover Move q2)∧(Taken Move q2=x2∧(Taken Move p1=xa∧(
Taken Move p2=xb∧(Taken Move p3=xc∧Taken Move p4=xd))))))))))))⊃(¬(xa=x2)∧(
¬(xb=x2)∧(¬(xc=x2)∧¬(xd=x2))))

*****label DIFFMOVERS;
***** simplify ¬BKRP=BKBP∧¬BKRP=BQNP;
350 ¬(BKRP=BKBP)∧¬(BKRP=BQNP)

*****tauteq ↑↑↑:#2 CALL_PN:CALL_PN+3,CALL_QY,CALL_QN,PRED_QN,↑↑↑,↑;
351 ¬(xa=x1)∧(¬(xb=x1)∧(¬(xc=x1)∧¬(xd=x1)))  (310 311 312 313 333 337)

*****tauteq ↑↑↑:#2 CALL_PN:CALL_PN+3,CALL_QY,CALL_QN,PRED_QN,↑↑↑,↑↑;
352 ¬(xa=x2)∧(¬(xb=x2)∧(¬(xc=x2)∧¬(xd=x2)))  (310 311 312 313 333 337)
```

11.1.1. We have presume a situation that is clearly impossible. We have posited the existence of six captured white piece, all different, and a fallen piece, all to be selected from the pool of six unaccounted for white chessmen. Our pigeon will not fit into this hole. We can tautologically produce the contradiction:

```
*****tauteq FALSE
*     ↑↑:↑,NOT_XN_EQ,WHICH_YYW,WHO_XA:WHO_XA+3,WHO_X1:WHO_X1+1,CALL_QN;
353 FALSE  (1 9 333 335)
```

11.2. This permits us to negate one of our assumptions. We of course choose the assumption that the BKRP promoted to the left of WKN1. Hence, we get something equivalent to specifying the promotion square of BKRP to be either WKN1 or WKR1.

```
*****label N1 OR R1;
*****¬I ↑ TAKE_2_ASSUMPTION;
354 ¬(¬(To Move qy=WKN1)∧¬(To Move qy=WKR1))  (1 9 333)
```



BKRP promoted on one of these squares.

figure 43

Section 4.2.11 The Source of the Promoting Move

In either case, there was a move when the pawn valued BKRP was on WKN1 or WKR1. Notice that for the next few steps, we are compelled to follow two parallel proof strands, one for each of the possible promotion squares. We will merge these strands as soon as possible.

```
*****VE BlackDidPromote qy BKRP WKN1;
355 To Move qy=WKN1⊃(Mover Move qy=BKRP⊃(PAWNPROM Move qy⊃(Val(Prevpos qy,
BKRP)=PB∧Pos(qy,WKN1)=BKRP)))

*****VE BlackDidPromote qy BKRP WKR1;
356 To Move qy=WKR1⊃(Mover Move qy=BKRP⊃(PAWNPROM Move qy⊃(Val(Prevpos qy,
BKRP)=PB∧Pos(qy,WKR1)=BKRP)))
```

11.2.1. Properly composed, we can use our chess eye to see backwards as well as just looking about. A black pawn on WKN1 came from one of WKB2, WKN2, or WKR2; on WKR1, from either WKN2 or WKR2.

```
*****VE BlackPawnMoveThm qx,qy,BKRP,WKN1 :
357 (Pos(qy,WKN1)=BKRP∧(¬(Pos(P0,WKN1)=BKRP)∧((PREDEGAME(qy,qx)∨qy=qx)∧Val(
Prevpos qy,BKRP)=PB)))⊃∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To
Move q=WKN1∧(VALUEP Val(Prevpos q,BKRP)∧((¬(Row WKN1=6)∧¬(Row WKN1=4))⊃((
From Move q=Makesquare(Wsucf Row WKN1,Column WKN1)∧Pos(Prevpos q,WKN1)=EMPTY
)∨((Taken Move q=Pos(Prevpos q,WKN1)∧WHITEPIECE Pos(Prevpos q,WKN1))∧(From
Move q=Makesquare(Wsucf Row WKN1,L2touchf Column WKN1)∨From Move q=
Makesquare(Wsucf Row WKN1,R2touchf Column WKN1)))))))))

*****VE BlackPawnMoveThm qx,qy,BKRP,WKR1 :
358 (Pos(qy,WKR1)=BKRP∧(¬(Pos(P0,WKR1)=BKRP)∧((PREDEGAME(qy,qx)∨qy=qx)∧Val(
Prevpos qy,BKRP)=PB)))⊃∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To
Move q=WKR1∧(VALUEP Val(Prevpos q,BKRP)∧((¬(Row WKR1=6)∧¬(Row WKR1=4))⊃((
From Move q=Makesquare(Wsucf Row WKR1,Column WKR1)∧Pos(Prevpos q,WKR1)=EMPTY
)∨((Taken Move q=Pos(Prevpos q,WKR1)∧WHITEPIECE Pos(Prevpos q,WKR1))∧(From
Move q=Makesquare(Wsucf Row WKR1,L2touchf Column WKR1)∨From Move q=
Makesquare(Wsucf Row WKR1,R2touchf Column WKR1)))))))))

*****simplify ¬(Pos(P0,WKR1)=BKRP)∧¬(Pos(P0,WKN1)=BKRP);
359 ¬(Pos(P0,WKR1)=BKRP)∧¬(Pos(P0,WKN1)=BKRP)
```

In either case, there was a position when BKRP was on this From square.

```
*****label N1_assume;
*****assume To Move qy=WKN1;
360 To Move qy=WKN1  (360)

*****label R1_assume;
*****assume To Move qy=WKR1;
361 To Move qy=WKR1  (361)

*****tauteq ↑↑↑↑↑:#2 ↑↑↑↑↑↑↑,↑↑↑↑↑,359,↑↑,CALL_QY;
362 ∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To Move q=WKN1∧(VALUEP
Val(Prevpos q,BKRP)∧((¬(Row WKN1=6)∧¬(Row WKN1=4))⊃((From Move q=Makesquare(
Wsucf Row WKN1,Column WKN1)∧Pos(Prevpos q,WKN1)=EMPTY)∨((Taken Move q=Pos(
Prevpos q,WKN1)∧WHITEPIECE Pos(Prevpos q,WKN1))∧(From Move q=Makesquare(
Wsucf Row WKN1,L2touchf Column WKN1)∨From Move q=Makesquare(Wsucf Row WKN1,
R2touchf Column WKN1)))))))))  (333 360)
```

```
*****tauteq ↑↑↑↑↑:#2 ↑↑↑↑↑↑↑,↑↑↑↑↑,359,↑↑,CALL_QY;
363 ∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To Move q=WKR1∧(VALUEP
Val(Prevpos q,BKRP)∧((¬(Row WKR1=6)∧¬(Row WKR1=4))⊃((From Move q=Makesquare(
Wsucf Row WKR1,Column WKR1)∧Pos(Prevpos q,WKR1)=EMPTY)∨((Taken Move q=Pos(
Prevpos q,WKR1)∧WHITEPIECE Pos(Prevpos q,WKR1))∧(From Move q=Makesquare(
Wsucf Row WKR1,L2touchf Column WKR1)∨From Move q=Makesquare(Wsucf Row WKR1,
R2touchf Column WKR1))))))))) (333 361)
```

We call that position q1. We use the chess eye to simplify the defining WFF of q1.

```
*****∃E ↑↑ q1;
364 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(To Move q1=WKN1∧(VALUEP
Val(Prevpos q1,BKRP)∧((¬(Row WKN1=6)∧¬(Row WKN1=4))⊃((From Move q1=
Makesquare(Wsucf Row WKN1,Column WKN1)∧Pos(Prevpos q1,WKN1)=EMPTY)∨((Taken
Move q1=Pos(Prevpos q1,WKN1)∧WHITEPIECE Pos(Prevpos q1,WKN1))∧(From Move q1=
Makesquare(Wsucf Row WKN1,L2touchf Column WKN1)∨From Move q1=Makesquare(
Wsucf Row WKN1,R2touchf Column WKN1)))))))) (364)

*****∃E ↑↑ q1;
365 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(To Move q1=WKR1∧(VALUEP
Val(Prevpos q1,BKRP)∧((¬(Row WKR1=6)∧¬(Row WKR1=4))⊃((From Move q1=
Makesquare(Wsucf Row WKR1,Column WKR1)∧Pos(Prevpos q1,WKR1)=EMPTY)∨((Taken
Move q1=Pos(Prevpos q1,WKR1)∧WHITEPIECE Pos(Prevpos q1,WKR1))∧(From Move q1=
Makesquare(Wsucf Row WKR1,L2touchf Column WKR1)∨From Move q1=Makesquare(
Wsucf Row WKR1,R2touchf Column WKR1)))))))) (365)

*****simplify ↑↑;
366 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(To Move q1=WKN1∧(VALUEP
Val(Prevpos q1,BKRP)∧((From Move q1=WKN2∧Pos(Prevpos q1,WKN1)=EMPTY)∨((Taken
 Move q1=Pos(Prevpos q1,WKN1)∧WHITEPIECE Pos(Prevpos q1,WKN1))∧(From Move q1
=WKB2∨From Move q1=WKR2)))))) (364)

*****simplify ↑↑;
367 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(To Move q1=WKR1∧(VALUEP
Val(Prevpos q1,BKRP)∧((From Move q1=WKR2∧Pos(Prevpos q1,WKR1)=EMPTY)∨((Taken
 Move q1=Pos(Prevpos q1,WKR1)∧WHITEPIECE Pos(Prevpos q1,WKR1))∧(From Move q1
=WKN2∨From Move q1=WKN2)))))) (365)
```

Hence, the from square of either q1 was one of the three possibilities.

```
*****tauteq (PREDEGAME(q1 qx)∨q1=qx)∧Mover Move q1=BKRP∧VALUEP Val(Prevpos
*q1, BKRP)∧   (From Move q1=WKR2∨From Move q1=WKB2∨From Move q1=WKN2) ↑↑;
368 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(VALUEP Val(Prevpos q1,BKRP
)∧(From Move q1=WKR2∨(From Move q1=WKB2∨From Move q1=WKN2)))) (364)

*****tauteq (PREDEGAME(q1 qx)∨q1=qx)∧Mover Move q1=BKRP∧VALUEP Val(Prevpos
*q1, BKRP)∧   (From Move q1=WKR2∨From Move q1=WKB2∨From Move q1=WKN2) ↑↑;
369 (PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(VALUEP Val(Prevpos q1,BKRP
)∧(From Move q1=WKR2∨(From Move q1=WKB2∨From Move q1=WKN2)))) (365)
```

By existential quantification, we obtain the same WFF as a consequence of either (promotion square) assumption.

```
*****∃I ↑↑ q1;
370 ∃q1.((PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(VALUEP Val(Prevpos q1
,BKRP)∧(From Move q1=WKR2∨(From Move q1=WKB2∨From Move q1=WKN2))))) (333
```

```
360)

*****∃I ↑↑ q1;
371 ∃q1.((PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(VALUEP Val(Prevpos q1
,BKRP)∧(From Move q1=WKR2∨(From Move q1=WKB2∨From Move q1=WKN2))))) (333
361)
```

We know the promotion square to be either WKN1 or WKR1.

```
*****taut To Move qy=WKN1 ∨ To Move qy=WKR1 N1_OR_R1;
372 To Move qy=WKN1∨To Move qy=WKR1  (1 9 333)
```

Hence, the presumed position q1 certainly exists, regardless. We have used an uncommon dependency removing inference rule, *or elimination* to generate this step. Without ∨E, we would have needed an addition inference.

```
*****∨E ↑,↑↑↑,↑↑;
373 ∃q1.((PREDEGAME(q1,qx)∨q1=qx)∧(Mover Move q1=BKRP∧(VALUEP Val(Prevpos q1
,BKRP)∧(From Move q1=WKR2∨(From Move q1=WKB2∨From Move q1=WKN2))))) (1 9)
```

Let us call the position from which black promoted his pawn qz. We know that the From square of qz must be one of WKB2, WKN2 or WKR2.

```
*****label CALL_QZ;
*****∃E ↑ qz;
374 (PREDEGAME(qz,qx)∨qz=qx)∧(Mover Move qz=BKRP∧(VALUEP Val(Prevpos qz,BKRP
)∧(From Move qz=WKR2∨(From Move qz=WKB2∨From Move qz=WKN2))))  (374)
```

11.2.2. We notice that the WKBP has not yet moved. Hence, in qz, the WKBP was on WKB2.

```
*****∀E ShortPawnPathThm qz,qx,WKB2,WKB2,WKBP,QBUD;
375 ∀sq.((MAY_PAWN_CAPTURES(WKB2,sq,Piececolor WKBP)∧MAY_PAWN_CAPTURES(sq,
WKB2,Piececolor WKBP))⊃(sq=WKB2∨sq=WKB2))⊃((Pos(qx,WKB2)=WKBP∧(Pos(P0,WKB2)=
WKB2,Piececolor WKBP)∨qz=qx)∧(VALUEP Val(qx,WKBP)∨(BOARD(qx,QBUD)∧(Valueon
WKBP∧((PREDEGAME(qz,qx)∨qz=qx)∧(VALUEP Val(qx,WKBP)∨(BOARD(qx,QBUD)∧(Valueon
(QBUD,WKB2)=PW∨Valueon(QBUD,WKB2)=PB))))))⊃(Pos(qz,WKB2)=WKBP∨Pos(qz,WKB2)=
WKBP))

*****label ON_WKBP;
*****simplify ↑;
376 (Pos(qx,WKB2)=WKBP∧((PREDEGAME(qz,qx)∨qz=qx)∧(VALUEP Val(qx,WKBP)∨BOARD(
qx,QBUD))))⊃(Pos(qz,WKB2)=WKBP∨Pos(qz,WKB2)=WKBP)
```

A similar statement can be made about the WKRP. It, too, was on WKR2 in qz.

```
*****∀E ShortPawnPathThm qz,qx,WKR2,WKR2,WKRP,QBUD;
377 ∀sq.((MAY_PAWN_CAPTURES(WKR2,sq,Piececolor WKRP)∧MAY_PAWN_CAPTURES(sq,
WKR2,Piececolor WKRP))⊃(sq=WKR2∨sq=WKR2))⊃((Pos(qx,WKR2)=WKRP∧(Pos(P0,WKR2)=
WKRP∧((PREDEGAME(qz,qx)∨qz=qx)∧(VALUEP Val(qx,WKRP)∨(BOARD(qx,QBUD)∧(Valueon
(QBUD,WKR2)=PW∨Valueon(QBUD,WKR2)=PB))))))⊃(Pos(qz,WKR2)=WKRP∨Pos(qz,WKR2)=
WKRP))

*****label ON_WKRP;
*****simplify ↑;
378 (Pos(qx,WKR2)=WKRP∧((PREDEGAME(qz,qx)∨qz=qx)∧(VALUEP Val(qx,WKRP)∨BOARD(
qx,QBUD))))⊃(Pos(qz,WKR2)=WKRP∨Pos(qz,WKR2)=WKRP)
```

But the From square of any move is empty immediately subsequent to that move. Hence, neither of these squares was the source square of the move of qz.

```
*****VE EmptyFrom qz WKBP WKB2;
379 Pos(qz,WKB2)=WKBP⊃¬(WKB2=From Move qz)

*****VE EmptyFrom qz WKRP WKR2;
380 Pos(qz,WKR2)=WKRP⊃¬(WKR2=From Move qz)
```

11.3. Hence, the From square ofqz must have been WKN2.

```
*****label FROM_QZ;
*****tauteq From Move qz=WKN2 CALL_QZ,ON_WKBP,ON_WKRP,↑↑:↑,ROW2_WP,QX_QBUD;
381 From Move qz=WKN2  (1 9 374)
```

Section 4.2.12 The Route to BKN7

And, as the From square of qz was WKN2, there must have existed yet another position, (we will call it py) for in which BKRP, pawn valued, was on WKN2.

```
*****VE PawnWasOnThm qx,qz,BKRP,WKN2;
382 ((PREDEGAME(qz,qx)∨qz=qx)∧(VALUEP Val(Prevpos qz,BKRP)∧(Mover Move qz=
BKRP∧(From Move qz=WKN2∧¬(Pos(P0,WKN2)=BKRP)))))⊃∃p.((Pos(p,WKN2)=BKRP∧(
PREDEGAME(p,qx)∧VALUEP Val(p,BKRP)))∧VALUEP Val(Prevpos p,BKRP))

*****simplify ¬(Pos(P0,WKN2)=BKRP);
383 ¬(Pos(P0,WKN2)=BKRP)

*****taut ↑↑:#2 CALL_QZ,FROM_QZ:↑;
384 ∃p.((Pos(p,WKN2)=BKRP∧(PREDEGAME(p,qx)∧VALUEP Val(p,BKRP)))∧VALUEP Val(
Prevpos p,BKRP))  (1 9)

*****label CALL_PY;
*****∃E ↑ py;
385 (Pos(py,WKN2)=BKRP∧(PREDEGAME(py,qx)∧VALUEP Val(py,BKRP)))∧VALUEP Val(
Prevpos py,BKRP)  (385)
```

And, similarly, a move that got him there.

```
*****VE BlackPawnMoveThm qx,py,BKRP,WKN2;
386 (Pos(py,WKN2)=BKRP∧(¬(Pos(P0,WKN2)=BKRP)∧((PREDEGAME(py,qx)∨py=qx)∧Val(
Prevpos py,BKRP)=PB)))⊃∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To
Move q=WKN2∧(VALUEP Val(Prevpos q,BKRP)∧((¬(Row WKN2=6)∧¬(Row WKN2=4))⊃((
From Move q=Makesquare(Wsucf Row WKN2,Column WKN2)∧Pos(Prevpos q,WKN2)=EMPTY
)∨((Taken Move q=Pos(Prevpos q,WKN2)∧WHITEPIECE Pos(Prevpos q,WKN2))∧(From
Move q=Makesquare(Wsucf Row WKN2,L2touchf Column WKN2)∨From Move q=
Makesquare(Wsucf Row WKN2,R2touchf Column WKN2)))))))))

*****VE PawnValuedBlackPieces Prevpos py,BKRP;
387 VALUEP Val(Prevpos py,BKRP)⊃Val(Prevpos py,BKRP)=PB

*****tauteq ↑↑:#2 CALL_PY-2,CALL_PY:↑;
388 ∃q.((PREDEGAME(q,qx)∨q=qx)∧(Mover Move q=BKRP∧(To Move q=WKN2∧(VALUEP
Val(Prevpos q,BKRP)∧((¬(Row WKN2=6)∧¬(Row WKN2=4))⊃((From Move q=Makesquare(
Wsucf Row WKN2,Column WKN2)∧Pos(Prevpos q,WKN2)=EMPTY)∨((Taken Move q=Pos(
```

```
Prevpos q,WKN2)∧WHITEPIECE Pos(Prevpos q,WKN2))∧(From Move q=Makesquare(
Wsucf Row WKN2,L2touchf Column WKN2)∨From Move q=Makesquare(Wsucf Row WKN2,
R2touchf Column WKN2))))))))) (1 9)
```

We call that position, pz.

```
*****label CALL_PZ;
***** ∃E ↑ pz;
389 (PREDEGAMF(pz,qx)∨pz=qx)∧(Mover Move pz=BKRP∧(To Move pz=WKN2∧(VALUEP
Val(Prevpos pz,BKRP)∧((¬(Row WKN2=6)∧¬(Row WKN2=4))⊃((From Move pz=
Makesquare(Wsucf Row WKN2,Column WKN2)∧Pos(Prevpos pz,WKN2)=EMPTY)∨((Taken
Move pz=Pos(Prevpos pz,WKN2)∧WHITEPIECE Pos(Prevpos pz,WKN2))∧(From Move pz=
Makesquare(Wsucf Row WKN2,L2touchf Column WKN2)∨From Move pz=Makesquare(
Wsucf Row WKN2,R2touchf Column WKN2)))))))) (389)
```

Applying the theorem that sees the possible source squares for a given pawn and square, we get that BKRP reached this square from one of WKB3, WKN3 or WKR3.

```
*****simplify ↑;
390 (PREDEGAME(pz,qx)∨pz=qx)∧(Mover Move pz=BKRP∧(To Move pz=WKN2∧(VALUEP
Val(Prevpos pz,BKRP)∧((From Move pz=WKN3∧Pos(Prevpos pz,WKN2)=EMPTY)∨((Taken
 Move pz=Pos(Prevpos pz,WKN2)∧WHITEPIECE Pos(Prevpos pz,WKN2))∧(From Move pz
=WKB3∨From Move pz=WKR3)))))) (389)
```

11.4. Now, we note that the WKNP, on the third row, has spent the entire game on the squares WKN2 and WKN3.

```
*****VE ShortPawnPathThm pz,qx,WKN3,WKN2,WKNP,QBUD;
391 ∀sq.((MAY_PAWN_CAPTURES(WKN2,sq,Piececolor WKNP)∧MAY_PAWN_CAPTURES(sq,
WKN3,Piececolor WKNP))⊃(sq=WKN2∨sq=WKN3))⊃((Pos(qx,WKN3)=WKNP∧(Pos(P0,WKN2)=
WKNP∧((PREDEGAME(pz,qx)∨pz=qx)∧(VALUEP Val(qx,WKNP)∨(BOARD(qx,QBUD)∧(Valueon
(QBUD,WKN3)=PW∨Valueon(QBUD,WKN3)=PB))))))⊃(Pos(pz,WKN3)=WKNP∨Pos(pz,WKN2)=
WKNP))

*****simplify ↑;
392 (Pos(qx,WKN3)=WKNP∧((PREDEGAME(pz,qx)∨pz=qx)∧(VALUEP Val(qx,WKNP)∨BOARD(
qx,QBUD))))⊃(Pos(pz,WKN3)=WKNP∨Pos(pz,WKN2)=WKNP)
```

11.5. In the move that brought BKRP to WKN2 (pz) he was certainly on the latter.

```
*****VE MoverOnTO pz,WKNP,WKN2;
393 (Pos(pz,WKN2)=WKNP∧To Move pz=WKN2)  .  =Mover Move pz
```

11.6. And, as the From square of any move is subsequently (immediately) empty, and WKNP was on WKN3, then the from square of the move qz must have been either WKR3 or WKB3.

```
*****VE EmptyFrom pz,WKNP,WKN3;
394 Pos(pz,WKN3)=WKNP⊃¬(WKN3=From Move pz)
```

12. But either of these squares implies the capture of a white piece on the white square, WKN2. This piece must, of course, have been one of the white pieces eligible for capture.

```
*****VE WHICH_QX_TAKEN pz,Pos(Prevpos pz,WKN2),WKN2;
395 CHESSPIECES Pos(Prevpos pz,WKN2)⊃(((PREDEGAME(pz,qx)∨pz=qx)∧(Taken Move
```

```
pz=Pos(Prevpos pz,WKN2)∧WHITEPIECE Pos(Prevpos pz,WKN2)))⊃((Pos(Prevpos pz,
WKN2)=WQ∨(Pos(Prevpos pz,WKN2)=WQN∨(Pos(Prevpos pz,WKN2)=WKN∨(Pos(Prevpos pz
,WKN2)=WQB∨(Pos(Prevpos pz,WKN2)=WKB∨Pos(Prevpos pz,WKN2)=ywr1)))))∧(¬(Pos(
Prevpos pz,WKN2)=yyw)∧((To Move pz=WKN2∧WHITESQUARES WKN2)⊃¬(Pos(Prevpos pz,
WKN2)=WQB))))) (1 9 252 286)

*****simplify WHITESQUARES WKN2∧¬(WKNP=BKRP);
396 WHITESQUARES WKN2∧¬(WKNP=BKRP)
```

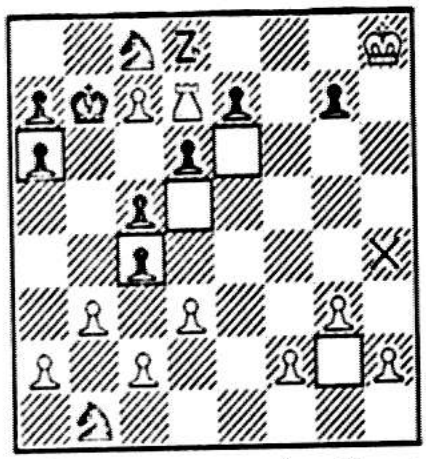
Let us refer to the white piece captured on WKN2 in pz as Pos(Prevpos pz, WKN2). This was certainly a CHESSPIECE (only chesspieces are ever captured).

```
*****simplify ∀p.CHESSPIECES Taken Move p;
397 ∀p.CHESSPIECES Taken Move p

*****∀E ↑ pz;
398 CHESSPIECES Taken Move pz
```

12.1. Hence, it must have been one of the white traveling white officers, and not the fallen piece.

```
*****tauteq CHESSPIECES Pos(Prevpos pz,WKN2)∧
*     Taken Move pz = Pos(Prevpos pz WKN2)∧
*     (Pos(Prevpos pz,WKN2)=WQ∨ Pos(Prevpos pz,WKN2)=WQN∨
*       Pos(Prevpos pz,WKN2)=WKN∨ Pos(Prevpos pz,WKN2)=WKB∨
*       Pos(Prevpos pz,WKN2)=ywr1)∧
*     ¬Pos(Prevpos pz,WKN2)=yyw  CALL_PZ+1;↑↑↑,↑,ROW3_WP,QX_QBUD;
399 CHESSPIECES Pos(Prevpos pz,WKN2)∧(Taken Move pz=Pos(Prevpos pz,WKN2)∧((
Pos(Prevpos pz,WKN2)=WQ∨(Pos(Prevpos pz,WKN2)=WQN∨(Pos(Prevpos pz,WKN2)=WKN∨
(Pos(Prevpos pz,WKN2)=WKB∨Pos(Prevpos pz,WKN2)=ywr1))))∧¬(Pos(Prevpos pz,
WKN2)=yyw))) (1 9 252 286 389)
```

We need also point out that these are five different pieces.



White officers were captured on these squares.

figure 44

```
*****VE DifferentTakenFour qx,pz,p1,p2,p3,p4,Pos(Prevpos pz,WKN2),xa,xb,xc,
*xd;
400 CHESSPIECES Pos(Prevpos pz,WKN2)⊃(((pz=qx∨PREDEGAME(pz,qx))∧((p1=qx∨
PREDEGAME(p1,qx))∧((p2=qx∨PREDEGAME(p2,qx))∧((p3=qx∨PREDEGAME(p3,qx))∧((p4=
qx∨PREDEGAME(p4,qx))∧(¬(Mover Move p1=Mover Move pz)∧(¬(Mover Move p2=Mover
Move pz)∧(¬(Mover Move p3=Mover Move pz)∧(¬(Mover Move p4=Mover Move pz)∧(
Taken Move pz=Pos(Prevpos pz,WKN2)∧(Taken Move p1=xa∧(Taken Move p2=xb∧(
Taken Move p3=xc∧Taken Move p4=xd)))))))))))⊃(¬(xa=Pos(Prevpos pz,WKN2))∧(
¬(xb=Pos(Prevpos pz,WKN2))∧(¬(xc=Pos(Prevpos pz,WKN2))∧¬(xd=Pos(Prevpos pz,
WKN2))))))

*****tauteq ↑:#2#2 ↑↑:↑,CALL_PZ+1,CALL_PN:CALL_PN+3,DIFFMOVERS;
401 ¬(xa=Pos(Prevpos pz,WKN2))∧(¬(xb=Pos(Prevpos pz,WKN2))∧(¬(xc=Pos(Prevpos
 pz,WKN2))∧¬(xd=Pos(Prevpos pz,WKN2))))  (1 9 310 311 312 313 389)
```

13. Hence, by the usual counting argument, the only piece that could have fallen from the board was the white queen's bishop.

```
*****tauteq yyw=WQB ↑,↑↑↑,WHICH_YYW,WHO_XA:WHO_XA+3,NOT_XN_EQ;
402 yyw=WQB  (1 9 252)

*****tauteq Pos(px WKR4)=WQB CALL_YYW,SAME_ON_WKR4,↑;
403 Pos(px,WKR4)=WQB  (1)
```

Removing dependencies, and generalizing, we see that, as a consequence of our chess axioms, if a chesspiece fell from WKR4 in a position which had GIVEN as a board, that chesspiece must have been the white queen's bishop, *quod erat demonstrandum*.

```
*****⊃I CALL_PX⊃↑;
404 (BOARD(px,GIVEN)∧CHESSPIECES Pos(px,WKR4))⊃Pos(px,WKR4)=WQB

*****label THE_THEOREM;
*****∀I ↑ px;
405 ∀px.((BOARD(px,GIVEN)∧CHESSPIECES Pos(px,WKR4))⊃Pos(px,WKR4)=WQB)
```

Chapter 5 Conclusions

Section 5.1 Perspective

We have here a mass of verbiage and proof. It is certainly important to step back and, in perspective, assess just what we have learned in its generation.

Let us reiterate: we have taken a difficult problem of retrograde analysis chess, detailed a set of axioms for the rules of chess, and have proven the solution to that puzzle within our axiom system. While not modeling the human process of proof discovery, we have modeled the human ability to *accept* a valid proof. That is, our FOL proof parallels and corresponds to the human proof, particularly in two important dimensions. We have, in FOL, been able to model both the ability to accept and structure inference (the basic deduction framework), and the ability to jump to the immediate conclusions of observation (our *chess eye*). We are exploring the *nature* of (adequate) reasoning sequences, rather than finding the (appropriate) heuristics for generating such sequences.

We also need to stress what we haven't done. We have not presented a program which would, in any sense, model the way the human intelligence arrives at the proof. Such a system would need elements of intuition and search, in addition to ability to correctly perform inference steps and computations. Like almost all proofs, our chess proof gives little explanation as to *why* some step was taken (other than that it worked); no dead ends or useless inferences litter the way.

Adequately modeling the human ability to generate a proof is an extremely difficult problem, essentially equivalent to solving (much of) the A.I. problem itself. Presenting a solution *acceptable* both to a human and a machine was, in itself, a hard problem. In a strong sense, being able to *accept* correct reasoning is a prerequisite for general intelligence. We do not foresee solution of the more difficult problem, that of a general computer intelligence, in the near future. Rather, we view examinations of representational systems (such as this paper) to be part of the (long) process of achieving the necessary understanding to eventually create an artificial intelligence.

Let us also emphasize that we are not, of course, asserting that the solution of the fallen piece problem reveals all aspects of knowledge and representation. We have been examining in this proof only several issues, particularly the interactions and interfacings of deduction and observational computation. This is by no means adequate for a thorough representational system. We have dealt in a highly structured and complete domain. We have not touched upon many modalities (knowledge, belief, desire) that a truly intelligent program would need to manipulate. Our expression of events (moves) and time, the relationship between positions in the same game) while useful and revealing, is that of a discrete system, not a general continuum. There are certainly many properties required of generally intelligent systems that we are not even aware of, and will not perceive the need for until we stumble into them.

Section 5.2 Representation and this Proof

One of the more interesting facets of this investigation is the comparison and selection of the various representational devices employed in our chess axioms.

Representational choices are based upon two primary criteria. We want that our representation should be *convenient*. We should be able to express (as easily as possible) the range of expected

problems and solutions within the model. Our representation must, however, retain integrity with respect to the problem domain. We are not interested in seeing how we can pervert the original problem into another, more tractable (though equivalent) domain. Rather, we must represent the *given* problem.[58]

Perhaps, while we are discussing natural representations, a pair of examples from our chess world would be appropriate. When chess pieces are captured, they cease, (in some strong sense) to exist. There is no *square* which we can point to, saying, that *piece* is on that *square*. Captured pieces vanish without a trace. Most theorems about pieces and squares must therefore begin: *if a piece* x *is on a square* sq *in a position* p *then* ... . Imagine instead that a captured piece merely changed its value, and became a *ghost*, nevertheless retaining reference to its capture square. Our axioms and proof would then be much simpler. Every piece would have a square of its own. Additionally, a position could reference those pieces captured in reaching it by pointing to the ghosts on various squares, rather than creating a hypothetical ancestor position in which they had been captured, and reasoning about that position (as we do now). Our counting arguments (most of the last hundred steps of the main proof) would then be much briefer.

Consider secondly, the notion of value and piece (which we will explore in greater detail further on). Let us now merely point out that the king pieces and the empty piece have unique and constant values (we have several theorems to this effect: see, for example, *KingValueThm1* and *EmptyIsMT*). But these theorems could be dispensed with, and several proofs reduced several steps, if we were to blur the distinction between VALUE and PIECE, and assert, for example, that BK=KB[59] and EMPTY=MT.[60] What would result would be (slightly) smaller but less natural proofs. It is not that it would be *wrong* to axiomatize in this fashion, so much as unpleasing.[61]

In the following subsections, we will examine some of the more interesting representational decisions embodied in our chess axioms.

Section 5.2.1 State Variables and Computable Objects

The major representational dichotomy in this system is the balance between POSITIONS, a state vector containing all of the information required to reconstruct a particular game (perhaps a list or moves or boards), and BOARDS which is a (concrete) representation of (most of) the current status of a game.

A passing glance at chess would reveal the necessity for the latter, though, presumably, not the former. After all, chess problems are (typically) presented in terms of chess boards, not as the entire game played to reach some position. Similarly, (almost all) chess moves are defined in terms of a chess board; this rook can move so, regardless of what line he used to reach his square, or which

---

58 It goes almost without saying, of course, that the representation must be correct (we must really be solving the problem). In most domains, generality is a desirable attribute; to be aesthetically pleasing, the selected axiom system should be able to express more than the limited issue at hand.

59 The piece BK is the same as the value KB.

60 After all, aesthetics is an issue of taste.

61 If we were embody this notion within our axiomatization, and to later seek to analyze problems where pawns could promote to kings, this simplification would get in our way.

square he began the game upon.[62]

One does not become *really aware* of the necessity for the state variable, (what we have called the *position*) until one approaches retrograde analysis. We frequently refer to (for example) the identity of a particular piece (which pawn was it in the opening?), to captures and moves of the game that reached some arrangement, and to the path some piece traveled. These notions are naturally those of the position, not inherent to a particular board. Many different games can be played to reach a given chess board; therefore, these are not aspects of the board *per se*.

The importance (in retrograde analysis) of this sort of temporal reasoning is reflected in the axioms by the predominance of the POSITIONS over the BOARDS (and over everything else). Rules are typically defined in terms of their effect on the *state of the world* (position), rather than their local effect on the playing board. Boards are employed almost exclusively for defining and computing the local moves of the various chess values. Thus, the predominant predicate for positions becomes SUCCESSOR, defining the (legal) transitions from position to position; for boards, MOVETO, expressing the local, legal paths of the various values. The basic movement consequence axioms begin at the positional level, only to descend to boards when considering the actual move.

The concepts of board and position are tied together in a predicate and a function. The function Tboard (total-board) extracts the board that would result from playing out a given position. The predicate BOARD is true when its second argument is either the Tboard of its first, or a less defined board (section 2.1.5).

Within the concept of observation and inference, this position and board dichotomy has further significance. Positions, as expressed in these axioms, are an elusive, intangible concept. There is nothing we can point to and say: "that is the position of interest". Rather, positions are the child of the inference scheme; we never (except the initial position) observed something to be true of a particular position. Boards, on the other hand, are concrete objects. The observations (computations) on boards are more important than the deductions applied to them. Each board has a distinct LISP model representation; they are the primary vision of the *chess eye*.

In retrospect, this separation into state variables and computable objects seems to have been a good decision. The problem would have been very intractable without the coherence provided by the state selectors. Similarly, *Chess induction* (Sections A.2 and 2.2.4.1) has proven to be a very useful and unifying concept, alien to the temporalities of a pure-board approach. The ability to compute on board representations has resulted in tremendous reduction in the total inference required.

Early in this research, there existed a distinction between *legal positions* and ordinary positions. Legal positions were those that (presumably) could be reached in a legal chess game. After the first iteration of proof, we observed that, essentially, we never proved anything about the illegal positions. The distinction between the two was then deleted from the axioms. On reflection, we find a parallel between those "illegal" positions, and several of the other unused sorts (such as EXSQUARES[63]). If we were to use these axioms in a *forward* direction (as opposed to this retrograde example) to create legal successors to a given position, we would probably axiomatize the "Nextpos" function (section 2.1.7.1), which would take a position and a move, and return the position resulting from making that

---

62 Castles, en passant captures, and various draw demand circumstances are exceptions to this rule.

63 Those squares the captured pieces occupy. For example, the function Pospcf returns an element in the domain of EXSQUARES

move in that position. This general function would not be obliged to return a legal position (and would not, if not referencing a legal move). Hence, the range of this function would therefore be declared to be on all positions, not merely the legal ones. The earlier impulse towards *legal* positions is therefore seen as an anticipation of this extension.

Positions, as described, are virtually not expressible within the model space; representational systems that depend to heavily upon doing model computation as the only inference mechanism will be unable to deduce results of the complexity of our given problem.

Section 5.2.2 Incompletely Defined Objects

Another perspective illuminated by the distinction between positions and boards is that of partially defined objects. That is, we need a mechanism for expressing predicates about objects not all of whose features are known to us.

There are two different kinds of *partial definition* which we consider here. The first is illustrated by the *positions* sort. Positions are fully defined, in that, any question we might have about a position can be answered by examining that position. This may seem paradoxical. After all, we never know anything about any position until we infer it. We resolve this paradox by never having any "real" positions.[64] Rather, all statements about positions are of the form *Assume we have a position with the following properties...* . Notice that there are no positional constants; only positional variables (and parameters). We perform no observational computations upon positions. And we have no explicit partial positions. Rather, an entire game can be replayed from any position.

Boards, on the other hand, are concrete objects. We want our LISP functions to be able to manipulate these objects. Within the current structure of FOL, this is possible only if the object is a *constant*. But we are confronted immediately, in the very problem statement, with a *variable* board, our problem being to complete the definition of the given, partially defined board.

There are only twelve different chess values. Clearly, one possible stratagem would be to consider each of the twelve possible totally defined boards, and prove that only only one of them could have arisen in a legal chess game. This approach fails, however, to satisfy both esthetic and practical considerations. Aesthetically, we are examining *reasoning*, and seek to handle more than simple case analysis. We certainly do enough of that in the rest of the proof. Practically, these case considerations can grow exponentially with the depth of analysis. If each possible board spawns a board with two more unknown squares, we soon have the cube of twelve cases to consider. Each consideration is likely to be a fair sized proof in itself. And this method will flounder on any consideration of unbounded sets.

Rather, we surmount this obstruction by the introduction of an *undefined* constant, to be inserted in the board structure whenever the value of a particular square is unknown. While this is a clever and transparent solution of the immediate problem, it has ramifications throughout the entire axiom structure. Most obviously, values on boards and values in positions are no longer trivially identical. Rather, that equality is conditional on the board being defined on that square.[65] This is usually painfuily obvious, but demands another step. Not, however, a terrible penalty. Greater confusion

---

64. This subsequent discussion rightfully ignores P0, the initial position.

65. That is, Val(p Pos(p sq))=Valueon(b sq) if ~Valueon(b sq)=UD (and BOARD(p b)) (this is the axiom VALUETRANSPOSITION).

arises, however, in the cases of more complicated predicates. What should the value of MOVETO be, say, if an undefined square blocks the way? It is certainly not true, but, in another sense, is not really false. That is, we would (sometimes) like to use MOVETO to show some move impossible; other times, to demonstrate (with the appropriate assignation of values to the undefined squares) that such a move could be accomplished. The solution adapted in this axiomatization is to make MOVETO demand a fully demonstrated possible move. Various theorems, such as *TransitiveSubboardOrthogonality* (section A.9.2.2) and *DiagonalThm* (section A.9.3.1) relate movement on partially defined boards to that on more complete boards.

An alternate possibility was not employed. One can easily imagine, within the present axiom structure, predicates such as MIGHT_MOVETO and MIGHT_ORTHO, which would be true if, say, the squares on the move's path were either empty or undefined (MT or UD) instead of only explicitly empty. Such predicates might simplify the definitions of several of the movement axioms, but complicate the translation to the more precise forms.

Section 5.2.3 Representation of Aspects

What may seem, perhaps, the most aberrant distinction embodied in these axioms is that between piece and value. Pieces, we recall, embody the identity of each of the thirty two chessmen, including, particularly, their initial squares. Values, on the other hand, are a reflection of the rank of an individual piece at a given point in the game. In playing chess, the names of the particular pieces are never invoked. Rather, the current value of any piece is adequate for determining its available moves. For the naive player, experience with chess comes from playing chess games, not from solving chess puzzles. Additionally, except for the rare occasion when a pawn has promoted, pieces do not change their value. Only in concocted retrograde analysis chess problems, is the path a piece followed important. Only in puzzles does one see such a bizarre collection of promotions. Therefore, perhaps, only to experienced puzzle solvers is the importance of this distinction obvious.

Let us point out that this is not an entirely happy arrangement, even though it is a necessary one. We need shave our lemmas and theorems to a tight tolerance of their intended use, matching piece and piece, value and value, unless we are willing to expend precious steps demonstrating to the machine (again) that this particular bishop does in fact have bishop value. Additionally, this equivalence, performed so naturally and immediately by the human, requires theorem invocation in the proof. We take consolation, however, in noting that the human tendency to jump to the conclusion that any officer (especially a non-queen officer) is not a promoted pawn is avoided by this deductive approach. Within a formal logic framework such as FOL, that rook valued piece is as likely to be a promoted pawn as one of the original rooks. It seems that any system wishing to generate a solution to real problems must rely heavily on grabbing the immediate, almost obvious device [McCarthy79b]

Some of the trouble associated with the Val function could have been avoided. It is not a *necessary* operator, the same result being indicated for pieces still uncaptured by the corresponding valueon the appropriate board. That is, Val (p x) = Valueon (Tboard p, Pospcf (p x)). This would result in a clean partition between piece and value, along the same line that divides position and board. It would, however, result in a larger and more cumbersome proof, as the translation, and its preconditions, would need frequent justification. Hence, we see Val to be a simplifying function, a short expression of a common notion.

Section 5.2.4 Expanding the Vision of the Chess Eye

Along with the more obvious (or, at least, having selected the system framework, obvious) functions and predicates of our system, we note several more creative and intrusive functions and predicates. These functions and predicates serve two functions. In some cases the predicate is of a definitional nature. That is, it is a short expression of a frequently invoked notion. An example of this definitional form is the predicate PROMOTEDPAWN. This predicate could be dispensed with by substituting its definition (axiom *MCONSEQI*) for each of its occurrences, a mechanical process. Its sole value lies in providing economy of expression.

This use of definitional predicates is a common device in first order logic, and deserves no further comment. More interesting are the constructive functions of this axiomatization, such as the Unmkmove and SQUARE_BETWEEN operators. They different from the simpler definitional axioms, and from all conventional logic definitions, in that they have associated *attachments* in the chess eye.

Consider the example of the Unmkmove functions, which take a board and a move, and return the board of the previous position. We are performing here what is (for a human) an essentially mechanical and observational task. However, to do the same work in a purely inferential framework requires both the declarations of another individual board, and a quantification check of the essential identity of that board, and the original board, on all of the uninvolved squares. Much effort is saved through the pure computation. Or, in the local colloquial, it's a winner.

What we must catalogue here instead is a pair of retrospective regrets. For one thing, a regret at not using this device to greater advantage. A second regret at the limits of the application of this device in our present FOL system. As the chess eye is limited to computing on constants, there is no mechanism for computing on the known properties of parametric objects, other than the clumsy use of a constant *undefined*. We will consider this regret in greater detail in section 5.8..

Section 5.2.5 Other Natural and Unnatural Notions

We conclude with a few additional comments on several of the minor sorts mentioned in section 2.1.1.

Many of the declarations and much of organization of this proof is devoted to simplifying the inference process. However, we must report that such simplification has not been pursued at the cost of sacrificing aesthetic values. An example of this devotion is the sort MOVES. It is a very common notion to speak of, for example, the move that reached this position, or the possible moves available in this position, or of the move that brought some piece to some square. Hence, the sort of MOVES, and the function Move, which extracts the last move made to reach its argument position.

However, careful examination of the entire proof reveals that never is a move referred to, except to speak of the move of a position.[66] Each of the common functions on a move, such as From and Mover is invariably invoked on the Move of some GAMEPOSITION. The proof would be somewhat simplified by the deletion of the MOVES predicate. However, the aesthetic criteria (it is, after all, a natural notion) demand its retention.

Perhaps one of the most obvious sorts is that of COLORS. After all, the combat of the black and

---

66 A slight exception occurs here with respect to lemmas solely concerned with the structure of the move hierarchy (such as MOVETYPES). However, like the major uses of moves, the use of a "sort" of moves is not required here, either.

white armies is fundamental metaphor of the game. But even within this natural division, there remains room for choice. It is convenient to have one's functions always evaluate to some value. We speak, for instance, of the Piececolor WK as WHITE. What then should the Piececolor EMPTY be? We considered introducing GREY, the color of the piece on any empty square. But, once again, this can hardly be defended as a natural notion. Secondly, and perhaps more importantly, it is not clear that having a GREY would serve to reduce the size of proofs.

Even as obvious a sort as the squares of the chessboard requires some decisions. We did not originally perceive the need for referencing the coordinates of squares (rows and column) at all. Later, as we needed to squeeze proofs where simplification could not carry us, these sorts became required. It is clear that we do not want to depend solely on coordinate pairs, however. Most square references need be only to fixed squares. The differences between rows and columns in the axioms could have been deleted, at the cost of a slight increase in incomprehensibility, and a slight decrease in length of proofs. Of course, if these axioms were to be used in situations requiring more algebraic manipulation of row and column values, the definitions of these sorts would require suitable expansion.

Section 5.3 Alternatives

So far, we have been examining the "micro" decisions involved in generating these axioms, considering choices from within our selected framework. While we believe that the representations chosen have been generally appropriate, it is still worthwhile to consider the consequences of various alternate choices.

Section 5.3.1 Levels of Axiomatization

Elsewhere in this paper (chapter 2) we spoke briefly about the choice of *level* of the axiomatization. Let us reiterate on that notion.

Almost any large mathematical proof can be made arbitrarily easy or difficult by the selection of the initial axiom structure. The situation in the chess world is essentially similar. For example, if we had taken all of the lemmas in appendix A as theorems (all of them are "facts" obvious to any experienced puzzle solver), this paper would be considerably smaller. Even beyond merely presentation of multiple lemmas, it should be possible to restructure the problem so that it is no longer a formal proof, but, rather, the sequential application of various "rules" for the solution of chess puzzles. But certainly, the more specific and useful the given rules to this particular problem, the less capable they would be of expressing other kinds of chess puzzles.

We could, of course, have proceeded in the opposite direction, defining, for example, the various piece movements as mathematical relationships, and entangled ourselves in the mathematical structure when proving even a simple move. While there are certainly many things thereby expressible that are difficult to state in the present axiomatization, the resulting proofs might easily be an order of magnitude larger.

Perhaps the only moral to this section is that one can make any problem arbitrarily difficult (and most problems arbitrarily easy) by selecting a suitable starting place, the given conditions. And that the size of this paper, and the complexity of the proofs is a reflection of our opinion of the appropriate generality of our axioms. Though this "moral" may seem obvious, it is an important criterion in the evaluation of any intelligent computer system.

Section 5.3.2 Prior's Modal Tense Logic and Positions

We have not, of course, presented enough evidence to conclude that first order logic, even augmented by semantic procedural attachments, is a general enough scheme to express all of the representation issues our intelligent computer will ever need. It's probably not. Even within the context of first order logic, our system examines only a minute corner of the universe of systems.

One notable omission is the lack, in our system, of equivalents of the various modal operators. Our retrograde chess puzzle embodies complete knowledge; there is no issue of the *beliefs* of individuals (in fact, no individuals). While a forward (competitive) analysis might include operators referring to the desires and goals of the players, our backwards attention precludes even this.[67]

Perhaps the one parallel to modal systems we can draw is to modal *tense* logics, for example, the modal tense logic of Prior ([Prior57], [Prior68])

Simply stated, Prior's system employes two modal operators, P and F, which signify *Past* and *Future*, respectively. Thus, for some proposition π, $P\pi$ states that π was true at some time in the past; similarly, $F\pi$ asserts π's occasioned future truth.

Now, as we deal with *retrograde* analysis, P is certainly the interesting operator. Thus, we might say, *if that pawn is on this square, then it is true that*, in the past, *that pawn captured an opposing piece on that square*. This may be contrasted with our present formulation of, *if that pawn is on this square, then there existed a position in the course of this game, for which the move of that position was a capture by that pawn on that square of an opposing chessman.*

Notice that our present notation is stating more than this modality. The hypothesis asserts not only the capture, but also presents us with the occasion (position) in which the capture occurred. More particularly, we can easily express in the present system anything asserted in the modal system. Thus, if there is to be any advantage to employing the modal P operator, it must come from permitting the deletion of some part of our present system. The obvious candidate for this elimination is our state vector, the position.

Now, by explicitly inventing the state where some proposition was true, we easily get both quantifications, *there existed a time when it was true*, and *it was always true*. Expressed in a modal form, these become *it was true in the past* ($P\pi$), and *there was no time when it was not true* ($\neg P\neg\pi$), somewhat clumsier, but still useable. Expressions of more complicated notion compound the complexity produced by the modal operator, on the other hand, there are a few situations where its employment would save a few steps.

Perhaps the major contrast between the current *positions* and the modal P operator is that the proposition asserted by the modal operator is one about the current situation; while the positional state vector makes a statement about a similar state vector, and then relates the two. This would be true even if the modal operator was defined upon a "board like" vector, rather than our present positions. As the axioms necessarily define attributes of states, they can easily be used to manipulate the resulting contrived state. Effectively, the current system gives a more particular individual to manipulate. A general moral of this research, echoed elsewhere (section 5.6.4) is that one is better off with a function that returns an individual, than a predicate presumed true about some less specified

---

67 This subject is touched upon in section 5.4.2

thing. But just as the predicate could work, our system could probably the transformed (kicking and screaming) into the modal form.

Section 5.3.3 Filling in the Blanks

A naive approach to this representation problem, particularly that of someone used to programing computers, and not considering the philosophical representation issues of artificial intelligence, would be what we call *the fill in the blanks* approach. This approach goes somewhat like this:

We have a situation (a board) as a problem. This board consists of sixty four squares. We "write" on each square whatever we know about that square. For example, in the given problem, we might state that the BQ2 square has some white rooked value, while the WQR4 square is unknown. We might have another table, that of the location of each piece (the white king is on BKR1), and so forth. Eventually, by manipulating the rules relating these tables, and filling in entries of the tables, we would arrive at our answer.

This approach bears a cursory resemblance to formal logic. The information contained in any table entry is simply expressible as a WFF of the predicate calculus. The table entry form is probably more convenient for heuristic manipulation. The programming table entry system differs from the proof approach (and resembles the planner-like languages) in that things can be both "true" and "false" at different points in the proof.

This system fails, however, in two important respects. For one, lacking the development and dependencies of the formal proof, it is difficult to express case analysis, a very important technique. While it is true that case selections can be made in this system by employing a recursive branching scheme, one might then discover that one is proving the same fact repeatedly for each of the different cases.[68]

More importantly, this simple scheme is unable to express first order facts about the chess world. Thus, while we could tell this system *Bishops always stay on the same color square*, (and have it use that rule its derivations), there is no way to derive or express that notion within the system.

We see that what we have here a confusion of a possible data structure (a representation) for an epistemology (another kind of representation).[69] We have inserted this straw man not so much as an example of a competitive system we wish to denigrate, but, rather, in the hope of clarifying the confusion surrounding the word representation as we have been using it in this paper.

Section 5.4 Our Representation Applied to Other Problems

So far, our attention has been concentrated on one specific example. It is worthwhile to examine how other problems would look in our formalism, without having to detail the entire proofs.

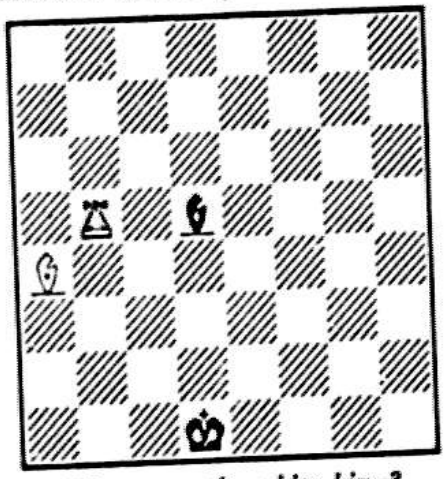
---

68 Case analysis here refers to considering each of the possible values a given unknown might have. Thus, in our original problem, there we twelve possible chess values for the fallen piece. In each of these cases, there are six possible values for the captured black piece, and so forth. Clearly, an unwieldy scheme.

69 We echo here some of the concerns of section 1.3.1.3

Section 5.4.1 Where was the King

Consider the following problem of retrograde analysis. We are presented with the board in *figure 45*, and told that the white king has fallen off; our problem is to determine his falling square. [70]



Where was the white king?

figure 45

This problem, while of similar retrograde form, differs in a very important respect from our earlier problem. Our earlier proof, and its axiom structure, are primarily concerned with almost completely defined boards. Here, too, we have an almost completely defined board. In the former problem, however, the undefined element was confined to a single square. Here we must contend with finding the undefined square.

Note that our earlier proof used, essentially, a list of squares and the pieces occupying them; here, we would prefer a list of pieces and their squares.

However, despite these difficulties, the problem is still tractable within our notation. We outline its solution.

The first step is, of course, to express the goal WFF in our formalism. Let the board of *figure 45* be called WHERE_KING. We know that there is some position, px, whose total board is the same as the board WHERE_KING, except that on some square sq, px is not empty, but rather contains the white king. We must therefore assume a WFF of the form:

$$\forall sq1.\ (Valueon(WHERE\_KING, sq1) = Valueon(Tboard\ px, sq1) \vee$$

$$Valueon(Tboard\ px, sq1) = KW)$$

That is, the total board of px agrees with WHERE_KING, except in those squares where the total board of px has a white king value.

We will be able to conclude a WFF of the form:

---

70 This problem is from [Gardner73]

$$Valueon(Tboard\ px, sq) = KW$$

where sq is the name of some individual square. (WQB3 in this case).

The proof first splits into three cases. Either the white king is on WQB2, WQN3, or some other square. We can easily prove the general chess theorem:

$$\forall px.\ \exists sq.\ Pos(px\ sq) = WK$$

that is, the white king is on some square in every (implicitly legal) position. We obtain a parameter for this square, let us call it sqx. Hence, it tautologically follows that:

$$sqx = WQB2 \vee sqx = WQN3 \vee (\neg sqx = WQB2 \wedge \neg sqx = WQN3)$$

It is a simple chess theorem to show that the two kings cannot coexist on neighboring squares. Hence, sqx is not WQB2.

$$\forall px\ b\ sq1\ sq2.((BOARD(px\ b) \wedge KINGMOVE(sq1\ sq2)) \supset$$

$$\neg(Valueon(b\ sq1) = KW \wedge Valueon(b\ sq2) = KB))$$

If the white king were on WQN3, then would be checked by both the black rook and bishop. Now, checks can occur only four ways (theorem *__CheckTypes__*). Black's last move was certainly not a castle, for his king is not on a castling square. There is no black pawn present to have just captured *en passant*. Therefore, for each check, either the checking piece made the last move for black, or the check was a discovered check. Since neither the bishop nor the rook could have move out of the other's way and given check, the situation is clearly impossible. Hence, the white king is not on either of these squares.

But then these squares must be empty, and the white bishop checking the black king.

$$Valueon(Tboard\ px,\ WQB2) = MT \wedge Valueon(Tboard\ px,\ WQN3) = MT$$

$$MOVETO(Tboard\ px,\ BW,\ WQR4,\ WQ1) \wedge Valueon(Tboard\ px,\ WQR4) = BW \wedge$$

$$Valueon(Tboard\ px,\ WQ1) = KB$$

It must be black's move.

Now, this bishop is cornered (section 3.4.2), unable to have moved to have created this check. Hence, white's last move must have taken a white piece out from between the bishop and the king.

$$SQUARE\_BETWEEN(WQR4,\ From\ Move\ px,\ WQ1) \wedge \neg\ Mover\ Move\ px = Pos(px\ WQR4)$$

But there is only one other available piece, the white king, to have made this discovery, and only two squares, (WQB2 and WQN3, again) between the bishop and the black king. The white king was certainly not on WQB2, as we have stated, kings are never in mutual check.

Therefore, the white king much be on WQN3 in Prevpos px. Now, we know that all the squares in Prevpos px have the same value as in px, except the To and From squares of that move. The From

square had the white king. The To square was either empty, or was occupied by a soon to be captured black piece.

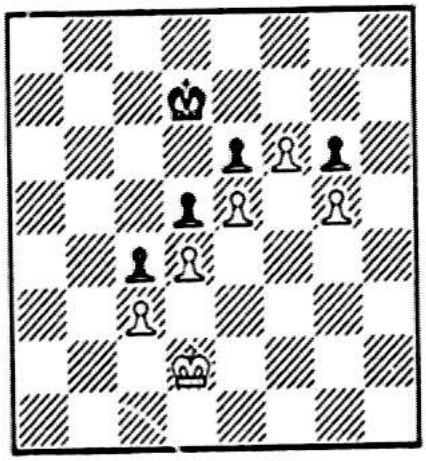
If the white king is on WQN3, a situation similar to the previous one arises. Black did not create the check by castling, nor did the bishop nor the rook move to cause that double check. But wait. The board in this position is not the same as the given board. We know that all of the squares have the same value, except the square to which the white king moved in generating the position px. This square could have contained a to be captured black piece, or, more specifically, the black pawn that has created this double check situation through an *en passant* capture. That pawn must have been on WQB3, and that must be the current square for the white king.

```
Valueon(Tboard px, WQB3) = KW
```

Section 5.4.2 Berliner's problem

Of course, the problem we argued in the last section is basically similar to sort of retrograde analysis for which these axioms were composed. Let us briefly consider then, how an entirely different sort of problem might be expressed in a suitable extension of this notation.

We consider board 1.7 from Berliner's thesis [Berliner74], the position diagrammed in *figure 46*.



Berliner's problem.

figure 46

Here the problem is of a different nature; rather than analyzing the ingredients that composed this

position, we instead have a more familiar task:[71] proving a *strategy* to lead white to victory.

What is essential here is expressing the notion that white can move his king around the pawn formation, and then to either capture the diagonal of black pawns, or promote his own. We expect some evaluation function to recognize that both of these are *won* positions.

Our current axiomatization obviously requires some extension before tackling this task. Our axioms look backwards; there is no expression that defines the legal successors of a position. Rather, we only restrict these successions. We hypothesize that suitable conditions from the appropriate ***MCONSEQ*** axioms (section 2.2.1.1) have been assembled into this definition, and that our simplifier easily recognizes the trivial cases of succession. We also hypothesize the simplification predicate WHITE_HAS_WON on some board or positional object, and a predicate on two positions, WHITE_CAN_ACCOMPLISH. WHITE_CAN_ACCOMPLISH(p1,p2) will be true if white can force a position with the properties of position p2, starting at position p1. We might have an axiom schema of the form:

$$\forall p1\ p2.((WHITE\_CAN\_ACCOMPLISH(p1\ p2) \wedge \neg WHITETURN\ p2 \wedge$$
$$\forall p3.\ (SUCCESSOR(p2\ p3) \supset \exists p4\ .((SUCESSOR(p3\ p4) \wedge \alpha\ p4\ ))))\ \supset$$
$$\exists p.\ (\alpha\ p \wedge WHITE\_CAN\_ACCOMPLISH(p1\ p) \wedge \neg WHITETURN\ p \wedge Prevpos\ Prevpos\ p=p2))$$

That is, we assume that p2 could be accomplished from a position p1, and p2 has black on move. For each of black's legal replies, p3, white has an answer, p4, for which some predicate α holds. It is therefore the case that there then exists[72] a p which white can reach from p1, is black's turn, and can be accomplished by white, is two moves after p2, and in which the predicate α is still true.[73]

It is fundamentally true that:

$$\forall p.\ WHITE\_CAN\_ACCOMPLISH(p,p)$$

that is, white can always accomplish the current state from the current state.

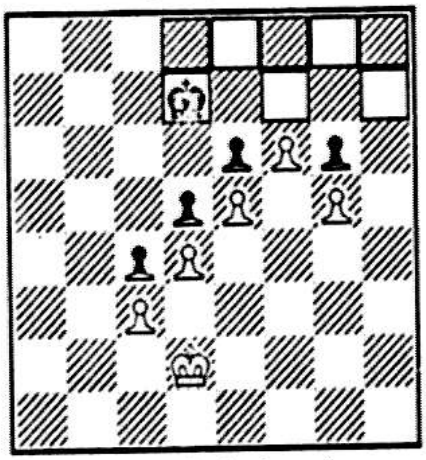
---

71 Or, at least to those whose experience with chess comes from playing it, familiar task.

72 This is, admittedly, a rather fanciful exists. What exists here is not so much a position, as a position for each possible response, all of which share some common properties (those indicated by the predicate parameter). However, as nothing can be proven about the abstract positions besides the information in the parameter, and the knowledge implicit in their specific common grandfather (p2), this device will succeed.

73 A similar axiom for white's turn may be formed by reversing the second clause's quantifiers.

We first establish that for any position with the given pawn structure, if the black king is not on one of the boxed squares in *figure 47*, then WHITE_HAS_WON.



The black king is limited to these squares.

figure 47

Let us call the given position px.

Now, by the hypothesized rules of WHITE_CAN_ACCOMPLISH, the successive predicates

$\exists p_i$. (WHITE_HAS_WON p_i ∨ (PREDEGAME(px, p_i)∧WHITE_CAN_ACCOMPLISH(px, p_i)∧
(Pos(p_i,BQ1)=BK∨... *(through each of the boxed squares)*) ∧
Pos(p_i,sq_i)=WK))

(where the sq_i range through the sequence WQB2, WQN2, WQR3, WQN4, BQB4) are all derivable.

Having brought the white king around to black's side, we could complete our proof by describing the little dance the monarchs engage as the white king pushes the black king away from the pawn on BK3. When the white king arrives at BQB4, either the black king is on BQ2, or some other of the boxed squares (or white has a won position). The case analysis continues for a few more ply, and is not very instructive.

We hope with these two examples that we have indicated that our axiomatization structure is general enough to express more that the single problem whose detailed solution we have presented.

Section 5.5 The Limitations of this Axiomatization

Of course, any statement about epistemological or heuristic approaches to A.I. ought to include a disclaimer cataloguing what that formalization is unable to solve or express.

We have are listing two different sorts of limitations; first, those places where our proof, as presented in chapter 4, fails to adequately model the human solution, and secondly, a consideration of our axiomatization's ability to handle various other sorts of chess problems.

Section 5.5.1 Difficulties Encountered in Generating this Proof

A comparison of the informal proof of section 1.6.2 and the FOL proof of chapter 4 shows the FOL proof to be substantially longer in handling two particular kinds of reasoning. A human puzzle solver can quickly check if a condition is satisfied by all pieces on the board. For example, the single human step 5.3.1, a check that none of the black pieces could have moved to discover check (if the captured piece, Z0, had been rook or queen), is transformed into steps 101-143 in the FOL proof. In simple cases, the quantification checking ability of FOL simplification mechanism can handle this situation. However, in the case of complicated predicates such as those used in steps 101-143, the preparation required to satisfy the proof checker about the appropriate simplifications was much greater than even the forty steps expended. More concisely, FOL is not as capable of checking predicates true of several objects on the board (for different reasons) as is a human.

Nor have we approached the human capacity for set manipulation. For example, in observing pawn captures, such as step 11, the human quickly and naturally perceives the mutual exclusion (inequality) of the members of the capture set. That is, the human can say, "Black captured four (or five, or six) white pieces on white squares." He understands quickly and easily the essential inequality of these captured pieces, and the various restrictions on their values (for example, none of the pieces currently on the board was captured). Our axiomatization, reluctant to do either arithmetic or set theory, and bound, as it is, to the heavy quantifier manipulations of natural deduction, cannot express this notion as easily. Rather, we must, for each capture, hypothesize the move that the capture was made on, and the captured piece, and prove the pairwise inequality of the various captured chesspieces. Thus, for example, the information quickly apparent to the human puzzle solver, after he notices the four piece captures, requires steps 301-330 of the main proof.

This problem is not, we feel, due to the clumsiness of the position (state vector) approach. Rather, our restriction to first order formalism, and our refusal to enmesh ourselves in a generalized set theory, has created a situation which requires dealing with each individual, individually. Our problem is still small enough that this is a reasonable activity; however, a system that would need to deduce truths about many objects would certainly need a more universal mechanism (set operators, for example) for manipulating sets of objects.

Section 5.5.2 Epistemological Axiomatic Limitations

There are more things in heaven and earth, Horatio,
than are dreamt of in your philosophy.

Hamlet, Act I, Scene 5

One of the nice things about a formal logic systems is the ability to easily extend the formalism, by the addition of new constants, axioms and attachments, to handle unforeseen or incompletely covered situations. Thus, while we have interpretted our task to be axiomatization of retrograde chess, it is a simple extension to include a definition of the SUCCESSOR relation, appropriate and useful Makemove functions (with attachments) and thereon to do *forward analysis* for chess. We have briefly touched upon these notions in considering Berliner's problem, section 5.4.2. However, as currently constituted, our axioms of chapter 2 are not capable of handling problems requiring this kind of forward analysis.

In any case, the purpose of this section is to detail which kinds of chess puzzles this axiomatization, in its current form, has trouble expressing.

Certainly the most common of all chess puzzles are the *white to play and mate in* n *moves* variety. For n sufficiently small, we really must confess lack of interest in most examples of this type of puzzle. Given the definitions of forward movement, and appropriate attachments, such puzzles are easy single step simplifications in FOL. A few of these puzzles rely on the ability to castle or capture en passant, and the justifications for en passant capture are occasionally quite complex, involving the sort of retrograde analysis we have been doing in this paper. These axioms are, of course, quite suited for that kind of analysis. *Castling* in *mate in n* puzzles has a more complex position; one can almost never prove that castling is legal, though often there is no reason to presume it illegal. These axioms can be used to prove, in the usual retrograde way, castling illegal, or the problem statement appended to include the appropriate restrictions on the position to imply its legality.[74] A minor fillip can be provided to these mate in *n* problems by the addition of *fairy chess* pieces **[Dawson73]**.[75] Of course, our axioms would need the natural extensions to handle fairy chess pieces.

A more complex situation is presented by the problems of the form *white to play and win (draw)*. What we have here is an extension described by the WHITE_CAN_ACCOMPLISH predicate of section 5.4.2. Additionally, there is the necessity of defining the predicates WHITE_HAS_WON and THIS_IS_A_DRAW. Clearly, they are non-trivial predicates, though they can be well defined in certain circumstances (particularly if white has a forced mate in n, an overwhelming material advantage of certain kinds (king and queen against king, for example) or insufficient material exists to force a win (king and bishop against king and knight). What might be a trivial win for a chess master can be completely opaque to average player. We imagine the attachments to such predicates would rely heavily on the I_DONT_KNOW response available in the attachment mechanism (section 2.1.7.2). Similarly, *self mates* and *help mates* require different definitions of the CAN_ACCOMPISH predicates.

In some sense, these are examples of *construction* problems: the problem solver is to present a sequence satisfying some property. Another type of construction problem, for which these axioms are very ill-equipped, and which lies on the periphery of chess problems, are problems of the form, *construct the board with the most (fewest) legal moves (captures, promotions, ...)*. Solutions to these sorts of problems are usually presented as "this is the best known solution", rather than "here is the solution, and this is why one can't do any better." As our system is directed towards proof and confirmation, it is naturally incapable of commenting on such results.

But, needless to say, these are not the tasks this axiomatization has been directed towards. Rather, we were considering retrograde analysis in our definitions, and it is more reasonable to inquire where our retrograde failures would lie.

It should be clear by now that the mathematical knowledge represented by these axioms is very minuscule. All mathematical manipulations have been accomplished by considering each case on our finite board separately, or by actually performing the implicit calculations in the simplification

---

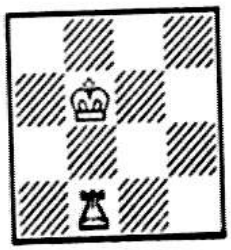
74 One reader of a draft of this paper inquired how the question "Assume castling is legal unless you can prove otherwise" might be handled. In general, this is an undecidable question; any axiom system as powerful as ours is incapable to proving whether or not certain statements are theorems. This follows from the $G_{\ddot{o}}$del undecidability result.

75 Fairy chess pieces are fanciful chess pieces with usual moves. Consider, for example, Dawson's Grasshopper and Nightrider. A Nightrider may make consecutive knight's moves, in a straight line; the Grasshopper moves along the orthogonals and diagonals, but only by hopping over one man of either color to the next square beyond. The reader interested in this mythology is invited to consult Dawson's book.

mechanism. This is clearly impractical for problems that rely on more complicated mathematical deductions. Similarly, those inferences promoted by set theoretic and counting arguments are painful tautology decisions in the current system; it is easy to construct examples of sets too large to be handled this way.

The current axiomatization is oriented towards unknowns centered around particular squares. Unknowns centered around unknown squares would cause greater difficulty for the simplification oriented system, though ought not to be impossible (section 5.4.1).

Another difficulty with this axiomatization is its insistence upon centering the problem around a specific squares and boards. For example, the question *Is white in check on the piece of a board in figure 48*:



Is white in check on this fragment?

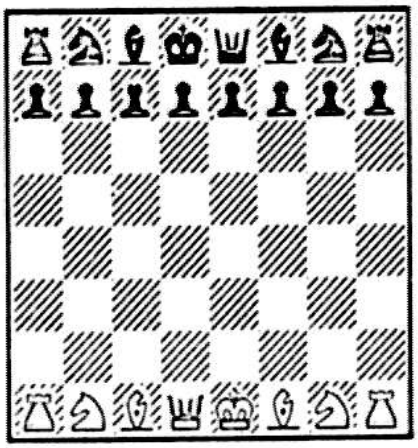
figure 48

is obviously observationally true, but its phrasing in this axiomatization would appear as:

```
(Valueon(b,Makesquare(drx,dcx))=KW∧Valueon(b,Makesquare(Bsucf drx,
    dcx))=MT ∧ Valueon(b, Makesquare(Bsucf Bsucf drx, dcx))=RB) ⊃
    WHITEINCHECK b
```

Hardly the natural interpretation. It is perhaps true that a notion of board fragment should have been included in the axiomatization. This points to a greater difficulty in this axiomatization; that the functions and objects of the chess model are not robust enough to handle perversions of their original sense. These attachments were the *obvious* simple direct functions to compute the obvious values; they were adequate (with some pushing and pulling) to function as the chess eye for this problem. However, it is now clear that a more flexible eye would be appropriate to handling a larger variety of problems. This more flexible eye would probably involve much more complicated functions.

A more germane example is provided by the following problem [Gardner59]:



White to play and mate in four.
As usual, this is a legal position.

figure 49

The puzzle here is to recognize that (as the black king and queen are on the wrong color squares) that black and white have switched sides, with the black pawns advancing to the seventh rank. While the current axiomatization could be used to prove that the given board, supposing the black pawns on the second rank, is not the board of any legal position, the "trick" of the problem cannot be expressed in this fixed board form.

There are also some chess puzzle concepts, such as "blocking structure" and "path" which lack the necessary counterparts in our axiomatization. These have their fumbling expression in our system (see, for example, much of the last seventy steps of the main proof), but this expression is not entirely satisfactory.

Section 5.6 General Representation Issues

Most of this paper has been dealing with representation issues of the chess world. If we did not think that these examinations were relevant to epistemological issues in general, we could not justify the attention we have given them. It is worthwhile, therefore, to turn our consideration to general representations issues, considering the light shed upon them by our example.

Section 5.6.1 Multiple Representations

> *A description must be able to represent partial knowledge about an entity and accommodate multiple descriptors which can describe the associated entity from different viewpoints.*
>
> *Bobrow and Winograd*[76]

One of the more complicated problems any generally intelligent computer will have to face is the difficulty of manipulating the various aspects and forms of particular objects. Any real world object (or class of objects) has a set of properties. For example, the book in front of me is red, weighs

---

76 [Bobrow77]

about four pounds, is made of paper, occupies a certain position (particularly, its near the phone and my drinking glass), not to mention the diversity of the information recorded within it.

Let us consider merely the problem of manipulating and examining the book in space. In general, the color, content, and composition of the object are not relevant to this task, and can be ignored.[77] They serve merely to confuse the heuristic portion of the program.

Even within the narrow domain concerned with the locus of the book, there exist many formats for storing locational information. The bulk and location of the book might be represented by marking the occupied squares on a visual grid [1]. We might encode much of the same information at a higher level, as a series of coordinates for the vertices of that rectangular parallelepiped [2]. If we've analyzed the scene, a linguistic description, such as *the book is to the right of the glass, and in front of the telephone* [3], or as a WFF or network, with explicit links or predicates, such as *RIGHT_OF(Book, Glass), and IN_FRONT_OF(Book, Telephone)* [4] might be the appropriate structure. Notice that we have here four different ways of representing what is essentially similar information.

It is not the case that one of these forms is the *correct* one. Rather, each is heuristically appropriate to uses at some time. The grid is both a typical input expression, and a possibly useful form for an algorithm seeking to quickly comparing scenes. The coordinate structure could be used to easily locate the desired object. A program whose primary task was human interaction might find storing sentences such as [3] a useful facility, while inference might require [4]. A program that needed to do all of these might very well keep several or even all four representations. Nor is it the case that these are equivalent representations. They represent different combinations of inference and deletion, and are not mutually rederivable.

We feel the pain of this problem very acutely, even within our limited set of chess problems. Particularly the dichotomies of board/position and value/piece reveal an aspect of this problem. As we have already devoted a section to their comparison (section 5.2), we shall restrict ourselves to a few brief conclusions here.

It is clear from this experience that representing information in canonical form (every fact has a particular, highly structured format to which it must fit) is a losing proposition. Such structuring must, of course, be to the most general form; however, most frequently, it is the particular form, with its implicit information, that is the most pliable for heuristic manipulation. Thus, while the positional notation is the most general form of representing a chess situation, actual computation is easiest when dealing with concrete boards. The same constraint applies, of course, to dealing with pieces and values.

Multiple representations require the ability to translate between forms. In the case of the board/position dichotomy, this translation is explicitly related in the ***TRANSPOS*** axiom and theorems (sections 2.1.9.2 and A.6). It is considerable trouble to interchange representational forms in mid-proof; unfortunately, unless great care is taken in matching proof segments and lemmas, it is a frequent occurrence.

An alternate facet of the multiple representation problem is the difficulty of transferring properties between different, but similar states of the world. The book is on the table. If I walk out of the

---

77. However, imagine the helium balloon: the manipulator ignores its compositional properties at its own peril.

room, around the building, and back in, will the book still be on the table in this new state? The entire issue transferring properties between similar states is a very complex "can of worms." This problem is visible in our chess system; while any move changes only a few of the pieces on the board, it creates an entirely new state, with unexplained properties. We have, however, one prominent success to report on this matter, that our the *chess induction* schema. We have found this schema to be very useful in tying together the properties of not only "close" states, but also relating states separated by many moves.[78]

In many respects, this requirement of transferring between different representations can be viewed as a metaphor for the heuristic portion of the A.I. problem itself. In that view, intelligent action consists of transversing some search space; multiple representations merely pervert that space (like other operators), adding short cuts and cul-de-sacs.

Section 5.6.2 Abstract and Concrete Representations

The previous section discussed the varieties of representations. Our experience with this chess proof leads us to an important conclusion about these formats: a system requiring complex and detailed deductions must frequently retain both *abstract* and *concrete* representations of its input.

Let us consider this vision example. The program knows that it is viewing some "scene". In some general sense, this is the abstract form for this hypothetical vision understanding system. As it manipulates the raster input, abstracting and specifying features, the abstract form becomes qualified, just as the abstract form of the position px is qualified in the course of our proof. Practically, the vision system might extract features, manipulate the resulting data structures, and return to the concrete input format only for clarification. Rarely, an inconsistency might force another analysis of the input.

The chess example has a parallel structure. Any problem is explicitly an element of the set of positions. Various features, such as the values on a given square, are concrete facets of the input analysis. In forward analysis (as, for example, outlined in section 5.4.2) we would use less of the *abstract* form, preferring to live in the secure computation of specific boards. Retrograde analysis, on the other hand, deals with a more "unsure" situation, and demands a more flexible representation. Hence, the predominance of the more abstract form (the position) in our proof.

Section 5.6.3 Heuristics and Representation

As we mentioned in section 1.2, the general A.I. problem naturally divides into epistemological and heuristic parts. This paper has been concerned with the minimal requirements for an epistemologically effective representation. However, a few words on the heuristic devices employed in generating this proof might prove interesting. It is to be remembered that these comments are of an introspective nature; that is, we describe what we found difficult and easy, and how a heuristic system might eventually be organized to acheive such a long deduction.

Both the generation of this proof, and its surface structure, show a clear division into three types of activities. First, a specification of the general proof outline must be obtained. In this proof, that corresponds to the "human" proof described in section 1.6.2. Then, into this outline, the appropriate

---

78 There is a third "multiple representation" issue that FOL handles for us automatically, that of keeping the context of any deduction. The dependency mechanism performs this task fairly well, though naturally, in a very conservative fashion.

lemmas and theorems must be formed. Finally, individual proofs must be constructed for each of the particular theorems.

This outline misstates slightly, in that the second and third activities, lemma selection and proof, occur concurrently, difficulties in proof often prompting new subgoals (lemmas) for the selection process.

It is clear that both for the human proof solver, and any future program, that these steps are listed in order of increasing difficulty. It is not clear how to generate an outline of the correct solution, given the problem. It is also clear that this more efficiently done the more the solver knows about the tricks and short cuts of solving chess puzzles. Thus, a human unacquainted with chess, and presented only the rules, would finding solving this problem a very difficult task, while it is trivially easy for the chess master. Proving actual lemmas, once one had the "hang" of it, was relatively easy.

Working within the context of a proof outline, the main difficulty in generating this proof arises from specification of the lemmas and sub-theorems to be used along the way. In a strong sense, the proof of almost all of the lemmas is relatively trivial, given the existence of all of the axioms they employ. In practice, if the proof of some theorem became too difficult, a useful lemma was assumed, the lemma's proof becoming another subgoal. Within the proof of any lemma, almost all of the steps are either axiom instantiations or simplifications.

Effectively, we are offering a personal confirmation of a judgment of Sacerdoti [Sacerdoti73], that it is more important (and more difficult) to determine the plan for a course of action, than to worry about filing in the detailed descriptions of that plan. Of course, one's proof can flounder on either set of hard places.

There has recently been some work on incorporating goal direction into the FOL system. The reader is referred to [Bulnes 79] for a description of that work.

Section 5.6.4 Functions and Predicates

In section 5.2.5 we mentioned the use of *special functions*, essentially, the use of an algorithm to compute a value (when the algorithm is known). This principle can be expanded into a general "moral" for axiomatizations are (usually) more tractable objects than predicates. In this section, we contrast the *functional* and *predicate* styles of axiomatizations.

A functional axiomatization is one where (relatively) unique relationships are expressed as the values of particular functions; a predicate system denotes these relationships as predicated relationships. That is, to find the instance satisfying some predicate, one manipulates the axioms of that relationship, and proves a unique correspondent.

What we are contrasting here is an intra-representational choice. That is, in generating an axiom system to represent some domain, one often has the choice of expressing some notion as either the value of a function, or the set of things true of some predicate.

An example of a "bad" axiom from our axiomatization may clarify this issue.

Consider the axiom ***MCONSEQL*** which defines the pawn promotion.

AXIOM MCONSEQL:

```
∀p . (PAWNPROM Move p=(LASTRANKER(To Move p,Color Prevpos p)∧
        SIMPLELEGALMOVE(Prevpos p p)∧
        PAWNS Mover Move p∧
        VALUEP Valueon(Tboard Prevpos p,From Move p)∧
       ((BVALUES Promoted Move p=BVALUES Val(Prevpos p Mover Move p))∧
        (WVALUES Promoted Move p=WVALUES Val(Prevpos p Mover Move p)))∧
        Val(p Mover Move p)=Promoted Move p));;
```

It states that a pawn promotion takes a pawn to the last rank, by an ordinary move, that the piece must have had pawn value at the start of the move, that black pawns promote to black pieces, that white pawns promote to white pieces, and the promotion value is from the set of possible promotion values (as defined by the definition of Promoted).

Now, this axiom is not *incorrect*. It is merely clumsy, and we regret having written it this way. We leave it in, however, to be the object lesson of this section. The axiom would have been more easily used if it had stated: [79]

```
AXIOM MCONSEQL:
    ∀p . (PAWNPROM Move p=(SIMPLELEGALMOVE(Prevpos p, p)∧(
            (WHITETURN p ∧
             Row To Move p = 8 ∧
             Val (Prevpos p, Mover Move p) = PB ∧
             (Val (p, Mover Move p)=QB ∨ Val (p, Mover Move p)=RB ∨
              Val (p, Mover Move p)=BB ∨ Val (p, Mover Move p)=NB))
        ∨   (¬WHITETURN p ∧
             Row To Move p = 1 ∧
             Val (Prevpos p, Mover Move p) = PW ∧
             (Val (p, Mover Move p)=QW ∨ Val (p, Mover Move p)=RW ∨
              Val (p, Mover Move p)=BW ∨ Val (p, Mover Move p)=NW)))));;
```

that is, if it had explicitly stated, referring to individuals and equality, what was intended, rather than referring, through the indirection of predicates, to sets of objects.

If this lesson seems too obvious, perhaps it is important to mention there are reasons for a predicate approach, to wit, that the various tautology deciders currently in FOL (TAUT, TAUTEQ) are much happier with predicates than with either equality (which TAUT cannot handle) or functions (with which TAUTEQ has trouble).

Section 5.6.5 Whorf's Law

The last section illustrates an important moral of representation theory, a Whorf's law of artificial intelligence [Whorf56].

Whorf's hypothesis was a linguistic one; that a person's language shapes the way he thinks. Our experience with fitting a chess proof into formal logic gives strong evidence that this notion extends to include formal representational systems, and is a useful notion to remember in generating them.

Obviously, a limited representation can only express limited notions. More significantly, the structure of the inference system, and the axioms, will subtly mold the resulting deduction. For example, FOL tautology decider TAUTEQ is capable of substituting equals for equals in predicates,

---

79. Recall that "WHITETURN p" implies BLACK just moved.

but not in functions.[80] This promotes an axiom structure incorporating more predicates and fewer functionals (an unhappy situation).

The FOL system has grown as this proof has progressed. When the proof was begun, the only supplemental inference commands (beyond the Prawitzian natural deduction rules) were TAUT and TAUTEQ. Semantic simplification, so essential to our chess eye, was developed concurrently with our experimental axiomatization. Copious use is made of these commands. More recently, after the first few iterations of proof had been completed, *syntactic simplification* (a massive substitution command), and a decider for monadic predicate calculus were introduced. It is noteworthy, however, that even if we wrote another iteration of this proof, these commands would probably not be important. This is because the axioms are not structured to take advantage of their presence. Such structuring would imply more equivalences and monadic predicates, while our current structure tends toward implicatives and dyadic (and greater) predicates.

Similarly, it is difficult in FOL (largely because it is a formal mathematical proof system, partially because a lot of effort would be required to check any change) to correct the course of a proof, to delete an offending command, to make a slight adjustment to an axiom, to change an incorrect declaration. This arrangement promotes a *stiffness* of expression; once incorporated, change is difficult. Because change is so difficult, it is easier to become set in ones ways, and harder to experiment. Again, we see an example of language influencing representation.

Section 5.6.6 States and Representations

Perhaps a section on the value of state vector representations in general epistemological situations is warranted. After all, the position, our state vector, has proven very successful at capturing some of the important aspects of our domain. For example, the notions of *must have happened in this game, though I don't know when and it was true then, and can't have changed, hence is true now* are very well specified by having the position as a history vector, and through the use of chess induction.

We contend that a similar ideas can for the basis for powerful representational mechanisms for A.I.. For example, retaining a notion of the *present state of the world, including the history of reaching it* resolves some of the confusions inherent in the naming of objects. A person who has lived but is now dead does not, in some sense, exist in this world. However, by retaining the history of the world in reaching this state, we are able to speak of him in the appropriate context. Similarly, a God that was able to know the rules involved in all state transitions would be a good resolution of the issue of omniscience.

But, of course, reasoning within this state of the world, and not knowing all of its rules, we cannot predict the future. We can, however, if we know the current state, reference that state as the expected descendant of some past state, and reason about the future (up to the current) in the past. We can not reasonably reason about the future, even in the past, beyond the current state, other than to say, "if the future has the following properties, then ...". Note that of all possible states and histories, we can distinguish one and call it, *reality*. This is typically the state we are *in*, just as the game that reached GIVEN, the problem board, was reality for that situation. We can name individuals in chess, such as the grasshoppers and nightriders, that have properties, just as we can name mythical flying horses, and state that this flying horse was named Pegasus, who sprang from the body of Medusa at her death, and so forth. We can speak of Pegasus if we are careful of

---

80. From a=b, TAUTEQ can deduce p(a)=p(b), but not F(a)=F(b).

mentioning the context within which we are speaking, just as we can speak of nightriders, if we move beyond the context of "real" chess, or "real" reality.[81] A full notion of such a state/reality duality might require additional predicates and individuals into that state.

Reasoning within specific states can have other benefits. For example, the reasoning about knowledge can sometimes be resolved by retaining the context within which each wise man is reasoning. Note that as an omniscient observer, we retain the right to reason about all contexts.

Transitions between states can be seen as the flow of reality. This chess problem has been very over simplified, for things happen in a discrete, regular fashion. In a more general, *real* world, processes will not behave as nicely. We will be able to find some regular laws (*if* x *is dead in state* s, *then* x *will be dead in all successor states to* s.), and able to use our induction schema to manipulate such rules (*if* x *is dead now*, then, by induction, x *will be dead in all future worlds*).

Reasoning in this state transition formalism has a distinct disadvantage, however; we can rarely be sure that what was true in state *s* will be true in the *successor of x*. We have heuristics for processing such situations (*if x leaves the room, all of x's clothes go with him*) but even such ordinary rules as *when I awake, things will basically be the same* can be violated, as Rip van Winkle discovered, much to his discomfort.

Section 5.7 Historical Context

It is perhaps useful to place this work within the historical context of representation systems in Artificial Intelligence.

The inference mechanisms employed by A.I. systems can profitably be divided into two varieties: syntactic and semantic. Syntactic inference is performed by considering the *form* of a particular goal and set of rules. If that form matches the standard required by the set of rules, one can conclude a result whose form is determined by both the result form of the rules, and the binding of entities in the match. Thus, for example, in a system structured as ours is, one employed the axioms and already proven WFF's, through the natural deduction rules of inference, to obtain new WFF's.

Semantic inference mechanisms are magic. The particular goal WFF is offered to some set of functions and data structures, and that oracle decides if the particular conclusion is correct. While it is (theoretically) possible to describe these data structures and functions in a mathematical form, raising them to the syntactic level of the first kind of structure, such an attitude is both unlikely to succeed, and, in some strong sense, *wrongthink*. Typically, the "black box of semantic routines" embodies some *model* of the world viewpoint of the system programmer.

This is not to imply that by calling such mechanisms "magic" we want to denigrate them. Rather, they will be the fundamental mechanisms of any successful A.I. system. The interactions allowed in purely syntactic constructs are too broad to be able to avoid exponential search.

However, it is also important to point out that model based reasoning is invariably too limited in its expressive power to perform complicated and varied inference. We need not only to compute in our models, but also to talk about them. It is the assertion and demonstration of this paper that deep inference is possible through a combination of both forms of representation.

---

81. Prior's modal logic approach to time was discussed in section 5.3.2.

Neither representational system is new to artificial intelligence. Purely syntactic approaches, such as resolution based theorem provers were once in vogue. But even in the ultimate example of such a system, QA3 [Green69] one sees the stirrings of the use of models. But Green employed model based computation only out of necessity and last resort.

At that time, Winograd's SHRDLU [Winograd72] was somewhat a competitor of QA3. While the Planner antecedent and consequent theorems employed in SHRDLU have a surface resemblance to formal logic, their employment in a simulation system made them essentially semantic conventions. While perceived as a great success at the time, the limitations of such a purely semantic approach have now become apparent (section 1.3.1.2; see also [Moore75]. Essentially, a purely model based system can efficiently manipulate the objects in that model, but has no mechanism for talking about those objects or the manipulations in a non-manipulative sense. Hence, Winograd's program could fail to place a block on a pyramid, but could not talk about the possibility of placing a block on a pyramid.

There have, however, been several successful combinations of syntactic and semantic representations. Perhaps the earliest and most impressive was the combination of diagrammatic based computation and syntactic deduction presented in the geometry machine [Gelernter63A][Gelernter63B]. Gelernter and his associates employed computation on a geometric model to aid in the discovery of syntactic proofs of elementary plane geometry theorems. The mathematics of the model system employed by the Geometry machine is explored in [Reiter]. Other, more recent programs have employed simple, explicit models to perform some of their necessary inferences. Examples of these programs are the electrical circuit systems at BBN [Brown73][Brown74] and MIT [Sussman75], Funt's system for predicting the paths of falling blocks [Funt77], and Rieger's [Rieger76] [Rieger77] program for approximating the workings of devices.

We can, perhaps, attempt a minor taxonomy of such model based systems. Besides the above distinction about the use and availability of both syntactic and semantic forms, we note two other distinctions. First certain of the above systems employ their models not only as inference mechanisms, but also as heuristic aids. The Geometry machine is a prime example of this use. The deduction presented in this paper has, of course, ignored heuristic issues, considering only epistemological questions.

Secondly, these programs can be divided by the *kind* of model they employ. Our model for the chess proofs has been an *exact* one. We are as sure of its correctness as we are of our axioms; we are sure that its functions completely and accurately model our knowledge of chess. To the limit of their electrical consideration (races, hazards, etc.) the electrical programs were also accurate models. Funt's block's program, however, applied an *approximate* model of the situation, performing a simulation of the falling blocks, under the watchful gaze of a simulated eye. Rieger's system is a similar simulation. Gelernter introduced unnecessary approximations into his system to keep it from being too accurate and helpful a model.[82]

Thirdly, all of the systems considered so far have employed a *single* model in their inference mechanism. Whether only single models are appropriate (to reflect the natural "human" single-minded view of the world) or multiple models are merely a further step is an open research question.

One important, complex and necessary step has been avoided by all these systems: a reification of

---

82. Gelernter wanted to study using the model as a heuristic, rather than inferential aid.

models, treating the models themselves as objects of the system. One can see an important suggestion in this direction in [Weyhrauch78], though it remains a research opportunity.

Section 5.8 FOL

On many occasions in this volume we have complained about the various limitations and privations imposed by our proof checker, FOL. This is not to imply that things are all that bad; FOL does what it does fairly well. However, while familiarity may not breed contempt, it at least breeds an awareness of deficiencies. We are obliged to attempt a rudimentry catalogue of our perceptions of where FOL could be improved.

The most elementary changes (at least from a structural point of view) involve the inclusion of additional inference rules. For example, a tautology resolver that could do substitutions inside of functions as well as on predicates would be a "relatively" simple fix that would have a large beneficial effect on total proof size. One can imagine, for example, that most of the uses of SUBST and SUBSTR, and many of the applications of ASSUME could be dispensed with were it not for the necessity of convincing TAUTEQ (again) that $a=b \supset F(a)=F(b)$. We have partially circumvented this constraint by the uses of the β functional parameter and the ***Substitution*** axiom; these temporary solutions are not, however, completely satisfactory.

Similarly, the FOL user should be allowed to define his own inference rules, providing the code to decide them. This proposal works in parallel to the more powerful semantic simplifier discussed below. Merely being able to substitute for parameter predicates and functions is not enough.

There are, of course, several more radical changes that the earnest FOL user would desire. The primary emotional complaint about writing FOL proofs is the necessity for *stiffly* expressing each (almost incorrectable) step. This problem is exhibited in several ways. In its simplest form, it can be perceived in FOL's refusal to forget any (unreferenced) declarations, or remove any (unused) inferences from the proof. The relatively inflexible syntax is also a source of annoyance. Similarly, the necessity of generating a permanent, particular proof step, particularly one that is only a propositional derivative of some other inference line (what FOL calls a VL) for use in only one instantiation, is a corresponding clumsiness.

In a larger sense, this stiffness is seen in the necessity of repetition of identical (or nearly identical) arguments (on different objects) to produce similar results. One frequently wishes to say, *this case is just like the last one, but use the axioms for white rather than black in proving it*. In its simplest form, this proposal might be incorporated as a proof schema; that is, follow arguments of the following form, and reach a similar result. A more grandiose schema might include a provision for reasoning by analogy, that is, taking a proof and finding the parallels to generate a similar proof.[83] The moral here, perhaps, is that communication is facilitated by informality, the ability to omit or abbreviate objects. FOL (like most programming languages, particularly the "lower level" programming languages) requires a fairly formal statement of action. This is uncomfortable. Writing a FOL proof of this size leaves one with the same feeling as writing a large assembly language program.

FOL can be a very uncooperative proof checker; while it is quit willing to deny the legality of some step, it is usually unable to explain why. It would be easier to write interactive proofs if the proof

---

83. We see some of the elementry steps for this work in the thesis of Kling [Kling71] on reasoning by analogy. Kling, however, only used the previous proof for selecting the axioms to be given to a resolution theorem prover; we suggest that this inference scheme try to follow the form of the given proof.

process included some guidance. We are all familiar with how debugging facilities easy the programming process. However, the inclusion of such facilities is not a trivial request; while a quantification check in semantic simplification could easily trace a failed step, such a facility within the present decision procedures would certainly be a difficult to implement.

We have here a giant example of a FOL proof; many different lemmas and theorems have been pieced together to accomplish the end result. However, we have received virtually no help from FOL in producing this structuring. While FOL permits one to declare any arbitrary WFF an axiom, it has no other mechanisms for structuring a proof. Two possible improvements might be suggested. One would like a *theorem* command, which would take and save a given result. Additionally, either a block structuring method or an analogical inference command would aid the engineering aspect of proof generation. The recent meta level facilities [Weyhrauch79] and goal structure commands [Bulnes79] could be used to alleviate these difficulties.

Of course, our major impressions and recommendations are reserved for the semantic simplification mechanism.

Our primary complaints concerning simplification center on the inability to apply all of the observational knowledge available to a given simplification. This Hydra rears its heads in many ways. In its simplest form, it is seen in the demand of simplification that all arguments to functions be "well defined" before they can pass through the FOL - model barrier. For example, if A is a constant, and y, a variable, simplification (and the common *call by value* implementations of LISP) are unable to compute:

(CAR (CONS A y)) = A

Having such a ...bolic evaluation would have been quite helpful in generating this proof. For example, on many occasions we would have a parameter board, a parameter value, and two constant squares, and wish to show that MOVETO was not true on these arguments (the squares resting at an angle beyond the movement of any piece, say, BKR1 and WQR2). A parameterized semantic simplification would accomplish this; as it is, we need to resort to the hack of instantiating the axiom ***MayMove***, and simplifying the result.[84]

There are times when even this dodge will not work. It these axioms we employed to do forward analysis, then the following object would prove useful: a position built up from a parameterized position (presumably, the given position of a problem) with successive generations of moves and boards appended. Now, there are many simplifications that are, by nature, observations, and should possible on such an object. However, as it is not an INDCONST, it would not even be passed to the simplification mechanism.

A partial solution of this problem would include tagging those objects that were "variables" to simplification, and allowing the user program to distinguish those tags.

Permitting the user program to see the variables of the FOL model could have other beneficial results. Consider the case of quantification checking. In the current simplification mechanism, the only quantification checking permitted is the check of a finite sort whose elements have been listed in

---

84. MayMove is defined as: ∀b ∀ sq1 sq2. (MOVETO (b,v, sq1, sq2) ⊃ (Column sq1 = Column sq2 ∨ KNIGHTMOVE (sq1, sq2) ∨ Row sq1 = Row sq2 ∨ SAMEDIAG (sq1, sq2) ∨ KINGMOVE (sq1, sq2) ∨ (TWOTOUCHING (Column sq1, Column sq2) ∧ (WSUC (Row sq1, Row sq2) ∨BSUC (Row sq1, Row sq2))))).

an EXTENSION command. This usually works well enough in the chess world for the simple problems we have considered (there are, however, some lemmas we have not proven because the desired simplification were too complex). But quantification checks can also be accomplished by other divisions of the variable set than into individuals. For example, it is true that all integers are either positive integers, negative integers or zero. A division into these sorts might permit the simplification and verification of some sentence of number theory or arithmetic. However, we would certainly not want to check every integer in establishing the validity of that WFF. Rather, the user should be allowed to define his own quantification checking mechanisms, either in addition to, or instead of the current extension checks.

In the current organization of FOL one must invariably know what one wants (that is, what WFF one wants in one's FOL proof) when commanding any inference step.[85] A more powerful scheme would be to permit the prompting of the simplification mechanism, which would then complete the assertion in the appropriate manner. In a simple form, as prompt of ∀ x. WFF might, instead of returning ¬∀ x. WFF instead counter with the more useful ∀x. (WFF ∨ x=y ∨ x=z). If ¬∀ x.WFF were what was desired, it would have been asked for.

On a grander scale, this mechanism could begin a approximation of the inference schemes of humans. Our proof is a very good case in point. The competent human puzzle solver can observe our problem chessboard and state: *The only legal move for white was to have just promoted a pawn to a rook, moving from* BQB2 *to* BQ1, and capturing a black piece. This (for the good puzzle solver) is an *observation*, not the deduction. It is the only possible legal (last) move for white. To generally mimic this ability in FOL, however, would require a simplification mechanism that could take an input board, and return a set of WFFs, one of which would be true of that board. This might be accomplished with an appropriate set of meta level reasoning commands.

A final point on the relationship between the FOL level WFFs and the semantic simplification's attachments. These attachments, you may recall, were presented in the axiom section (chapter 2) of the proof. This is because they share many attributes with the axioms, both conceptually and functionally. Conceptually, they are among the building blocks from which the rest of the proof was created. Functionally, these attachments can serve the same crucial role as axioms. That is, defects in these functions can permit the horror of a contradictory deduction (and hence the deduction of any WFF.) By the semantics of the LISP programming language in which they are imbeded, these axioms acquire meaning. It is an unhappy circumstance that the meaning incorporated by these semantics is not somehow transferable to the FOL axiom level. Similarly, it is unfortunate that the FOL level axioms cannot be *compiled*, somewhat automatically, into simplification level attachments. While it is true that this inability places the burden of a redundant inefficiency on the FOL user, this is not our major point. Rather, our intention points towards the time when some similar scheme might be incorporated into an intelligent program. Optimally, such a program would build up frequent action patterns, effectively *learning* new processes. With our representation formalism, this could well be modeled by the compilation of FOL WFFs into the corresponding model attachments.

---

85. There is a small, but important exception to this rule. One can semantically simplify an expression of the form $F(x_1,x_2,\ldots x_n)$, and obtain as an inferred step: $F(x_1,x_2,\ldots x_n)=y$, for function F and terms x_i and y.

Section 5.9 Evaluation and Summary

The reader is probably by now feeling somewhat overwhelmed by the mass of argument and detail, proof and text that we have presented. Let us conclude, therefore, by summarizing and evaluating our important points.

Many doctoral dissertation seems to be of the form: "I've solved the A.I. problem, except for a few implementation details I've not bothered to work out." It is a primary premise of this work that "solving the A.I. problem" is a very difficult task, and much fundamental work on both representations (epistemology) and search (heuristics) remains to be done before its solution.

This particular paper has been centered upon consideration of *epistemologically effective* representations. There is a common fallacy in most A.I. work, that because some particular representation, invariably employed in some particular task, was sufficient to solve some of the problems of that domain, that that representation can be extended to the rest of that domain, and onwards to the rest of whatever we want our computers to do. Our perspective has been from the opposite direction. We have started with a very general representation (formal logic),[86] discovered it inadequate for modeling even a limited problem domain (chess puzzles), and extended it (by the use of the *Chess Eye*) to where it can more easily and naturally represent certain interesting problems. However, there are many questions about chess puzzles that are inconvenient or impossible to ask within our representation as presently formulated.

It is important to remind the reader that the fact that our representational formalism is based on first order logic does not imply that we are suggesting the use of general purpose theorem provers. Rather, our comments in this work have been reserved primarily for the epistemological part of the A.I. problem; we have spoken very little, if at all, about appropriate heuristic mechanisms. Our approach has been, in some sense, bottom up (consideration of the nature of reasoning sequences) rather than top down (discovery of the appropriate state search methods).

It should be remembered in evaluating all of the particulars of the individual representational choices that we have made, that first order logic is a family of representations. The failure of one particular example of a formal logic system is a failure of that selection of individuals, predicates and functions, not the failure of the entire notion of formal systems.

There are several general representational issues illuminated by this work. In earlier sections, we have pointed out the distinction between concrete (board) and abstract (positional) entities, and the importance and value of state vectors (positions) in durational deductions. We have seen many facets of the multiple representation issue, such as differing representations for the same object (board and position, piece and value) and preserving properties between similar objects (chess induction) illustrated by our axiomatization. We have noted the necessity for both syntactic and semantic representations; there are many important syntactic type deductions which are not expressible within a model.

Our major emphasis, however, has been on the interrelation between observation (the *Chess Eye*) and deduction (the overlying FOL language). We have closely examined a particular observational framework and found it adequate for computing certain properties (functions on closed objects).

---

86. It is true that formal logic is a general representation schema. It is also true that certain Turing machines are universal computing devices. However, this latter knowledge is of little use in actually programming computers.

However, we have been left with the sense of the impotence of this scheme: not all of the computation and observation we wish to make is on fully defined, well structured objects. The issue of adapting and generalizing the simplification mechanism to include these observations remains a major place for future exploration.

Appendix A Chess Lemmas

Due to space limitations, Appendix A, the proofs of the general chess lemmas and theorems, has not been included in this volume. Those proofs may be found in the author's Ph.D. dissertation, which is available from *University Microfilms, 300 North Zeeb Road, Ann Arbor, Michigan 48106.*

Appendix B Proof Lemmas

This appendix presents the various lemmas and theorems relevant only to our given puzzle, yet too detailed to belong in the main exposition, chapter 4. As such, its form is essentially similar to appendix A. However, unlike that appendix, these lemmas are listed chronologically; that is, in the order of their use in the main proof.

Section B.1 Undefined Squares on the Given Chessboard

Our first problem lemmas are, trivially, a single simplifications. Just as there are many useful facts about chess derivable in a single simplification, and useable in many contexts (as presented in section A.1), similarly, there are observations about the interesting boards of this problem. (Observations we prefer to have to compute only once.)

We observe that the only undefined square on the problem board is WKR4

```
******label GivenUD;
******simplify Vsq.(Valueon(GIVEN sq)=UDssq=WKR4);
1 Vsq.(Valueon(GIVEN,sq)=UDssq=WKR4)
```

We also note which squares on GIVEN have white pieces on them.

```
******label GivenWV;
******simplify Vsq.(WVALUES Valueon(GIVEN sq)>(sq=BKR1vsq=BQ1vsq=BQ2vsq=WQR2vsq=WQN3vsq=WQB2vsq=WQ3vsq=WKB2vsq
=WKN3vsq=WKR2));
2 Vsq.(WVALUES Valueon(GIVEN,sq)>(sq=BKR1v(sq=BQ1v(sq=BQ2v(sq=WQR2v(sq=WQN3v(sq=WQB2v(sq=WQ3v(sq=WKB2v(sq=
WKN3vsq=WKR2)))))))))
```

Section B.2 "Blocked on the Total Board, Too"

This lemma proves a precondition for the *CORNER* theorem. That theorem assumes that a piece checks the opposing king, and that the check did not arise from a castle, en passant capture, or pawn promotion. We wish to show that the checking piece does not have any entry square to move to make the check that does not also check the king. (Starting a move with the opposing king in check is clearly impossible.) The theorem will then let us conclude that the check was a discovered check.

Unfortunately, the theorem must speak of the total boards of the relevant positions. We do not have a total board, rather, a (fairly complete) board fragment. What can we conclude of the relationship between these two? We would like the appropriate predicates (MOVETO, Valueon) to correspond between the two boards. But proving this is some work. This illustrates the difficultly of transferring properties between similar objects within our formalism.

Our lemma begins with a lemma of its own. One of the orthogonality theorems, *OrthoThmX* (section A.9.2) states that the ORTHO relation (on a board, two squares lie on the same horizontal or vertical, with no pieces between them) is sometimes equivalent between a board and its sub-board. More specifically, ORTHO will remain true around a square if that square has no undefined squares sharing its rows and columns. The lemma considers orthogonality between GIVEN and the total board (Tboard) of some position for which GIVEN is a BOARD. A square sq is proposed, which shares neither a row nor a column with the *x-ed* square, WKR4. As this is the only undefined square on GIVEN, then orthogonality is equivalent, on this square, between the two boards. We label this lemma *BLOCKLEM* (blocking lemma).

```
******label L1;
*******assume ¬(Row WKR4=Row sq)∧¬(Column WKR4=Column sq);
1 ¬(Row WKR4=Row sq)∧¬(Column WKR4=Column sq)  (1)

*******assume sq3=WKR4;
2 sq3=WKR4  (2)

*******subst † in ††;
3 ¬(Row sq3=Row sq)∧¬(Column sq3=Column sq)  (1 2)

******⊃I ††⊃†;
4 sq3=WKR4⊃(¬(Row sq3=Row sq)∧¬(Column sq3=Column sq))  (1)

******∀E GivenUD sq3;
5 Valueon(GIVEN,sq3)=UD∨sq3=WKR4

******taut Valueon(GIVEN,sq3)=UD⊃(sq3=sqv(¬(Row sq3=Row sq)∧¬(Column sq3=Column sq)))†,††;
6 Valueon(GIVEN,sq3)=UD⊃(sq3=sqv(¬(Row sq3=Row sq)∧¬(Column sq3=Column sq)))  (1)

******∀I † sq3;
7 ∀sq3.(Valueon(GIVEN,sq3)=UD⊃(sq3=sqv(¬(Row sq3=Row sq)∧¬(Column sq3=Column sq))))  (1)

******∀E OrthoThmX q,GIVEN,sq,sq1;
8 BOARD(q,GIVEN)⊃(∀sq3.(Valueon(GIVEN,sq3)=UD⊃(sq3=sqv(¬(Row sq3=Row sq)∧¬(Column sq3=Column sq))))⊃(ORTHO(
Tboard q,sq,sq1)≡ORTHO(GIVEN,sq,sq1)))

******taut BOARD(q,GIVEN)⊃(ORTHO(Tboard q,sq,sq1)≡ORTHO(GIVEN,sq,sq1)) ††,†;
9 BOARD(q,GIVEN)⊃(ORTHO(Tboard q,sq,sq1)≡ORTHO(GIVEN,sq,sq1))  (1)

******⊃I L1⊃†;
10 (¬(Row WKR4=Row sq)∧¬(Column WKR4=Column sq))⊃(BOARD(q,GIVEN)⊃(ORTHO(Tboard q,sq,sq1)≡ORTHO(GIVEN,sq,sq1))
)

******label BLOCKLEM;
******∀I † q sq sq1;
11 ∀q sq sq1.((¬(Row WKR4=Row sq)∧¬(Column WKR4=Column sq))⊃(BOARD(q,GIVEN)⊃(ORTHO(Tboard q,sq,sq1)≡ORTHO(
GIVEN,sq,sq1))))
```

We know that for rooks, the MOVETO predicate is equivalent to the ORTHO predicate.

```
******∀E MOVING1 Tboard q,RW,BQ2,sq1;
12 MOVETO(Tboard q,RW,BQ2,sq1)≡((VALUER RW∧ORTHO(Tboard q,BQ2,sq1))∨((VALUEB RW∧DIAG(Tboard q,BQ2,sq1))∨((
VALUEQ RW∧ORTHO(Tboard q,BQ2,sq1))∨((VALUEQ RW∧DIAG(Tboard q,BQ2,sq1))∨((VALUEK RW∧KINGMOVE(BQ2,sq1))∨((
VALUEN RW∧KNIGHTMOVE(BQ2,sq1))∨(VALUEP RW∧PAWNMOVE(Tboard q,RW,BQ2,sq1)))))))

******∀E MOVING1 Tboard q,RW,BQN2,sq1;
13 MOVETO(Tboard q,RW,BQN2,sq1)≡((VALUER RW∧ORTHO(Tboard q,BQN2,sq1))∨((VALUEB RW∧DIAG(Tboard q,BQN2,sq1))∨((
VALUEQ RW∧ORTHO(Tboard q,BQN2,sq1))∨((VALUEQ RW∧DIAG(Tboard q,BQN2,sq1))∨((VALUEK RW∧KINGMOVE(BQN2,sq1))∨((
VALUEN RW∧KNIGHTMOVE(BQN2,sq1))∨(VALUEP RW∧PAWNMOVE(Tboard q,RW,BQN2,sq1)))))))

******label L2;
******simplify ††;
14 MOVETO(Tboard q,RW,BQ2,sq1)≡ORTHO(Tboard q,BQ2,sq1)

******simplify ††;
15 MOVETO(Tboard q,RW,BQN2,sq1)≡ORTHO(Tboard q,BQN2,sq1)
```

And, by our lemma above, on the interesting squares (BQ2 and BQN2) orthogonality is equivalent between GIVEN and the total board.

```
******VE BLOCKLEM q,BQ2,sq1;
16 (¬(Row WKR4=Row BQ2)∧¬(Column WKR4=Column BQ2))⊃(BOARD(q,GIVEN)⊃(ORTHO(Tboard q,BQ2,sq1)≡ORTHO(GIVEN,BQ2,
sq1)))

******VE BLOCKLEM q,BQN2,sq1;
17 (¬(Row WKR4=Row BQN2)∧¬(Column WKR4=Column BQN2))⊃(BOARD(q,GIVEN)⊃(ORTHO(Tboard q,BQN2,sq1)≡ORTHO(GIVEN,
BQN2,sq1)))

******label L3;
******simplify ††;
18 BOARD(q,GIVEN)⊃(ORTHO(Tboard q,BQ2,sq1)≡ORTHO(GIVEN,BQ2,sq1))

******simplify ††;
19 BOARD(q,GIVEN)⊃(ORTHO(Tboard q,BQN2,sq1)≡ORTHO(GIVEN,BQN2,sq1))
```

We would have our result, except that we must prove that the rook could not have moved from the undefined square. But no piece can make that giant L jump, as consultation to a move excluding theorem shows.

```
******VE MayMove Tboard q,RW,BQ2,WKR4;
20 MOVETO(Tboard q,RW,BQ2,WKR4)⊃(Column BQ2=Column WKR4∨(KNIGHTMOVE(BQ2,WKR4)∨(Row BQ2=Row WKR4∨(SAMEDIAG(BQ2
,WKR4)∨(KINGMOVE(BQ2,WKR4)∨(TWOTOUCHING(Column BQ2 Column WKR4)∧(WSUC(Row BQ2,Row WKR4)∨BSUC(Row BQ2,Row WKR4
))))))))

******label L4;
******simplify †;
21 ¬MOVETO(Tboard q,RW,BQ2,WKR4)
```

We can see the desired result is true on the given board. The steps above prove the equivalence between this observation, and the result on the total board.

```
******simplify Vsq2.(ORTHO(GIVEN,BQ2,sq2)⊃(¬(Valueon(GIVEN,sq2)=MT)∨ORTHO(GIVEN,BQN2,sq2)));
22 Vsq2.(ORTHO(GIVEN,BQ2,sq2)⊃(¬(Valueon(GIVEN,sq2)=MT)∨ORTHO(GIVEN,BQN2,sq2)))

******label L5;
******VE † sq1;
23 ORTHO(GIVEN,BQ2,sq1)⊃(¬(Valueon(GIVEN,sq1)=MT)∨ORTHO(GIVEN,BQN2,sq1))

******VE SubBoards4X q GIVEN sq1;
24 BOARD(q,GIVEN)⊃(Valueon(GIVEN,sq1)=Valueon(Tboard q,sq1,Valueon(GIVEN,sq1,UD)

******VE GivenUD sq1;
25 Valueon(GIVEN,sq1)=UD≡sq1=WKR4

******assume BOARD(q,GIVEN);
26 BOARD(q,GIVEN)  (26)

******tauseq MOVETO(Tboard q RW BQ2 sq1)⊃(¬Valueon(Tboard q sq1)=MT ∨MOVETO(Tboard q RW BQN2 sq1)) L2,L2+1,L3,
*L3+1,L4:†;
27 MOVETO(Tboard q,RW,BQ2,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RW,BQN2,sq1))  (26)
```

Generalization and removal of dependencies.

```
******VI † sq1;
28 Vsq1.(MOVETO(Tboard q,RW,BQ2,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RW,BQN2,sq1)))  (26)

******⊃I †††⊃†;
29 BOARD(q,GIVEN)⊃Vsq1.(MOVETO(Tboard q,RW,BQ2,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RW,BQN2,sq1)
))
```

```
******label BlockedGivenThm;
******VI ↑ q;
38 Vq.(BOARD(q,GIVEN)⊃Vsq1.(MOVETO(Tboard q,RW,BQ2,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RW,BQN2,
sq1)))
```

Section B.3 Where A White Pawn on BQB2 Goes

A lemma for the main proof, to derive the possible moves of the promoting pawn on BQB2. Naturally, we turn first to the move defining axioms, MCONSEQ.

```
******label L1;
******assume SUCCESSOR(r,p)∧(¬CASTLING(r,p)∧(¬EN_PASSANT(r,p)∧(PAWNPROM Move p∧(¬WHITETURN p∧(From Move p=BQB2
∧BOARD(p,b))))));
1 SUCCESSOR(r,p)∧(¬CASTLING(r,p)∧(¬EN_PASSANT(r,p)∧(PAWNPROM Move p∧(¬WHITETURN p∧(From Move p=BQB2∧BOARD(p,b
))))))  (1)

******VE MconseqIX r,p;
2 SUCCESSOR(r,p)⊃(PAWNPROM Move p≡(LASTRANKER(To Move p,Color r)∧(SIMPLELEGALMOVE(r,p)∧(PAWNS Mover Move p∧(
VALUEP Valueon(Tboard r,From Move p)∧(((BVALUES Promoted Move p≡BVALUES Val(r,Mover Move p))∧(WVALUES
Promoted Move p≡WVALUES Val(r,Mover Move p)))∧Val(p,Mover Move p)=Promoted Move p))))))

******VE MCONSEQK r,p;
3 SIMPLELEGALMOVE(r,p)≡(¬(From Move p=To Move p)∧(MOVETO(Tboard r,Valueon(Tboard r,From Move p),From Move p,
To Move p)∧((SIMPLE Move p∧Valueon(Tboard r,To Move p)=MT)∨(CAPTURE Move p∧(PIECEVALUES Valueon(Tboard r,To
Move p)∧¬(Valuecolor Valueon(Tboard r,To Move p)=Color r))))))

******VE MCONSEQA r,p;
4 SUCCESSOR(r,p)⊃((¬WHITETURN r≡WHITETURN p)∧(Prevpos p=r∧(¬POSITIONINCHECK(p,Color r)∧(((WHITEPIECE Mover
Move p≡WHITETURN r)∧(Pos(r,From Move p)=Mover Move p∧(Pos(p,To Move p)=Mover Move p∧(Pos(p,From Move p)=EMPTY
∧((CAPTURE Move p⊃Pos(r,To Move p)=Taken Move p)∧(CASTLING(r,p)∨(EN_PASSANT(r,p)∨SIMPLELEGALMOVE(r,p))))))))))
))
```

We not that the Val, Valueon, and Pos functions all express different representations of similar objects, and that these representations are intimately connected.

```
******VE ValueTranspositionA r,Mover Move p,From Move p;
5 Pos(r,From Move p)=Mover Move p⊃Valueon(Tboard r,From Move p)=Val(r,Mover Move p)
```

One awkward point is the necessity of reminding the proof checker of the differences between black and white.

```
******VE BorW_Piece_ Mover Move p;
6 ¬(BLACKPIECE Mover Move p≡WHITEPIECE Mover Move p)
```

And we can see that any pawn valued white value piece must be PW.

```
******simplify Vv.((VALUEP v∧¬BVALUES v)≡v=PW);
7 Vv.((VALUEP v∧¬BVALUES v)≡v=PW)

******VE ↑ Valueon(Tboard r,From Move p);
8 (VALUEP Valueon(Tboard r,From Move p)∧¬BVALUES Valueon(Tboard r,From Move p))≡Valueon(Tboard r,From Move p)
=PW

******label L2;
******VE ColorChoices r,Mover Move p;
9 (BVALUES Val(r,Mover Move p)≡BLACKPIECE Mover Move p)∧(WVALUES Val(r,Mover Move p)≡WHITEPIECE Mover Move p)
```

More specifically, we consult the definitions of the PW's moves, and simplify the result.

```
******VE MOVING1 Tboard r,PW,From Move p,To Move p;
18 MOVETO(Tboard r,PW,From Move p,To Move p)≡((VALUER PW∧ORTHO(Tboard r,From Move p,To Move p))∨((VALUEB PW∧
DIAG(Tboard r,From Move p,To Move p))∨((VALUEQ PW∧ORTHO(Tboard r,From Move p,To Move p))∨((VALUEQ PW∧DIAG(
Tboard r,From Move p,To Move p))∨((VALUEK PW∧KINGMOVE(From Move p,To Move p))∨((VALUEN PW∧KNIGHTMOVE(From
Move p,To Move p))∨(VALUEP PW∧PAWNMOVE(Tboard r,PW,From Move p,To Move p))))))))

******simplify ↑;
11 MOVETO(Tboard r,PW,From Move p,To Move p)≡PAWNMOVE(Tboard r,PW,From Move p,To Move p)

******VE PAWNMOVING1 Tboard r,PW,From Move p,To Move p;
12 PAWNMOVE(Tboard r,PW,From Move p,To Move p)≡((WPAWNMOVE(Tboard r,From Move p,To Move p)∧WVALUES PW)∨(
BPAWNMOVE(Tboard r,From Move p,To Move p)∧BVALUES PW))

******simplify ↑;
13 PAWNMOVE(Tboard r,PW,From Move p,To Move p)≡WPAWNMOVE(Tboard r,From Move p,To Move p)

******VE PAWNMOVING2 Tboard r,From Move p,To Move p;
14 WPAWNMOVE(Tboard r,From Move p,To Move p)≡((Column From Move p=Column To Move p∧(WSUC(Row From Move p,Row
To Move p)∧Valueon(Tboard r,To Move p)=MT))∨((Column From Move p=Column To Move p∧(Row From Move p=7∧(Valueon
(Tboard r,To Move p)=MT∧(Valueon(Tboard r,Makesquare(6,Column From Move p))=MT∧Row To Move p=5))))∨(
TWOTOUCHING(Column From Move p,Column To Move p)∧(WSUC(Row From Move p,Row To Move p)∧BVALUES Valueon(Tboard
r,To Move p)))))
```

It therefore tautologically follows that a white pawn can move in only one of three (ordinary) motions.

```
******tauteq ↑:#2 L1:↑;
15 (Column From Move p=Column To Move p∧(WSUC(Row From Move p,Row To Move p)∧Valueon(Tboard r,To Move p)=MT))
∨((Column From Move p=Column To Move p∧(Row From Move p=7∧(Valueon(Tboard r,To Move p)=MT∧(Valueon(Tboard r,
Makesquare(6,Column From Move p))=MT∧Row To Move p=5))))∨(TWOTOUCHING(Column From Move p,Column To Move p)∧(
WSUC(Row From Move p,Row To Move p)∧BVALUES Valueon(Tboard r,To Move p))))  (1)
```

Substitution the square we know it moved from, and simplifying, we get:

```
******taut From Move p=BQB2 L1;
16 From Move p=BQB2  (1)

******substr ↑ in ↑↑;
17 (Column BQB2=Column To Move p∧(WSUC(Row BQB2,Row To Move p)∧Valueon(Tboard r,To Move p)=MT))∨((Column BQB2
=Column To Move p∧(Row BQB2=7∧(Valueon(Tboard r,To Move p)=MT∧(Valueon(Tboard r,Makesquare(6,Column BQB2))=MT
∧Row To Move p=5))))∨(TWOTOUCHING(Column BQB2,Column To Move p)∧(WSUC(Row BQB2,Row To Move p)∧BVALUES Valueon
(Tboard r,To Move p))))  (1)

******label L3;
******simplify ↑;
18 (3=Column To Move p∧(WSUC(2,Row To Move p)∧Valueon(Tboard r,To Move p)=MT))∨(TWOTOUCHING(3,Column To Move
p)∧(WSUC(2,Row To Move p)∧BVALUES Valueon(Tboard r,To Move p)))  (1)
```

Now, we check all of the squares, and discover only three possible destinations for our promoting white pawn.

```
******simplify ∀sq.(((3=Column sq∧WSUC(2,Row sq))⊃sq=BQB1)∧((TWOTOUCHING(3,Column sq)∧WSUC(2,Row sq))⊃(sq=BQN1
∨sq=BQ1)));
19 ∀sq.(((3=Column sq∧WSUC(2,Row sq))⊃sq=BQB1)∧((TWOTOUCHING(3,Column sq)∧WSUC(2,Row sq))⊃(sq=BQN1∨sq=BQ1)))

******VE ↑ To Move p;
20 ((3=Column To Move p∧WSUC(2,Row To Move p))⊃To Move p=BQB1)∧((TWOTOUCHING(3,Column To Move p)∧WSUC(2,Row
To Move p))⊃(To Move p=BQN1∨To Move p=BQ1))
```

A little rearrangement of the result, molding it into a more convenient form for the main proof. Effectively, we clutter this lemma with substitutions, rather than the main proof.

```
******VE VALUETRANSPOSITION p,Mover Move p,To Move p,b;
21 (Pos(p,To Move p)=Mover Move p∧BOARD(p,b))⊃(Valueon(b,To Move p)=Val(p,Mover Move p)∨Valueon(b,To Move p)=
UD)

******tauteq WVALUES Valueon (b To Move p)∨Valueon(b To Move p)=UD †,L1↓L1+3,L2;
22 WVALUES Valueon(b,To Move p)∨Valueon(b,To Move p)=UD  (1)

******assume To Move p=sqx;
23 To Move p=sqx  (23)

******substr † in ††;
24 WVALUES Valueon(b,sqx)∨Valueon(b,sqx)=UD  (1 23)

******⊃I †† ⊃ †;
25 To Move p=sqx⊃(WVALUES Valueon(b,sqx)∨Valueon(b,sqx)=UD)  (1)

******∀I † sqx;
26 ∀sqx.(To Move p=sqx⊃(WVALUES Valueon(b,sqx)∨Valueon(b,sqx)=UD))  (1)

******∀E † BQN1;
27 To Move p=BQN1⊃(WVALUES Valueon(b,BQN1)∨Valueon(b,BQN1)=UD)  (1)

******∀E †† BQB1;
28 To Move p=BQB1⊃(WVALUES Valueon(b,BQB1)∨Valueon(b,BQB1)=UD)  (1)

******∀E ††† BQ1;
29 To Move p=BQ1⊃(WVALUES Valueon(b,BQ1)∨Valueon(b,BQ1)=UD)  (1)

******tauteq (To Move p=BQN1∧((WVALUES Valueon(b,BQN1)∨Valueon(b,BQN1)=UD)∧BVALUES Valueon(Tboard r,To Move p)
s))∨((To Move p=BQ1∧((WVALUES Valueon(b,BQ1)∨Valueon(b,BQ1)=UD)∧BVALUES Valueon(Tboard r,To Move p)))∨(To
sMove p=BQB1∧(WVALUES Valueon(b,BQB1)∨Valueon(b,BQB1)=UD)))  †††↓†,18,28;
30 (To Mo e p=BQN1∧((WVALUES Valueon(b,BQN1)∨Valueon(b,BQN1)=UD)∧BVALUES Valueon(Tboard r,To Move p)))∨((To
Move p=BQ1∧((WVALUES Valueon(b,BQ1)∨Valueon(b,BQ1)=UD)∧BVALUES Valueon(Tboard r,To Move p)))∨(To Move p=BQB1∧
(WVALUES Valueon(b,BQB1)∨Valueon(b,BQB1)=UD)))  (1)
```

Removing dependencies, and generalizing, we get our lemma.

```
******⊃I L1⊃†;
31 (SUCCESSOR(r,p)∧(¬CASTLING(r,p)∧(¬EN_PASSANT(r,p)∧(PAWNPROM Move p∧(¬WHITETURN p∧(From Move p=BQB2∧BOARD(p
,b)))))))⊃((To Move p=BQN1∧((WVALUES Valueon(b,BQN1)∨Valueon(b,BQN1)=UD)∧BVALUES Valueon(Tboard r,To Move p))
)∨((To Move p=BQ1∧((WVALUES Valueon(b,BQ1)∨Valueon(b,BQ1)=UD)∧BVALUES Valueon(Tboard r,To Move p)))∨(To Move
p=BQB1∧(WVALUES Valueon(b,BQB1)∨Valueon(b,BQB1)=UD))))

******label PXPawnTo;
******∀I † r p b;
32 ∀r p b.((SUCCESSOR(r,p)∧(¬CASTLING(r,p)∧(¬EN_PASSANT(r,p)∧(PAWNPROM Move p∧(¬WHITETURN p∧(From Move p=BQB2
∧BOARD(p,b)))))))⊃((To Move p=BQN1∧((WVALUES Valueon(b,BQN1)∨Valueon(b,BQN1)=UD)∧BVALUES Valueon(Tboard r,To
Move p)))∨((To Move p=BQ1∧((WVALUES Valueon(b,BQ1)∨Valueon(b,BQ1)=UD)∧BVALUES Valueon(Tboard r,To Move p)))∨(
To Move p=BQB1∧(WVALUES Valueon(b,BQB1)∨Valueon(b,BQB1)=UD)))))
```

Section B.4 A Rook or Queen on BQ1 is Cornered

The purpose of this lemma is to set up one of the more complicated conditions of the corner theorem for the main proof. Effectively, somewhat similar in direction to the proof of section B.2, though a bit more complicated.

We seek to prove that, for any position q, which has a a board QBUD, that a black rook or queen valued piece on BQ1 can move only to squares that still check the white king on BKR1. We will use this conclusion in the main proof to show that if the piece the pawn captured was a black rook or queen, then that checking rook or queen must have arrived at that state by a discovered check.

We begin, of course, by assuming the existence of a position q, which has QBUD as a board.

```
******label L1;
***** assume BOARD(q,QBUD);
1 BOARD(q,QBUD)  (1)
```

We consider first the case of diagonal moves. As QBUD is a board of this position, and, as we can observe, QBUD is well defined and not empty on the two diagonal blocking squares (BQB2 and BK2), we know that the total board (Tboard) of q must also have those squares occupied.

```
******VE SubBoards4X q,QBUD,BQB2;
2 BOARD(q,QBUD)⊃(Valueon(QBUD,BQB2)=Valueon(Tboard q,BQB2)∨Valueon(QBUD,BQB2)=UD)

******VE SubBoards4X q,QBUD,BK2;
3 BOARD(q,QBUD)⊃(Valueon(QBUD,BK2)=Valueon(Tboard q,BK2)∨Valueon(QBUD,BK2)=UD)

******simplify ¬(Valueon(QBUD,BK2)=MT)∧(¬(Valueon(QBUD,BK2)=UD)∧(¬(Valueon(QBUD,BQB2)=UD)∧¬(Valueon(QBUD,BQB2)
*=MT)));
4 ¬(Valueon(QBUD,BK2)=MT)∧(¬(Valueon(QBUD,BK2)=UD)∧(¬(Valueon(QBUD,BQB2)=UD)∧¬(Valueon(QBUD,BQB2)=MT)))

******label L3;
***** tauteq ¬(Valueon(Tboard q,BQB2)=MT)∧¬(Valueon(Tboard q,BK2)=MT) L1;↑;
5 ¬(Valueon(Tboard q,BQB2)=MT)∧¬(Valueon(Tboard q,BK2)=MT)  (1)
```

Of course, TAUTEQ will not be pleased unless we do its function substitutions for it.

```
******assume sq1=BQB2;
6 sq1=BQB2  (6)

******assume sq1=BK2;
7 sq1=BK2  (7)

******subst ↑↑ in ↑↑↑;
8 ¬(Valueon(Tboard q,sq1)=MT)∧¬(Valueon(Tboard q,BK2)=MT)  (1 6)

******subst ↑↑ in ↑↑↑↑;
9 ¬(Valueon(Tboard q,BQB2)=MT)∧¬(Valueon(Tboard q,sq1)=MT)  (1 7)

******⊃I ↑↑↑↑⊃↑↑;
10 sq1=BQB2⊃(¬(Valueon(Tboard q,sq1)=MT)∧¬(Valueon(Tboard q,BK2)=MT))  (1)

******⊃I ↑↑↑↑⊃↑↑;
11 sq1=BK2⊃(¬(Valueon(Tboard q,BQB2)=MT)∧¬(Valueon(Tboard q,sq1)=MT))  (1)
```

We employ a lemma, *DiagBQ1Lemma*, which states that for any position with chesspiece on BQB2 and BK2, there are no diagonal moves from BQ1. *DiagBQ1Lemma* is proven in the next subsection.

```
******VE DiagBQ1Lemma Tboard q,sq1;
12 ¬(DIAG(Tboard q,BQ1,sq1)∧(¬(sq1=BQB2)∧(¬(sq1=BK2)∧(¬(Valueon(Tboard q,BQB2)=MT)∧¬(Valueon(Tboard q,BK2)=MT
)))))
```

We can therefore tautologically conclude that, on the total board of q, the only squares diagonally reachable from BQ1 are occupied.

```
******label L4;
***** tauteq ¬DIAG(Tboard q,BQ1,sq1)v¬(Valueon(Tboard q,sq1)=MT) L3,↑↑↑;↑;
13 ¬DIAG(Tboard q,BQ1,sq1)v¬(Valueon(Tboard q,sq1)=MT)  (1)
```

We proceed to the orthogonal cases. The board QBUD is undefined on exactly two squares, BQ1 and WKR4 (the fallen piece square). Essentially, we seek those conditions which must be true of a sub-board, before we can deduce that movement relations on it are equivalent to those on a more defined board. We have a theorem, *OrthoThmX*[87] that related orthogonality between sub-boards and total boards. It states that undefined squares that neither share a row nor a column with the given square (or are equal to it) do not effect the ORTHO relation. We seek to establish that the undefined squares of QBUD (BQ1 and WKR4) are not relevant to the current orthogonality question. As usual, we require copious substitutions.

```
******simplify ¬(Row WKR4=Row BQ1)∧¬(Column WKR4=Column BQ1);
14 ¬(Row WKR4=Row BQ1)∧¬(Column WKR4=Column BQ1)

******assume sq3=WKR4;
15 sq3=WKR4  (15)

******subst ↑ in ↑↑;
16 ¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1)  (15)

******⊃I ↑↑⊃↑;
17 sq3=WKR4⊃(¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1))

******simplify ∀sq.(Valueon(QBUD,sq)=UD⊃(sq=BQ1∨sq=WKR4));
18 ∀sq.(Valueon(QBUD,sq)=UD⊃(sq=BQ1∨sq=WKR4))

*****∀E ↑ sq3;
19 Valueon(QBUD,sq3)=UD⊃(sq3=BQ1∨sq3=WKR4)

*****tauteq Valueon(QBUD,sq3)=UD⊃(sq3=BQ1∨(¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1))) ↑,↑↑↑;
20 Valueon(QBUD,sq3)=UD⊃(sq3=BQ1∨(¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1)))

******label L5;
***** ∀I ↑ sq3;
21 ∀sq3.(Valueon(QBUD,sq3)=UD⊃(sq3=BQ1∨(¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1))))
```

We invoke our chess eye, the simplification mechanism, to show that BQ1 is orthogonally cornered, and adequately defined.

```
******simplify ∀sq.(ORTHO(QBUD,BQ1,sq)⊃(ORTHO(QBUD,BKR1,sq)∨(¬(Valueon(QBUD,sq)=UD)∧¬(Valueon(QBUD,sq)=MT))));
22 ∀sq.(ORTHO(QBUD,BQ1,sq)⊃(ORTHO(QBUD,BKR1,sq)∨(¬(Valueon(QBUD,sq)=UD)∧¬(Valueon(QBUD,sq)=MT))))

******label L6;
***** ∀E ↑ sq1;
23 ORTHO(QBUD,BQ1,sq1)⊃(ORTHO(QBUD,BKR1,sq1)∨(¬(Valueon(QBUD,sq1)=UD)∧¬(Valueon(QBUD,sq1)=MT)))
```

And, of course, invocations of various theorems to show the equivalence of our different representations.

```
******∀E OrthoThmX q,QBUD,BQ1,sq1;
```

---

87. Section A.9.2.1.

```
24 BOARD(q,QBUD)⊃(∀sq3.(Valueon(QBUD,sq3)≠UD⊃(sq3=BQ1∨(¬(Row sq3=Row BQ1)∧¬(Column sq3=Column BQ1))))⊃(ORTHO(
Tboard q,BQ1,sq1)≡ORTHO(QBUD,BQ1,sq1)))

*****VE TransitiveSubboardOrthogonality QBUD,Tboard q,BKR1,sq1;
25 SUBOARD(QBUD,Tboard q)⊃(ORTHO(QBUD,BKR1,sq1)⊃ORTHO(Tboard q,BKR1,sq1))

*****VE SUB_BOARDS3 q,QBUD;
26 BOARD(q,QBUD)≡SUBOARD(QBUD,Tboard q)

*****label L7;
***** VE SubBoards4X q QBUD sq1;
27 BOARD(q,QBUD)⊃(Valueon(QBUD,sq1)≠Valueon(Tboard q,sq1)∨Valueon(QBUD,sq1)≠UD)
```

We have enough here to show the desired relationships for orthogonality and diagonality. However, the theorem stipulates the relation be on MOVETO, not some lesser predicate. So we must show the adequacy of our deduction for each of the rook and the queen. Ladies first.

MOVETO is defined by the axiom *MOVING1*. Simplification narrows the choices.

```
*****VE MOVING1 Tboard q,QB,BQ1,sq1;
28 MOVETO(Tboard q,QB,BQ1,sq1)≡((VALUER QB∧ORTHO(Tboard q,BQ1,sq1))∨((VALUEB QB∧DIAG(Tboard q,BQ1,sq1))∨((
VALUEQ QB∧ORTHO(Tboard q,BQ1,sq1))∨((VALUEQ QB∧DIAG(Tboard q,BQ1,sq1))∨((VALUEK QB∧KINGMOVE(BQ1,sq1))∨((
VALUEN QB∧KNIGHTMOVE(BQ1,sq1))∨(VALUEP QB∧PAWNMOVE(Tboard q,QB,BQ1,sq1)))))))))

*****VE MOVING1 Tboard q,QB,BKR1,sq1;
29 MOVETO(Tboard q,QB,BKR1,sq1)≡((VALUER QB∧ORTHO(Tboard q,BKR1,sq1))∨((VALUEB QB∧DIAG(Tboard q,BKR1,sq1))∨((
VALUEQ QB∧ORTHO(Tboard q,BKR1,sq1))∨((VALUEQ QB∧DIAG(Tboard q,BKR1,sq1))∨((VALUEK QB∧KINGMOVE(BKR1,sq1))∨((
VALUEN QB∧KNIGHTMOVE(BKR1,sq1))∨(VALUEP QB∧PAWNMOVE(Tboard q,QB,BKR1,sq1)))))))))

******simplify ††;
30 MOVETO(Tboard q,QB,BQ1,sq1)≡(ORTHO(Tboard q,BQ1,sq1)∨DIAG(Tboard q,BQ1,sq1))

******simplify ††;
31 MOVETO(Tboard q,QB,BKR1,sq1)≡(ORTHO(Tboard q,BKR1,sq1)∨DIAG(Tboard q,BKR1,sq1))
```

We can then tautologically conclude the desired WFF for the black queen (valued) piece.

```
*****label QUEENMOVE;
***** tauteq MOVETO(Tboard q,QB,BQ1,sq1)⊃ (¬(Valueon(Tboard q,sq1)≠NT)∨MOVETO(Tboard q,QB,BKR1,sq1)) L1,L4,L5
*,L6:L7,††:↑;
32 MOVETO(Tboard q,QB,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)≠NT)∨MOVETO(Tboard q,QB,BKR1,sq1))  (1)
```

Similarly for the rook.

```
*****VE MOVING1 Tboard q,RB,BQ1,sq1;
33 MOVETO(Tboard q,RB,BQ1,sq1)≡((VALUER RB∧ORTHO(Tboard q,BQ1,sq1))∨((VALUEB RB∧DIAG(Tboard q,BQ1,sq1))∨((
VALUEQ RB∧ORTHO(Tboard q,BQ1,sq1))∨((VALUEQ RB∧DIAG(Tboard q,BQ1,sq1))∨((VALUEK RB∧KINGMOVE(BQ1,sq1))∨((
VALUEN RB∧KNIGHTMOVE(BQ1,sq1))∨(VALUEP RB∧PAWNMOVE(Tboard q,RB,BQ1,sq1)))))))))

*****VE MOVING1 Tboard q,RB,BKR1,sq1;
34 MOVETO(Tboard q,RB,BKR1,sq1)≡((VALUER RB∧ORTHO(Tboard q,BKR1,sq1))∨((VALUEB RB∧DIAG(Tboard q,BKR1,sq1))∨((
VALUEQ RB∧ORTHO(Tboard q,BKR1,sq1))∨((VALUEQ RB∧DIAG(Tboard q,BKR1,sq1))∨((VALUEK RB∧KINGMOVE(BKR1,sq1))∨((
VALUEN RB∧KNIGHTMOVE(BKR1,sq1))∨(VALUEP RB∧PAWNMOVE(Tboard q,RB,BKR1,sq1)))))))))

******simplify ††;
35 MOVETO(Tboard q,RB,BQ1,sq1)≡ORTHO(Tboard q,BQ1,sq1)

******simplify ††;
36 MOVETO(Tboard q,RB,BKR1,sq1)≡ORTHO(Tboard q,BKR1,sq1)
```

```
******label ROOKMOVE;
******tauteq MOVETO(Tboard q,RB,BQ1,sq1)⊃ (¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RB,BKR1,sq1)) L1,L5,L6:
*L7,↑↑:↑;
37 MOVETO(Tboard q,RB,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,RB,BKR1,sq1))  (1)
```

It will be more convenient for use to have the theorem in terms of some rook or queen valued v. So a little more fiddling.

```
*******assume v=RB∨v=QB;
38 v=RB∨v=QB  (38)

******tauteq QUEENMOVE:[QB←v] QUEENMOVE,ROOKMOVE,↑;
39 MOVETO(Tboard q,v,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,v,BKR1,sq1))  (1 38)

******∀I ↑ sq1;
40 ∀sq1.(MOVETO(Tboard q,v,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,v,BKR1,sq1)))  (1 38)
```

Removing dependencies, and generalizing, we obtain our theorem.

```
******⊃I ↑↑↑ ⊃ ↑;
41 (v=RB∨v=QB)⊃∀sq1.(MOVETO(Tboard q,v,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,v,BKR1,sq1)))  (
1)

******⊃I L1 ⊃ ↑;
42 BOARD(q,QBUD)⊃((v=RB∨v=QB)⊃∀sq1.(MOVETO(Tboard q,v,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(Tboard q,v
,BKR1,sq1))))

******label Trapped_QX_QB1_Thm;
***** ∀I ↑ q v;
43 ∀q v.(BOARD(q,QBUD)⊃((v=RB∨v=QB)⊃∀sq1.(MOVETO(Tboard q,v,BQ1,sq1)⊃(¬(Valueon(Tboard q,sq1)=MT)∨MOVETO(
Tboard q,v,BKR1,sq1)))))
```

Section B.4.1 Blocked Diagonal Movement

We promised a proof of the blocked diagonal movement lemma used in the previous section. We seek to prove that pieces on BK2 and BQB2 are sufficient to block bishop like movement (DIAG) from BQ1.

We observe that any square sharing a diagonal with BQ1 must either have BK2 or BQB2 on that diagonal, or be one of those squares.

```
******label L1;
******simplify ∀sq3.(SAMEDIAG(BQ1,sq3)⊃ ((BETWEEN(1,2,Row sq3)∧(SAMEDIAG(sq3,BK2)∨SAMEDIAG(sq3,BQB2)))∨(sq3=
*BQB2∨sq3=BK2)));
1 ∀sq3.(SAMEDIAG(BQ1,sq3)⊃((BETWEEN(1,2,Row sq3)∧(SAMEDIAG(sq3,BK2)∨SAMEDIAG(sq3,BQB2)))∨(sq3=BQB2∨sq3=BK2)))
```

Diagonal movement is defined in terms of the SAMEDIAG predicate.

```
******∀E MOVING3 b,BQ1,sq;
2 DIAG(b,BQ1,sq)≡(SAMEDIAG(BQ1,sq)∧∀sq3.((SAMEDIAG(BQ1,sq3)∧(SAMEDIAG(sq,sq3)∧BETWEEN(Row BQ1,Row sq3,Row sq)
))⊃Valueon(b,sq3)=MT))
```

We assume these squares are occupied, and that it is possible to diagonally move to some other square. We will show this supposition to be false.

```
******assume DIAG(b,BQ1,sq)∧(¬(sq=BQB2)∧(¬(sq=BK2)∧(¬(Valueon(b,BQB2)=MT)∧¬(Valueon(b,BK2)=MT))));
3 DIAG(b,BQ1,sq)∧(¬(sq=BQB2)∧(¬(sq=BK2)∧(¬(Valueon(b,BQB2)=MT)∧¬(Valueon(b,BK2)=MT))))  (3)
```

We abstract part of the definition of diagonal movement.

```
******taut Vsq3.((SAMEDIAG(BQ1,sq3)∧(SAMEDIAG(sq,sq3)∧ BETWEEN(Row BQ1,Row sq3,Row sq)))⊃Valueon(b,sq3)=MT) 2;
*3;
4 Vsq3.((SAMEDIAG(BQ1,sq3)∧(SAMEDIAG(sq,sq3)∧BETWEEN(Row BQ1,Row sq3,Row sq)))⊃Valueon(b,sq3)=MT)  (3)
```

This condition on diagonal movement is true for both BK2 and BQB2.

```
******VE ↑ BK2;
5 (SAMEDIAG(BQ1,BK2)∧(SAMEDIAG(sq,BK2)∧BETWEEN(Row BQ1,Row BK2,Row sq)))⊃Valueon(b,BK2)=MT  (3)

******VE ↑↑ BQB2;
6 (SAMEDIAG(BQ1,BQB2)∧(SAMEDIAG(sq,BQB2)∧BETWEEN(Row BQ1,Row BQB2,Row sq)))⊃Valueon(b,BQB2)=MT  (3)
```

We apply our original observation to the parameter square sq.

```
******VE L1 sq;
7 SAMEDIAG(BQ1,sq)⊃((BETWEEN(1,2,Row sq)∧(SAMEDIAG(sq,BK2)∨SAMEDIAG(sq,BQB2)))∨(sq=BQB2∨sq=BK2))
```

We compute the rows and columns of the relevant squares.

```
******simplify Row BQ1=1∧(Row BQB2=2∧(Row BK2=2∧(SAMEDIAG(BQ1,BK2)∧SAMEDIAG(BQ1,BQB2))));
8 Row BQ1=1∧(Row BQB2=2∧(Row BK2=2∧(SAMEDIAG(BQ1,BK2)∧SAMEDIAG(BQ1,BQB2))))
```

Which is enough to produce a contradiction. We negate our assumption, and generalize to our theorem.

```
******tautoq FALSE L1↑;
9 FALSE  (3)

******¬I ↑,DIAG(b,BQ1,sq)∧(¬(sq=BQB2)∧(¬(sq=BK2)∧(¬(Valueon(b,BQB2)=MT)∧¬(Valueon(b,BK2)=MT))));
10 ¬(DIAG(b,BQ1,sq)∧(¬(sq=BQB2)∧(¬(sq=BK2)∧(¬(Valueon(b,BQB2)=MT)∧¬(Valueon(b,BK2)=MT)))))

******label DiagBQ1Lemma;
******VI ↑ b sq;
11 Vb sq.¬(DIAG(b,BQ1,sq)∧(¬(sq=BQB2)∧(¬(sq=BK2)∧(¬(Valueon(b,BQB2)=MT)∧¬(Valueon(b,BK2)=MT)))))
```

Section B.4.2 Consequences of a Distant Pawn Promotion

Our final specific lemma. We apply the fact that every promotion square to the left of WKN1 requires the capture of two white pieces, and to express this fact in the form most convenient for the main proof.

To begin with, we need to convince the proof checker that each of the eighth row squares that aren't WKN1 or WKR1 require two captures from BKR2. The simplest way to convince the program, is to let it see for itself.

```
******simplify Vsq.(Row sq=8⊃(Pawncaptures(sq,BKR2)≥2∨sq=WKR1∨sq=WKN1));
1 Vsq.(Row sq=8⊃(Pawncaptures(sq,BKR2)≥2∨(sq=WKR1∨sq=WKN1)))
```

We assume we have the appropriate pawn promotion move. Hence, by the axioms of chess, this move must have been to the eighth row, and the piece must have had pawn value at the start of the move.

```
******label L1;
```

```
******assume PAWNPROM Move q∧Mover Move q=BKRP∧¬To Move q=WKR1∧¬To Move q=WKN1;
2 PAWNPROM Move q∧(Mover Move q=BKRP∧(¬(To Move q=WKR1)∧¬(To Move q=WKN1)))  (2)

******VE †† To Move q;
3 Row To Move q=8⊃(Pawncaptures(To Move q,BKR2)≥2∨(To Move q=WKR1∨To Move q=WKN1))

******VE BlackPromtesOn88 q,BKRP;
4 (PAWNPROM Move q∧Mover Move q=BKRP)⊃Row To Move q=8

******VE MCONSEQL q;
5 PAWNPROM Move q≡(LASTRANKER(To Move q,Color Prevpos q)∧(SIMPLELEGALMOVE(Prevpos q,q)∧(PAWNS Mover Move q∧(
VALUEP Valueon(Tboard Prevpos q,From Move q)∧(((BVALUES Promoted Move q≡BVALUES Val(Prevpos q,Mover Move q))∧
(WVALUES Promoted Move q≡WVALUES Val(Prevpos q,Mover Move q))/∧Val(q,Mover Move q)=Promoted Move q)))))

******VE ValueTranspositionR Prevpos q,BKRP,From Move q;
6 Pos(Prevpos q,From Move q)=BKRP⊃Valueon(Tboard Prevpos q,From Move q)=Val(Prevpos q,BKRP)

******VE MCONSEQR Prevpos q,q;
7 SUCCESSOR(Prevpos q,q)⊃((¬WHITETURN Prevpos q≡WHITETURN q)∧(Prevpos q=Prevpos q∧(¬POSITIONINCHECK(q,Color
Prevpos q)∧((WHITEPIECE Mover Move q≡WHITETURN Prevpos q)∧(Pos(Prevpos q,From Move q)=Mover Move q∧(Pos(q,To
Move q)=Mover Move q∧(Pos(q,From Move q)=EMPTY∧((CAPTURE Move q⊃Pos(Prevpos q,To Move q)=Taken Move q)∧(
CASTLING(Prevpos q,q)∨(EN_PASSANT(Prevpos q,q)∨SIMPLELEGALMOVE(Prevpos q,q))))))))))

******VE POSITION_RULES q;
8 SUCCESSOR(Prevpos q,q)∧PREDEGAME(P0,q)
```

The BKRP started the game on BKR2. If he has made at least two captures, then, of course, in two positions of this game, BKRP captured white pieces.

```
******simplify Pos(P0,BKR2);
9 Pos(P0,BKR2)=BKRP

******VE _PawnStructure_3 P0,q,BKRP,To Move q,BKR2;
10 (VALUEP Val(Prevpos q,BKRP)∧(Pos(q,To Move q)=BKRP∧(PREDEGAME(P0,q)∧(Pos(P0,BKR2)=BKRP∧Pawncaptures(To
Move q,BKR2)≥2))))⊃∃q1 q2 x1 x2.((PREDEGAME(q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧
(TAKINGS Move q2∧(Mover Move q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))

******tautoeq †:#2 L1:†;
11 ∃q1 q2 x1 x2.((PREDEGAME(q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2
∧(Mover Move q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))  (2)

******3E † q1,q2,x1,x2;
12 (PREDEGAME(q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move
q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2)))))))  (12)
```

These pieces were not the same piece.

```
******VE DifferentTaken q2,q1,q,x2,x1;
13 (((q1=q∨PREDEGAME(q1,q))∧(q2=q∨PREDEGAME(q2,q)))∧((¬(To Move q2=To Move q1)∨(¬(Mover Move q2=Mover Move q1
)∨(PREDEGAME(q2,q1)∨¬(q2=q1))))∧(Taken Move q2=x2∧Taken Move q1=x1)))⊃¬(x2=x1)
```

And both these captures occurred during the game that reach the presumed position.

```
******VE TransitiveGenealogy q2,q1,q;
14 (PREDEGAME(q2,q1)∧PREDEGAME(q1,q))⊃PREDEGAME(q2,q)
```

Hence, all the good things we desire of them are true.

```
******tautoeq †††:∧¬x1=x2 †††:†;
15 ((PREDEGAME(q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move
```

```
q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))∧¬(x1=x2)  (12)
```

Removing dependencies and quantify, we get our lemma.

```
*****3I ↑ x2 x1 q2 q1 ;
16 ∃q1 q2 x1 x2.(((PREDEGAME(q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move
q2∧(Mover Move q1=BKRP∧(Mover Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))∧¬(x1=x2))  (2)

*****⊃I L1⊃↑;
17 (PAWNPROM Move q∧(Mover Move q=BKRP∧(¬(To Move q=WKR1)∧¬(To Move q=WKN1))))⊃∃x2 x1 q2 q1.(((PREDEGAME(q1,q
)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move q1=BKRP∧(Mover Move
 q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))∧¬(x1=x2))

*****label FarTaken;
*****∀I ↑ q;
18 ∀q.((PAWNPROM Move q∧(Mover Move q=BKRP∧(¬(To Move q=WKR1)∧¬(To Move q=WKN1))))⊃∃x2 x1 q2 q1.(((PREDEGAME(
q1,q)∨q1=q)∧(PREDEGAME(q2,q1)∧(PREDEGAME(P0,q2)∧(TAKINGS Move q1∧(TAKINGS Move q2∧(Mover Move q1=BKRP∧(Mover
Move q2=BKRP∧(Taken Move q1=x1∧Taken Move q2=x2))))))))∧¬(x1=x2)))
```

Appendix C FOL Command Frequency

Command frequency for FOL rules of inference used in this proof, grouped by command type and use location.

Inf. rule	Main proof	M.P.&App B	Chap 3&App A	Total	Per cent
Quantifier manipulation:				1201	53%
∀E	191	242	724	966	43%
∀I	4	15	158	173	8%
∃I	4	5	20	25	1%
∃E	15	16	21	37	2%
Chess eye:				225	10%
SIMPLIFY	90	115	100	225	10%
Decision procedures:				381	17%
TAUTEQ	66	79	168	247	11%
TAUT	11	15	110	125	6%
UNIFY	2	2	7	9	0%
Substitution commands:				99	4%
SUBSTR	6	8	61	69	3%
SUBST	2	6	24	30	1%
Dependency introduction and removal:				322	14%
ASSUME	6	18	139	157	7%
⊃I	3	14	139	153	7%
¬I	1	2	9	11	0%
∨E	1	1	0	1	0%
Miscellaneous:				25	1%
∧I	3	3	13	16	1%
∧E	0	0	8	8	0%
■I	0	0	1	1	0%
Total:	405	541	1702	2253	
LABEL	77	103	302	405	

Appendix D A Constructive Solution to the Puzzle

To assauge the fears of those *still* not convinced that the fallen piece must have been a bishop, we present a constructive proof. More particularly, a game that reaches the given position.

1.	P-K4	P-KR4
2.	B-QR6	P×B
3.	N-KB3	P-KR5
4.	N-KN5	B-QN2
5.	N-K6	P/KB2×N
6.	K-K2	B-Q4
7.	P×B	P-QB4
8.	K-KB3	N-QB3
9.	K-KN4	Q-QB2
10.	K-KN5	O-O-O
11.	K-KN6	P-Q3
12.	P-Q3	R-Q2
13.	B-KN5	R-KR2
14.	Q-KB3	N-Q1
15.	Q×B	Q-QB3
16.	N-QR3	P-KR6
17.	R/KR1-KN1	N-KB3
18.	P×Q	N-Q4
19.	P-KN3	N-QN3
20.	R-N2	P×R
21.	R-K1	P-N8→N
22.	R-K4	N-KB6
23.	R-QN4	K-N1
24.	Q-KB5	K-R1
25.	Q-Q5	N-QB1
26.	R-N7	N-Q7
27.	R×R	P×Q
28.	N-QB4	P×N
29.	K×R	N-N8
30.	P-QN3	K-N1
31.	P-B7,ch	K-N2
32.	K-R8	K-N3
33.	B-KR4	K-N2
34.	P×N→R	

Appendix E Listing of the Chess Theorems

For the convenience of the reader, a list of the general chess lemmas and theorems used in this paper.

```
define AllStart_: ∀ t. ∃ sq. Pos (P0 sq) =t;;

define AllStartX: ∀ x. ∃ sq. Pospcf (P0, x) =sq;;

define AlternateBlack: ∀ r b. ((BOARD (r, b) ∧ BLACKINCHECK b) ⊃ (POSITIONINCHECK
(r, BLACK) ∧ ¬WHITETURN r));;

define AlternateWhite: ∀ r b. ((BOARD (r, b) ∧ WHITEINCHECK b) ⊃ (POSITIONINCHECK
(r, WHITE) ∧ WHITETURN r));;

define BishopMovementValues: ∀ r p ybi sq sq1. ((SUCCESSOR (r, p) ∧ (ybi=Mover
Move p ∧ (sq=To Move p ∧ sq1=From Move p))) ⊃ (MOVETO (Tboard r, Valueon (Tboard
r, From Move p) , From Move p, To Move p) ≡ DIAG (Tboard r, sq1, sq)));;

define BishopMoves: ∀ r p ybi sq sq1. ((SUCCESSOR (r, p) ∧ (ybi=Mover Move p ∧
(sq=To Move p ∧ sq1=From Move p))) ⊃ (MOVETO (Tboard r, Valueon (Tboard r, From
Move p) , From Move p, To Move p) ⊃ (WHITESQUARES (sq1) ≡ WHITESQUARES (sq))));;

define BishopStaysOnSameColor: ∀ r p ybi sq1 sq. ((SUCCESSOR (r, p) ∧ (Pos (p,
sq) =ybi ∧ Pos (r, sq1) =ybi)) ⊃ (WHITESQUARES (sq) ≡ WHITESQUARES (sq1)));;

define BishopsIsOnSameColor: ∀ r sq1 sq2 ybi. ((Pos (P0, sq1) =ybi ∧ Pos (r, sq2)
=ybi) ⊃ (WHITESQUARES (sq1) ≡ WHITESQUARES (sq2)));;

define BlackCapturedOnThm: ∀ r q y x sq. (Prevpos q=r ⊃ (To Move q=sq ⊃ (Mover
Move q=y ⊃ ((Taken Move q=x ∧ WHITEPIECE y) ⊃ (¬WHITEPIECE x ∧ (¬Row (sq) =3 ⊃
Pos (r sq) =x))))));;

define BlackCapturedThm: ∀ p sq. (To Move p=sq ⊃ ((ORDINARY Move p ∧ BVALUES
Valueon (Tboard Prevpos p, sq)) ⊃ CAPTURE Move p));;

define BlackCastleThm: ∀ r p sq. ((SUCCESSOR (r, p) ∧ (CASTLING (r, p) ∧
WHITETURN p)) ⊃ (Pos (p, sq) =BK ⊃ (sq=BKN1 ∨ sq=BQB1))) ;;

define BlackCheckingThm: ∀ r. ¬ (POSITIONINCHECK (r, BLACK) ∧ ¬BLACKINCHECK
Tboard r);;

define BlackCornered: ∀ r q b vb sq sqx. ((SUCCESSOR (r, q) ∧ (¬EN_PASSANT (r, q)
∧ (¬CASTLING (r, q) ∧ (¬BLASTRANK (sq) ∧ ((BOARD (q, b) ∧ (Valueon (b, sq) =vb ∧
(Valueon (b, sqx) =KW ∧ MOVETO (b, vb, sq, sqx)))) ∧ ¬VALUEP vb))))) ⊃ (∀ sq1.
(MOVETO (Tboard q, vb, sq, sq1) ⊃ (¬ (Valueon (Tboard q, sq1) =MT) ∨ MOVETO
(Tboard q, vb, sqx, sq1))) ⊃ ((ORDINARY Move q ∧ SQUARE_BETWEEN (sq, From Move q,
sqx)) ∧ ¬ (Mover Move q=Pos (q, sq))))));;

define BlackDidPromote: ∀ p ybp sq. (To Move p=sq ⊃ (Mover Move p=ybp ⊃ (PAWNPROM
Move p ⊃ (Val (Prevpos p, ybp) =PB ∧ Pos (p, sq) =ybp))));;

define BlackDoesNotStartInCheck_: ¬BLACKINCHECK START;;
```

```
define BlackEnPassantThm2: ∀ r q b. ((SUCCESSOR (r, q) ∧ (EN_PASSANT (r, q) ∧
WHITETURN q)) ⊃ (∀ dcx. ¬ (Valueon (b, Makesquare (6, dcx)) =PB ∨ Valueon (b,
Makesquare (6, dcx)) =UD) ⊃ ¬BOARD (q, b)));;

define BlackKingThm: ∀ r sq. (Val (r, Pos (r, sq)) =KB ≡ Pos (r, sq) =BK);;

define BlackMPCLemma: ∀ p b sq sqx. (((Valueon (b, sq) =PB ∧ BOARD (p, b)) ∧ Pos
(P0, sqx) =Pos (p, sq)) ⊃ MAY_PAWN_CAPTURES (sqx, sq, BLACK));;

define BlackPawnCaptureThm: ∀ p ybp sq1 sq2 sq3 b. ((Pos (P0, sq1) =ybp ∧ (Pos
(p, sq2) =ybp ∧ (MUST_PAWN_CAPTURES (sq1, sq2, Piececolor ybp) ∧ (BOARD (p, b) ∧
Valueon (b, sq2) =PB)))) ⊃ ((sq3=sq2 ∨ (SAMEDIAG (sq2, sq3) ∧ (SAMEDIAG (sq3,
sq1) ∧ BETWEEN (Row (sq2) , Row (sq3) , Row (sq1))))) ⊃ ∃ q3 x3. ((PREDEGAME (q3,
p) ∨ q3=p) ∧ ((TAKINGS Move q3 ∧ (Mover Move q3=ybp ∧ (To Move q3=sq3 ∧ Taken
Move q3=x3))) ∧ (PREDEGAME (Prevpos q3, p) ∧ (To Move q3=sq3 ⊃ (Mover Move q3=ybp
⊃ ((Taken Move q3=x3 ∧ ¬WHITEPIECE ybp) ⊃ (WHITEPIECE x3 ∧ (¬ (Row (sq3) =6) ⊃
Pos (Prevpos q3, sq3) =x3)))))))));;

define BlackPawnMoveThm: ∀ p q1 ybp sq. ((Pos (q1, sq) =ybp ∧ (¬ (Pos (P0, sq)
=ybp) ∧ ((PREDEGAME (q1, p) ∨ q1=p) ∧ Val (Prevpos q1, ybp) =PB))) ⊃ ∃ q.
((PREDEGAME (q, p) ∨ q=p) ∧ (Mover Move q=ybp ∧ (To Move q=sq ∧ (VALUEP Val
(Prevpos q, ybp) ∧ ((¬ (Row sq=6) ∧ ¬ (Row sq=4)) ⊃ ((From Move q=Makesquare
(Wsucf Row sq, Column sq) ∧ Pos (Prevpos q, sq) =EMPTY) ∨ ((Taken Move q=Pos
(Prevpos q, sq) ∧ WHITEPIECE Pos (Prevpos q, sq)) ∧ (From Move q=Makesquare
(Wsucf Row sq, L2touchf Column sq) ∨ From Move q=Makesquare (Wsucf Row sq,
R2touchf Column sq)))))))))));;

define BlackPawnPathThm: ∀ p x sq1 sq2. (Mover Move p=x ⊃ (To Move p=sq2 ⊃ (From
Move p=sq1 ⊃ (Val (Prevpos p, x) =PB ⊃ (¬ (Row (sq2) =6) ⊃ (ORDINARY Move p ∧
((Column (sq1) =Column (sq2) ∧ (BSUC (Row (sq1) , Row (sq2)) ∧ Valueon (Tboard
Prevpos p, sq2) =MT)) ∨ ((Column (sq1) =Column (sq2) ∧ (Row (sq1) =2 ∧ (Valueon
(Tboard Prevpos p, sq2) =MT ∧ (Valueon (Tboard Prevpos p, Makesquare (3, Column
(sq1))) =MT ∧ Row (sq2) =4)))) ∨ (TWOTOUCHING (Column (sq1) , Column (sq2)) ∧
(BSUC (Row (sq1) , Row (sq2)) ∧ WVALUES Valueon (Tboard Prevpos p,
sq2)))))))))));;

define BlackPawnValueLemma: ∀ p b sq. ((BOARD (p, b) ∧ Valueon (b, sq) =PB) ⊃
BPAWNS Pos (p, sq));;

define BlackPawnValueSquares: ∀ p b t sq. ¬ (PROMOTEDPAWN (p, t) ∧ (BOARD (p, b)
∧ (Valueon (b, sq) =PB ∧ Pos (p, sq) =t)));;

define BlackPawnsAre_: ∀ t. ((t=BKP ∨ t=BQP ∨ t=BKNP ∨ t=BKBP ∨ t=BKRP ∨ t=BQBP ∨
t=BQNP ∨ t=BQRP) ≡ BPAWNS (t));;

define BlackPawnsOn2Start_: ∀ sq. (BPAWNS Pos (P0 sq) ≡ Row (sq) =2);;

define BlackPromtesOn8A: ∀ q ybp. (PROMOTEDPAWN (q, ybp) ⊃ ∃ p. ((PAWNPROM Move p
∧ ((PREDEGAME (p, q) ∨ p=q) ∧ Mover Move p=ybp)) ∧ Row To Move p=8));;

define BlackPromtesOn8B: ∀ p ybp. ((PAWNPROM Move p ∧ Mover Move p=ybp) ⊃ Row To
Move p=8);;

define BlackValuesAre_: ∀ vb. (vb=KB ∨ (vb=QB ∨ (vb=RB ∨ (vb=NB ∨ (vb=BB ∨
vb=PB)))));;
```

```
define BlackpieceArePawnsOr_: ∀ t. (BLACKPIECE t ≡ (BPAWNS t ∨ t=BK ∨ t=BQ ∨
t=BKN ∨ t=BKB ∨ t=BKR ∨ t=BQB ∨ t=BQN ∨ t=BQR));;

define BlackpiecePawnsAre_: ∀ t. ((BLACKPIECE (t) ∧ PAWNS (t)) ≡ BPAWNS (t));;

define Blocked_BKB: ∀ r b sq. ((BOARD (r, b) ∧ (Valueon (b, BK2) =PB ∧ (Valueon
(b, BKN2) =PB ∧ Pos (r, sq) =BKB))) ⊃ sq=BKB1);;

define BlockedGivenThm: ∀ q. (BOARD (q, GIVEN) ⊃ ∀ sq1. (MOVETO (Tboard q, RW,
BQ2, sq1) ⊃ (¬ (Valueon (Tboard q, sq1) =MT) ∨ MOVETO (Tboard q, RW, BQN2,
sq1)))) ;;

define BoardTboard: ∀ r. BOARD (r, Tboard r);;

define BorW_Piece_: ∀ x. ¬ (BLACKPIECE x ≡ WHITEPIECE x);;

define BorW_Value_: ∀ vpc. ¬ (BVALUES vpc ≡ WVALUES vpc);;

define CAPPP_SortThm: ∀ m. ((PAWNPROM m ∧ CAPTURE m) ⊃ CAPPP m);;

define ChesspiecePieceValueThm: ∀ r x. PIECEVALUES Val (r, x);;

define ChsInd: (P P0 ∧ ∀ r p. ((P r ∧ SUCCESSOR (r, p)) ⊃ P p)) ⊃ ∀ r. P r;;

define ColorChoices: ∀ r t. ((BVALUES Val (r, t) ≡ BLACKPIECE t) ∧ (WVALUES Val
(r, t) ≡ WHITEPIECE t));;

define ColorTaken: ∀ p. (TAKINGS Move p ⊃ ((WHITETURN p ⊃ WHITEPIECE Taken Move
p) ∧ (¬WHITETURN p ⊃ BLACKPIECE Taken Move p)));;

define ColorsAre_: ∀ c. (c=BLACK ∨ c=WHITE);;

define Colours_: WHT (WHITE) ∧ BLK (BLACK) ∧ ¬WHT (BLACK) ∧ ¬BLK (WHITE) ∧
¬WHITE=BLACK;;

define DiagCommute: ∀ b sq1 sq2. (DIAG (b, sq1, sq2) ≡ DIAG (b, sq2, sq1))

define DiagonalThm: ∀ a b sq1 sq2. (SUBOARD (a, b) ⊃ (∀ sq3. ((SAMEDIAG (sq1,
sq3) ∧ (SAMEDIAG (sq2, sq3) ∧ BETWEEN (Row (sq1) , Row (sq3) , Row (sq2)))) ⊃ ¬
(Valueon (a, sq3) =UD)) ⊃ (DIAG (a, sq1, sq2) ≡ DIAG (b, sq1, sq2))));;

define DieOnce: ∀ q p x. (Taken Move p=x ⊃ (PREDEGAME (q, p) ⊃ ¬ (Taken Move
q=x)));;

define DifferentTaken: ∀ p1 p2 q x y. ((((p2=q ∨ PREDEGAME (p2, q)) ∧ (p1=q ∨
PREDEGAME (p1, q))) ∧ ((¬ (To Move p1=To Move p2) ∨ (¬ (Mover Move p1=Mover Move
p2) ∨ (PREDEGAME (p1, p2) ∨ ¬ (p1=p2)))) ∧ (Taken Move p1=x ∧ Taken Move p2=y)))
⊃ ¬ (x=y));;

define DifferentTakenFour: ∀ q p p1 p2 p3 p4 y x1 x2 x3 x4. (((p=q ∨ PREDEGAME
(p, q)) ∧ ((p1=q ∨ PREDEGAME (p1, q)) ∧ ((p2=q ∨ PREDEGAME (p2, q)) ∧ ((p3=q ∨
PREDEGAME (p3, q)) ∧ ((p4=q ∨ PREDEGAME (p4, q)) ∧ (¬ (Mover Move p1=Mover Move
p) ∧ (¬ (Mover Move p2=Mover Move p) ∧ (¬ (Mover Move p3=Mover Move p) ∧ (¬
(Mover Move p4=Mover Move p) ∧ (Taken Move p=y ∧ (Taken Move p1=x1 ∧ (Taken Move
```

```
p2=x2 ∧ (Taken Move p3=x3 ∧ Taken Move p4=x4))))))))))))) ⊃ (¬ (x1=y) ∧ (¬ (x2=y)
∧ (¬ (x3=y) ∧ ¬ (x4=y)))));;

define EmptyFrom: ∀ q x sq. (Pos (q, sq) =x ⊃ ¬ (sq=From Move q));;

define EmptyIsMT: ∀ r t. (t=EMPTY ≡ Val (r, t) =MT);;

define EquiOrthoThm: ∀ a b sq1 sq2. ((SUBOARD (a, b) ∧ (¬ (sq1=sq2) ∧ ((Column
(sq1) =Column (sq2) ∧ ∀ sq3. ((BETWEEN (Row (sq1) , Row (sq3) , Row (sq2)) ∧
Column (sq3) =Column (sq1)) ⊃ ¬ (Valueon (a, sq3) =UD))) ∨ (Row (sq1) =Row (sq2)
∧ ∀ sq3. ((BETWEEN (Column (sq1) , Column (sq3) , Column (sq2)) ∧ Row (sq3) =Row
(sq1)) ⊃ ¬ (Valueon (a, sq3) =UD)))))) ⊃ (ORTHO (a, sq1, sq2) ≡ ORTHO (b, sq1,
sq2)));;

define FarTaken: ∀ q. ((PAWNPROM Move q ∧ (Mover Move q=BKRP ∧ (¬ (To Move
q=WKR1) ∧ ¬ (To Move q=WKN1)))) ⊃ ∃ q1 q2 x1 x2. (((PREDEGAME (q1, q) ∨ q1=q) ∧
(PREDEGAME (q2, q1) ∧ (PREDEGAME (P0, q2) ∧ (TAKINGS Move q1 ∧ (TAKINGS Move q2 ∧
(Mover Move q1=BKRP ∧ (Mover Move q2=BKRP ∧ (Taken Move q1=x1 ∧ Taken Move
q2=x2)))))))) ∧ ¬ (x1=x2))) ;;

define GameRelations5: ∀ r. ¬PREDEGAME (r, P0);;

define GivenUD: ∀ sq. (Valueon (GIVEN sq) =UD ≡ sq=WKR4);;

define GivenWV: ∀ sq. (WVALUES Valueon (GIVEN sq) ⊃ (sq=BKR1 ∨ sq=BQ1 ∨ sq=BQ2 ∨
sq=WQR2 ∨ sq=WQN3 ∨ sq=WQB2 ∨ sq=WQ3 ∨ sq=WKB2 ∨ sq=WKN3 ∨ sq=WKR2));;

define GrandchildGenealogy: ∀ r q p. ((SUCCESSOR (r, q) ∧ PREDEGAME (q, p)) ⊃
PREDEGAME (r, p));;

define GrandparentGenealogy: ∀ q p. (PREDEGAME (q, p) ⊃ PREDEGAME (Prevpos q,
p));;

define GrandparentGenealogyX: ∀ q p p1. (((PREDEGAME (p, p1) ∨ p=p1) ∧ (PREDEGAME
(q, p) ∨ q=p)) ⊃ PREDEGAME (Prevpos q, p1));;

define GrandparentGenealogyY: ∀ r1 r p. ((SUCCESSOR (r, p) ∧ (PREDEGAME (r1, r) ∨
r1=r)) ⊃ PREDEGAME (r1, p));;

define KingCommute: ∀ sq1 sq2. (KINGMOVE (sq1, sq2) ≡ KINGMOVE (sq2, sq1))

define KingLemma: ∀ r p t. ((SUCCESSOR (r, p) ∧ VALUEK Val (p, t)) ⊃ Val (r, t)
=Val (p, t));;

define KingValueThm: ∀ r b sq. ((BOARD (r, b) ∧ ¬ (Valueon (b, sq) =UD)) ⊃ ((Pos
(r, sq) =WK ≡ Valueon (b, sq) =KW) ∧ (Pos (r, sq) =BK ≡ Valueon (b, sq) =KB)));;

define KingValuesAre_: ∀ v. (VALUEK v ≡ (v=KB ∨ v=KW));;

define KingsAre_: ∀ t. (KINGS t ≡ (t=BK ∨ t=WK));;

define KnightCommute: ∀ sq1 sq2. (KNIGHTMOVE (sq1, sq2) ≡ KNIGHTMOVE (sq2, sq1))

define MayMove: ∀ b v sq1 sq2. (MOVETO (b, v, sq1, sq2) ⊃ (Column (sq1) =Column
```

```
(sq2) ∨ (KNIGHTMOVE (sq1, sq2) ∨ (Row (sq1) =Row (sq2) ∨ (SAMEDIAG (sq1, sq2) ∨
(KINGMOVE (sq1, sq2) ∨ (TWOTOUCHING (Column (sq1) , Column (sq2)) ∧ (WSUC (Row
(sq1) , Row (sq2)) ∨ BSUC (Row (sq1) , Row (sq2)))))))))) ;;

define MconseqfX: ∀ r r1 sq x. (((r1=r ∨ PREDEGAME (r1, r)) ∧ Taken Move r1=x) ⊃
¬ (Pos (r, sq) =x));;

define MconseqhX: ∀ r q b sq sq1. ((Pos (q, sq) =P[illegible]r, sq1) ∧ ((SUCCESSOR (r,
q) ∧ (¬PAWNPROM Move q ∨ ¬ (Pos (q, sq) =Mover Mov[illegible])) ∧ (BOARD (q, b) ∧ ¬
(Valueon (b, sq) =UD)))) ⊃ Valueon (Tboard r, sq1) =Valueon (b, sq));;

define MconseqkX: ∀ r p. ((SUCCESSOR (r, p) ∧ ORDINARY Move p) ⊃ (¬ (From Move
p=To Move p) ∧ (MOVETO (Tboard r, Valueon (Tboard r, From Move p) , From Move p,
To Move p) ∧ ((SIMPLE Move p ⊃ Pos (r, To Move p) =EMPTY) ∧ ((CAPTURE Move p ⊃
(WHITEPIECE Taken Move p ≡ WHITETURN p)) ∧ ¬ (CAPTURE Move p ≡ SIMPLE Move
p))))));;

define MconseqlX: ∀ r q. (SUCCESSOR (r, q) ⊃ (PAWNPROM Move q ≡ (LASTRANKER (To
Move q, Color r) ∧ (SIMPLELEGALMOVE (r, q) ∧ (PAWNS Mover Move q ∧ (VALUEP
Valueon (Tboard r, From Move q) ∧ (((BVALUES Promoted Move q ≡ BVALUES Val (r,
Mover Move q)) ∧ (WVALUES Promoted Move q ≡ WVALUES Val (r, Mover Move q))) ∧ Val
(q, Mover Move q) =Promoted Move q)))))));;

define MconseqmX: ∀ r p. (SUCCESSOR (r, p) ⊃ ((CASTLE Move p ≡ CASTLING (r, p)) ∧
((ENPASSANT Move p ≡ EN_PASSANT (r, p)) ∧ (ORDINARY Move p ≡ SIMPLELEGALMOVE (r,
p)))));;

define MightBeBB: ∀ r t. (Val (r, t) =BB ⊃ ((t=BKB ∨ t=BQB) ∨ (BPAWNS t ∧
PROMOTEDPAWN (r, t))));;

define MightBeNB: ∀ r t. (Val (r, t) =NB ⊃ ((t=BKN ∨ t=BQN) ∨ (BPAWNS t ∧
PROMOTEDPAWN (r, t))));;

define MightBeRW: ∀ r t. (Val (r, t) =RW ⊃ ((t=WKR ∨ t=WQR) ∨ (WPAWNS t ∧
PROMOTEDPAWN (r, t))));;

define Mobility: ∀ r sq x. ((Pos (r, sq) =x ∧ ¬ (Pos (P0, sq) =x)) ⊃ ∃ q.
((PREDEGAME (q, r) ∨ q=r) ∧ ((Mover Move q=x ∧ To Move q=sq) ∨ (CASTLE Move q ∧
(Alsomover Move q=x ∧ Alsoto Move q=sq)))));;

define MoveChoices: ∀ p. (((CASTLE Move p ≡ CASTLING (Prevpos p, p)) ∧
((ENPASSANT Move p ≡ EN_PASSANT (Prevpos p, p)) ∧ (ORDINARY Move p ≡
SIMPLELEGALMOVE (Prevpos p, p)))) ∧ ((MOVES p ⊃ (ENPASSANT p ∨ (CASTLE p ∨
ORDINARY p))) ∧ ((MOVES p ⊃ ¬ (ENPASSANT p ∧ CASTLE p)) ∧ ((MOVES p ⊃ ¬
(ENPASSANT p ∧ ORDINARY p)) ∧ (MOVES p ⊃ ¬ (CASTLE p ∧ ORDINARY p)))))));;

define MovedValues: ∀ r p b sq sqx. (((SUCCESSOR (r, p) ∧ (¬EN_PASSANT (r, p) ∧
(¬CASTLING (r, p) ∧ (¬PAWNPROM Move p ∧ BOARD (p, b))))) ∧ (From Move p=sq ∧ To
Move p=sqx)) ⊃ (MOVETO (Tboard r, Val (p, Mover Move p) , sq, sqx) ∧ (¬ (Valueon
(b, sqx) =UD) ⊃ (MOVETO (Tboard r, Valueon (b, sqx) , sq, sqx) ∧ ((WHITETURN p ⊃
BVALUES Valueon (b, sqx)) ∧ (¬WHITETURN p ⊃ WVALUES Valueon (b, sqx)))))));;

define MovedValuesX: ∀ r p x sq sqx. (x=Mover Move p ⊃ ((¬PAWNS x ∧ ¬KINGS x) ⊃
((SUCCESSOR (r, p) ∧ (From Move p=sq ∧ To Move p=sqx)) ⊃ MOVETO (Tboard r, Val
(P0, x) , sq, sqx))));;
```

```
define MovementValues: V r p x sq sq1. ((SUCCESSOR (r, p) ∧ (x=Mover Move p ∧
(¬PAWNS x ∧ (sq=To Move p ∧ sq1=From Move p)))) ⊃ (MOVETO (Tboard r, Valueon
(Tboard r, From Move p) , From Move p, To Move p) = MOVETO (Tboard r, Val (P0, x)
, sq1, sq)));;

define MoverOnTO: V q y sq. ((Pos (q, sq) =y ∧ To Move q=sq) ⊃ y=Mover Move q);;

define MovetoCommute: V v b sq1 sq2. (¬VALUEP v ⊃ (MOVETO (b, v, sq1, sq2) =
MOVETO (b, v, sq2, sq1)));;

define NoBlackPawnsOn1Row: V r x sq. ((Val (r, x) =PB ∧ Pos (r, sq) =x) ⊃ ¬ (Row
(sq) =1));;

define NoEndInCheck: V r c. (¬ (c=Color r) ⊃ ¬POSITIONINCHECK (r, c));;

define NoPromtedInP0: V x. ¬PROMOTEDPAWN (P0, x);;

define NonmoverStays: V r q sq x. ((SUCCESSOR (r, q) ∧ (Pos (q, sq) =x ∧ (¬ROOKS
x ∧ ¬ (x=Mover Move q)))) ⊃ Pos (r, sq) =x);;

define NotBPFrom1Thm: V p b. ((¬CASTLING (Prevpos p, p) ∧ (BOARD (p, b) ∧
(¬EN_PASSANT (Prevpos p, p) ∧ (WHITETURN p ∧ Row From Move p=1)))) ⊃ ((¬ (Valueon
(b, To Move p) =UD) ⊃ MOVETO (Tboard Prevpos p, Valueon (b, To Move p) , From
Move p, To Move p)) ∧ (¬PAWNPROM Move p ∧ (MOVETO (Tboard Prevpos p, Val (Prevpos
p, Mover Move p) , From Move p, To Move p) ∧ (¬ (Valueon (b, To Move p) =UD) ⊃
(¬VALUEP Valueon (b, To Move p) ∧ BVALUES Valueon (b, To Move p)))))));;

define NotChesspieceEmpty_: V t. (¬CHESSPIECES t = t=EMPTY);;

define NotFromBKBBlocked: V r p sq. ((SUCCESSOR (r, p) ∧ (Mover Move p=BKB ∧ (Pos
(p, BK2) =BKP ∧ (Pos (p, BKN2) =BKNP ∧ Pos (p, sq) =BKB)))) ⊃ ¬ (From Move
p=BKB1));;

define NotMPC_Black2to1_: V dcx1 dcx2. ¬MAY_PAWN_CAPTURES (Makesquare (2 dcx1)
Makesquare (1 dcx2) BLACK);;

define NotPawnValuePromotedPawns: V r yp. (¬VALUEP Val (r, yp) ⊃ PROMOTEDPAWN (r,
yp));;

define OfficerValueThm: V r t. (¬PAWNS t ⊃ Val (P0, t) =Val (r, t));;

define OfficerValueThmX: V r t t1. ((¬PAWNS t ∧ t=t1) ⊃ Val (P0, t) =Val (r,
t1));;

define OnlyPawnsPromote: V r r1 t. ((¬VALUEP Val (r1, t) ∧ PREDEGAME (r1, r)) ⊃
Val (r, t) =Val (r1, t));;

define OrthoCommute: V b sq1 sq2. (ORTHO (b, sq1, sq2) = ORTHO (b, sq2, sq1))

define OrthoThmX: V q b sqx sq1. (BOARD (q, b) ⊃ (V sq3. (Valueon (b, sq3) =UD ⊃
(sq3=sqx v (¬ (Row (sq3) =Row (sqx)) ∧ ¬ (Column (sq3) =Column (sqx)))))) ⊃ (ORTHO
(Tboard q, sqx, sq1) = ORTHO (b, sqx, sq1))));;

define OtherSideStays: V r p sq x. ((SUCCESSOR (r, p) ∧ ((WHITEPIECE x =
WHITETURN p) ∧ Pos (p, sq) =x)) ⊃ Pos (r, sq) =x);;
```

```
define PXPawnTo: ∀ r p b. ((SUCCESSOR (r, p) ∧ (¬CASTLING (r, p) ∧ (¬EN_PASSANT
(r, p) ∧ (PAWNPROM Move p ∧ (¬WHITETURN p ∧ (From Move p=BQB2 ∧ BOARD (p,
b)))))))) ⊃ ((To Move p=BQN1 ∧ ((WVALUES Valueon (b, BQN1) ∨ Valueon (b, BQN1)
=UD) ∧ BVALUES Valueon (Tboard r, To Move p))) ∨ ((To Move p=BQ1 ∧ ((WVALUES
Valueon (b, BQ1) ∨ Valueon (b, BQ1) =UD) ∧ BVALUES Valueon (Tboard r, To Move
p))) ∨ (To Move p=BQB1 ∧ (WVALUES Valueon (b, BQB1) ∨ Valueon (b, BQB1) =UD)))))
;;

define ParentGenealogy: ∀ r2 r1 q. ((SUCCESSOR (r1, q) ∧ PREDEGAME (r2, q)) ⊃
(PREDEGAME (r2, r1) ∨ r2=r1));;

define PawnValuedBlackPieces: ∀ r yb. (VALUEP Val (r, yb) ⊃ Val (r, yb) =PB) ;;

define PawnValuedPawnsThm: ∀ r t. (VALUEP Val (r, t) ⊃ PAWNS t);;

define PawnValuesAre_: ∀ v. (VALUEP v ≡ (v=PB ∨ v=PW));;

define PawnWasOnThm: ∀ q p x sq. (((PREDEGAME (p, q) ∨ p=q) ∧ (VALUEP Val
(Prevpos p, x) ∧ (Mover Move p=x ∧ (From Move p=sq ∧ ¬ (Pos (P0, sq) =x))))) ⊃ ∃
p. ((Pos (p, sq) =x ∧ (PREDEGAME (p, q) ∧ VALUEP Val (p, x))) ∧ VALUEP Val
(Prevpos p, x)));;

define PieceChoices_: ∀ x. ((WHITEPIECE (x) ≡ (Piececolor (x) =WHITE)) ∧
(BLACKPIECE (x) ≡ (Piececolor (x) =BLACK)));;

define PiecevaluesAreChesspieces: ∀ r b sq. ((BOARD (r, b) ∧ PIECEVALUES Valueon
(b, sq)) ⊃ CHESSPIECES Pos (r, sq));;

define PiecevaluesAreChesspiecesX: ∀ r sq. (PIECEVALUES Valueon (Tboard r, sq) ⊃
CHESSPIECES Pos (r, sq));;

define PrevGameposition: ∀ p sq x. ((((WHITEPIECE x ≡ WHITETURN p) ∧ Pos (p, sq)
=x) ∧ ¬ (Pos (P0, sq) =x)) ⊃ ∃ q. Prevpos p=q) ;;

define PreviousPawnValue: ∀ r p t. (Prevpos p=r ⊃ (VALUEP Val (p, t) ⊃ VALUEP Val
(r, t)));;

define RetainValueColor: ∀ r1 r2 t. ((BVALUES Val (r2, t) ≡ BVALUES Val (r1, t))
∧ (WVALUES Val (r2, t) ≡ WVALUES Val (r1, t)));;

define RooksAre_: ∀ t. (ROOKS t ≡ (t=BKR ∨ t=WKR ∨ t=WQR ∨ t=BQR));;

define RowColumnSquareThm: ∀ sq1 sq2. (Row (sq1) =Row (sq2) ⊃ (Column (sq1)
=Column (sq2) ⊃ sq1=sq2));;

define SameColorsOnDiagonals_: ∀ sq1 sq2. (SAMEDIAG (sq1 sq2) ⊃ (WHITESQUARES
(sq1) ≡ WHITESQUARES (sq2)));;

define ShortPawnPathThm: ∀ r p sq1 sq2 x b. (∀ sq. ((MAY_PAWN_CAPTURES (sq2, sq,
Piececolor x) ∧ MAY_PAWN_CAPTURES (sq, sq1, Piececolor x)) ⊃ (sq=sq2 ∨ sq=sq1)) ⊃
((Pos (p, sq1) =x ∧ (Pos (P0, sq2) =x ∧ ((PREDEGAME (r, p) ∨ r=p) ∧ (VALUEP Val
(p, x) ∨ (BOARD (p, b) ∧ (Valueon (b, sq1) =PW ∨ Valueon (b, sq1) =PB)))))) ⊃
(Pos (r, sq1) =x ∨ Pos (r, sq2) =x)));;
```

```
define StandingBlackPawnThm: ∀ r p b sq. ((Valueon (b, sq) =PB ∧ (Row (sq) =2 ∧
(BOARD (p, b) ∧ SUCCESSOR (r, p)))) ⊃ (Valueon (Tboard r, sq) =PB ∧ Pos (r, sq)
=Pos (p, sq)));;

define SubBoards4X: ∀ r b sq. (BOARD (r, b) ⊃ (Valueon (b, sq) =Valueon (Tboard
r, sq) ∨ Valueon (b, sq) =UD));;

define SubboardTransitivity: ∀ b1 b2 b3. ((SUBOARD (b1, b2) ∧ SUBOARD (b2, b3)) ⊃
SUBOARD (b1, b3));;

define SubboardTransitivityX: ∀ a b r. ((SUBOARD (a, b) ∧ BOARD (r, b)) ⊃ BOARD
(r, a));;

define Substitution2: ∀ j1 j2 k1 k2. (j1=j2 ⊃ (k1=k2 ⊃ (β2 (j1 k1) =β2 (j2
k2))));;

define Substitution: ∀ j k. (j=k ⊃ β j=β k);;

define ThreeNB: ∀ r b x sq1 sq2 sq3 sqx. (∀ t. ((BPAWNS t ∧ PROMOTEDPAWN (r, t))
⊃ t=x) ⊃ (((¬ (sq1=sq2) ∧ (¬ (sq1=sq3) ∧ ¬ (sq2=sq3))) ∧ ((Val (r, Pos (r, sq1))
=NB ∨ (BOARD (r, b) ∧ Valueon (b, sq1) =NB)) ∧ ((Val (r, Pos (r, sq2)) =NB ∨
(BOARD (r, b) ∧ Valueon (b, sq2) =NB)) ∧ (Val (r, Pos (r, sq3)) =NB ∨ (BOARD (r,
b) ∧ Valueon (b, sq3) =NB))))) ⊃ (PROMOTEDPAWN (r, x) ∧ ((¬ (sq1=sqx) ∧ (¬
(sq2=sqx) ∧ ¬ (sq3=sqx))) ⊃ (¬ (Pos (r, sqx) =x) ∧ (¬ (Pos (r, sqx) =BKN) ∧ ¬
(Pos (r, sqx) =BQN)))))));;

define TransitiveGenealogy: ∀ r p q. ((PREDEGAME (r, p) ∧ PREDEGAME (p, q)) ⊃
PREDEGAME (r, q));;

define TransitiveSubboardMovement: ∀ a b v sq1 sq2. ((SUBOARD (a, b) ∧ MOVETO (a,
v, sq1, sq2)) ⊃ MOVETO (b, v, sq1, sq2));;

define TransitiveSubboardOrthogonality: ∀ a b sq1 sq2. (SUBOARD (a, b) ⊃ (ORTHO
(a, sq1, sq2) ⊃ ORTHO (b, sq1, sq2)));;

define TransitiveSuccession: ∀ r p x. ((PROMOTEDPAWN (r, x) ∧ SUCCESSOR (r, p)) ⊃
PROMOTEDPAWN (p, x));;

define TransitiveUNMKCAPPP: ∀ p a b sq1 sq2 v v1. ((BOARD (p, Unmkcapppmove (a,
sq1, sq2, v)) ∧ (SUBOARD (b, Unmkcapppmove (a, sq1, sq2, v1)) ∧ v=v1)) ⊃ BOARD
(p, b));;

define Trapped_QX_QB1_Thm: ∀ q v. (BOARD (q, QBUD) ⊃ ((v=RB ∨ v=QB) ⊃ ∀ sq1.
(MOVETO (Tboard q, v, BQ1, sq1) ⊃ (¬ (Valueon (Tboard q, sq1) =MT) ∨ MOVETO
(Tboard q, v, BKR1, sq1))))) ;;

define UDIsNotVW_: ∀ vw. ¬vw=UD;;

define Unique: ∀ r sq1 sq2 x. (Pos (r, sq1) =x ⊃ (Pos (r, sq2) =x ≡ sq1=sq2));;

define UnmovedBlackPawnThm: ∀ r b ybp sq. ((Pos (P0, sq) =ybp ∧ (Valueon (b, sq)
=PB ∧ BOARD (r, b))) ⊃ (Pos (P0, sq) =Pos (r, sq) ∧ Pospcf (r, ybp) =sq));;

define UnmovedWhitePawnThm: ∀ r b ywp sq. ((Pos (P0, sq) =ywp ∧ (Valueon (b, sq)
=PW ∧ BOARD (r, b))) ⊃ (Pos (P0, sq) =Pos (r, sq) ∧ Pospcf (r, ywp) =sq));;
```

```
define UnpromotedFrom: ∀ r q b x sq. ((SUCCESSOR (r, q) ∧ (¬WLASTRANK sq ∧ (BOARD
(q, b) ∧ (Valueon (b, sq) =vw ∧ (Pos (q, sq) =x ∧ Mover Move q=x))))) ⊃ Valueon
(Tboard r, From Move q) =vw) ;;

define ValueChoices_: ∀ vpc. ((WVALUES (vpc) ≡ Valuecolor (vpc) =WHITE) ∧
(BVALUES (vpc) ≡ Valuecolor (vpc) =BLACK));;

define ValueColorRetentionThm: ∀ r rl t. (PREDEGAME (rl, r) ⊃ ((BVALUES Val (rl,
t) ≡ BVALUES Val (r, t)) ∧ (WVALUES Val (rl, t) ≡ WVALUES Val (r, t))));;

define ValueFunctionChoices_: ∀ v. ((WVALUES (v) ⊃ Valuecolor (v) =WHITE) ∧
(BVALUES (v) ⊃ Valuecolor (v) =BLACK));;

define ValueTranspositionA: ∀ r t sq. (Pos (r, sq) =t ⊃ Valueon (Tboard r, sq)
=Val (r, t)) ;;

define ValueTranspositionB: ∀ r sq b. (BOARD (r, b) ⊃ (Valueon (b, sq) =Val (r,
Pos (r, sq)) ∨ Valueon (b, sq) =UD));;

define ValueTranspositionC: ∀ r sq. Valueon (Tboard r, sq) =Val (r, Pos (r,
sq));;

define WasAlwaysSomewhere: ∀ r rl sq x. ((PREDEGAME (rl, r) ∧ Pos (r, sq) =x) ⊃ ∃
sql. Pos (rl, sql) =x);;

define WasHere: ∀ r p sq x. ((SUCCESSOR (r, p) ∧ Pos (p, sq) =x) ⊃ ∃ sq. Pos (r,
sq) =x);;

define WasOn: ∀ p x. (Taken Move p=x ⊃ ∃ sq. Pos (Prevpos p, sq) =x);;

define WasPawnValue: ∀ rl r t. (((PREDEGAME (rl, r) ∧ VALUEP Val (r, t)) ∨ r=rl)
⊃ Val (r, t) =Val (rl, t));;

define WasPawnValueX: ∀ q p t. (((PREDEGAME (q, p) ∧ VALUEP Val (Prevpos p, t)) ∨
q=p) ⊃ Val (Prevpos p, t) =Val (Prevpos q, t));;

define WhereBishopTaken: ∀ q ybi sq sqx. ((To Move q=sq ∧ (Pos (P0, sqx) =ybi ∧ ¬
(WHITESQUARES sqx ≡ WHITESQUARES sq))) ⊃ ¬ (Taken Move q=ybi));;

define WhereOfficierTaken: ∀ q x sq. ((To Move q=sq ∧ (Taken Move q=x ∧ ¬PAWNS
x)) ⊃ Pos (Prevpos q, sq) =x);;

define WhereWhitePawns: ∀ p q x sq sq1 sq2 sq3 sq4 sq5 sq6 sq7 sq8. ((Pos (q,
sq1) =WQRP ∧ (Pos (q, sq2) =WQNP ∧ (Pos (q, sq3) =WQBP ∧ (Pos (q, sq4) =WQP ∧
(Pos (q, sq5) =WKP ∧ (Pos (q, sq6) =WKBP ∧ (Pos (q, sq7) =WKNP ∧ Pos (q, sq8)
=WKRP)))))))) ⊃ (((¬ (sq=sq1) ∧ (¬ (sq=sq2) ∧ (¬ (sq=sq3) ∧ (¬ (sq=sq4) ∧ (¬
(sq=sq5) ∧ (¬ (sq=sq6) ∧ (¬ (sq=sq7) ∧ ¬ (sq=sq8))))))))) ⊃ ¬WPAWNS Pos (q, sq)) ∧
((x=Taken Move p ∧ (PREDEGAME (p, q) ∨ p=q)) ⊃ ¬WPAWNS x)));;

define WhichBlackPawn: ∀ q b sq. ((BOARD (q, b) ∧ Valueon (b, sq) =PB) ⊃ ((Pos
(q, sq) =BQRP ∧ (Pospcf (q, BQRP) =sq ∧ MAY_PAWN_CAPTURES (BQR2, sq, BLACK))) ∨
((Pos (q, sq) =BQNP ∧ (Pospcf (q, BQNP) =sq ∧ MAY_PAWN_CAPTURES (BQN2, sq,
BLACK))) ∨ ((Pos (q, sq) =BQBP ∧ (Pospcf (q, BQBP) =sq ∧ MAY_PAWN_CAPTURES (BQB2,
sq, BLACK))) ∨ ((Pos (q, sq) =BQP ∧ (Pospcf (q, BQP) =sq ∧ MAY_PAWN_CAPTURES
```

```
(BQ2, sq, BLACK))) ∨ ((Pos (q, sq) =BKP ∧ (Pospcf (q, BKP) =sq ∧
MAY_PAWN_CAPTURES (BK2, sq, BLACK))) ∨ ((Pos (q, sq) =BKBP ∧ (Pospcf (q, BKBP)
=sq ∧ MAY_PAWN_CAPTURES (BKB2, sq, BLACK))) ∨ ((Pos (q, sq) =BKNP ∧ (Pospcf (q,
BKNP) =sq ∧ MAY_PAWN_CAPTURES (BKN2, sq, BLACK))) ∨ (Pos (q, sq) =BKRP ∧ (Pospcf
(q, BKRP) =sq ∧ MAY_PAWN_CAPTURES (BKR2, sq, BLACK)))))))))));;

define WhichWhitePawn: ∀ q b sq. ((BOARD (q, b) ∧ Valueon (b, sq) =PW) ⊃ ((Pos
(q, sq) =WQRP ∧ (Pospcf (q, WQRP) =sq ∧ MAY_PAWN_CAPTURES (WQR2, sq, WHITE))) ∨
((Pos (q, sq) =WQNP ∧ (Pospcf (q, WQNP) =sq ∧ MAY_PAWN_CAPTURES (WQN2, sq,
WHITE))) ∨ ((Pos (q, sq) =WQBP ∧ (Pospcf (q, WQBP) =sq ∧ MAY_PAWN_CAPTURES (WQB2,
sq, WHITE))) ∨ ((Pos (q, sq) =WQP ∧ (Pospcf (q, WQP) =sq ∧ MAY_PAWN_CAPTURES
(WQ2, sq, WHITE))) ∨ ((Pos (q, sq) =WKP ∧ (Pospcf (q, WKP) =sq ∧
MAY_PAWN_CAPTURES (WK2, sq, WHITE))) ∨ ((Pos (q, sq) =WKBP ∧ (Pospcf (q, WKBP)
=sq ∧ MAY_PAWN_CAPTURES (WKB2, sq, WHITE))) ∨ ((Pos (q, sq) =WKNP ∧ (Pospcf (q,
WKNP) =sq ∧ MAY_PAWN_CAPTURES (WKN2, sq, WHITE))) ∨ (Pos (q, sq) =WKRP ∧ (Pospcf
(q, WKRP) =sq ∧ MAY_PAWN_CAPTURES (WKR2, sq, WHITE)))))))))));;

define WhiteCapturedOnThm: ∀ r q y x sq. (Prevpos q=r ⊃ (To Move q=sq ⊃ (Mover
Move q=y ⊃ ((Taken Move q=x ∧ ¬WHITEPIECE y) ⊃ (WHITEPIECE x ∧ (¬ (Row (sq) =6) ⊃
Pos (r, sq) =x))))));;

define WhiteCapturedThm: ∀ p sq. (To Move p=sq ⊃ ((ORDINARY Move p ∧ WVALUES
Valueon (Tboard Prevpos p, sq)) ⊃ CAPTURE Move p));;

define WhiteCastleThm: ∀ r p sq. ((SUCCESSOR (r, p) ∧ (CASTLING (r, p) ∧
¬WHITETURN p)) ⊃ (Pos (p, sq) =WK ⊃ (sq=WKN1 ∨ sq=WQB1))) ;;

define WhiteCornered: ∀ r q b vw sq sqx. ((SUCCESSOR (r, q) ∧ (¬EN_PASSANT (r, q)
∧ (¬CASTLING (r, q) ∧ (¬WLASTRANK (sq) ∧ ((BOARD (q, b) ∧ (Valueon (b, sq) =vw ∧
(Valueon (b, sqx) =KB ∧ MOVETO (b, vw, sq, sqx)))) ∧ ¬VALUEP vw))))) ⊃ (∀ sq1.
(MOVETO (Tboard q, vw, sq, sq1) ⊃ (¬ (Valueon (Tboard q, sq1) =MT) ∨ MOVETO
(Tboard q, vw, sqx, sq1)))) ⊃ ((ORDINARY Move q ∧ SQUARE_BETWEEN (sq, From Move q,
sqx)) ∧ ¬ (Mover Move q=Pos (q, sq)))));;

define WhiteEnPassantThm1: ∀ r q. ((SUCCESSOR (r, q) ∧ (EN_PASSANT (r, q) ∧
¬WHITETURN q)) ⊃ (Valueon (Tboard q, To Move q) =PW ∧ Row (To Move q) =3));;

define WhiteEnPassantThm2: ∀ r q b. ((SUCCESSOR (r, q) ∧ (EN_PASSANT (r, q) ∧
¬WHITETURN q)) ⊃ (∀ dcx. ¬ (Valueon (b, Makesquare (3, dcx)) =PW ∨ Valueon (b,
Makesquare (3, dcx)) =UD) ⊃ ¬BOARD (q, b)));;

define WhiteKingThm: ∀ r sq. (Val (r, Pos (r, sq)) =KW ≡ Pos (r, sq) =WK);;

define WhiteMPCLemma: ∀ p b sq sqx. (((Valueon (b, sq) =PW ∧ BOARD (p, b)) ∧ Pos
(P0, sqx) =Pos (p, sq)) ⊃ MAY_PAWN_CAPTURES (sqx, sq, WHITE));;

define WhitePawnMovement: ∀ b b1 sqx sq1 sq2. (sqx=sq2 ⊃ (Valueon (b1, sq2) =PW ⊃
(MOVETO (b, Valueon (b1, sqx) , sq1, sq2) ≡ ((Column sq1=Column sq2 ∧ (WSUC (Row
sq1, Row sq2) ∧ Valueon (b, sq2) =MT)) ∨ ((Column sq1=Column sq2 ∧ (Row sq1=7 ∧
(Valueon (b, Makesquare (6, Column sq1)) =MT ∧ Row sq2=5))) ∨ (Valueon (b, sq2)
=MT ∧ (TWOTOUCHING (Column sq1, Column sq2) ∧ (WSUC (Row sq1, Row sq2) ∧ BVALUES
Valueon (b, sq2)))))))));;

define WhitePawnValueLemma: ∀ p b sq. ((BOARD (p, b) ∧ Valueon (b, sq) =PW) ⊃
WPAWNS Pos (p, sq));;
```

```
define WhitePawnsAre_: V t. ((t=WKP v t=WQP v t=WKNP v t=WKBP v t=WKRP v t=WQBP v
t=WQNP v t=WQRP) = WPAWNS (t));;

define WhitepieceAre_: V t. (WHITEPIECE t = ( t=WKP v t=WQP v t=WKNP v t=WKBP v
t=WKRP v t=WQBP v t=WQNP v t=WQRP v t=WK v t=WQ v t=WKN v t=WKB v t=WKR v t=WQB v
t=WQN v t=WQR));;

define WhitepieceArePawnsOr_: V t. (WHITEPIECE t = (WPAWNS t v t=WK v t=WQ v
t=WKN v t=WKB v t=WKR v t=WQB v t=WQN v t=WQR));;
```

Bibliography

Berliner74 Berliner, Hans J., Chess as Problem Solving: The Development of a Tactics Analyzer, (*dissertation*) Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1974.

Bobrow77 Bobrow, Daniel and Terry Winograd, "An Overview of KRL, A knowledge Representation Language", Cognative Science, Vol 1, No. 1., January 1977.

Brown73 Brown, J. S., R. R. Burton, and F. Zdybel, "A Model-Driven Question Answering System for Mixed Initiative Computer Assisted Instruction", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, May 1973.

Brown74 Brown, John Seely, Richard R. Burton, Alan G. Bell, and Robert J. Bobrow, SOPHIE: A sophisticated instructional environment, BBN Technical report, December 1974.

Bulnes79 Bulnes, Juan, GOAL: A Goal Oriented Command Language for Interactive Proof Construction, (*dissertation*), Department of Computer Science, Stanford University, *forthcoming* 1979.

Chomsky72 Chomsky, Noam, Language and Mind, New York, Harcourt Brace Jovanovich, 1972.

Dawson73 Dawson, T. R., Five Classics of Fairy Chess, New York, Dover Publications, 1973.

Filman76 Filman, Robert E., and Richard W. Weyhrauch, An FOL Primer, Stanford Artificial Intelligence Laboratory Memo 288, October 1976.

Funt77 Funt, Brian V., "Whisper: A problem Solving System Utilizing Diagrams and a Parallel Proessing Retina", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, August 1977.

Gardner59 Gardner, Martin, "Mathematical Games", Scientific American, Vol. 201, No. 5, May 1959.

Gardner73 Gardner, Martin, "Mathematical Games", Scientific American, Vol. 215, No. 5, May 1973.

Gelernter63a Gelernter, H., "Realization of a Geometry-Theorem Proving Machine", in Feigenbaum and Feldman (eds), Computers and Thought, New York, McGraw Hill, 1963.

Gelernter63b Gelernter, H., J. R. Hansen, and D. W. Loveland, "Empirical Explorations of the Geometry-Theorem Proving Machine", in Feigenbaum and Feldman (eds), Computers and Thought, New York, McGraw Hill, 1963.

Gizycki72 Gizycki, Jerzy, A History of Chess, translated by A. Wojciechowski, D. Ronowicz, and W. Bartoszewski, London, The Abbey Library, 1972.

Green69 Green, Claude Cordell, The Application of Theorem Proving to Question Answering Systems, Stanford Artificial Intelligence Laboratory Memo 96, June 1969.

Hayes77 Hayes, P. J., "In Defense of Logic", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, August 1977.

Hewitt71 Hewitt, Carl, PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot, (*dissertation*) Department of Electrical Engineering, Massachusetts Institute of Technology, February 1971.

Hewitt73 Hewitt, Carl, "A Universal Modular ACTOR Formalism for Artificial Intelligence", Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, August 1973.

Kling71 Kling, Robert E., Reasoning by Analogy with Applications to Heuristic Problem Solving: A Case Study, (*dissertation*) Computer Science Department, Stanford University, Stanford Artificial Intelligence Laboratory Memo 147, August 1971.

Kowalski76 Kowalski, Robert A., Algorithm = Logic + Control, Department of Computing and Control, Imperial College Research Report 77/3, November 1976.

McCarthy68 McCarthy, John, "Programs with Common Sense", in Minsky, Marvin (ed), Semantic Information Processing, Cambridge Massachusetts, MIT Press, 1968.

McCarthy69 McCarthy, John, and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in B. Meltzer and D. Michie (eds), Machine Intelligence 4, Edinburgh, Edinburgh University Press, 1969.

McCarthy78 McCarthy, John and Masahiko Sato, Takeshi Hayashi, Shigeru Igarashi,, On the Model Theory of Knowledge, Stanford Artificial Intelligence Laboratory Memo 312, April 1978.

McCarthy79a McCarthy, John, "First Order Theories of Individual Concepts and Propositions", in D. Michie (ed), Machine Intelligence 9, *forthcoming* 1979.

McCarthy79b McCarthy, John, "Circumscription Induction - A Way of Jumping to Conclusions", *submitted to* Artificial Intelligence 1979.

Minsky68 Minsky, Marvin, "Introduction", in Minsky, Marvin (ed), Semantic Information Processing, Cambridge Massachusetts, MIT Press, 1968.

Moore75 Moore, Robert Carter, Reasoning from Incomplete Knowledge in a Procedural Deduction System, (*Master's thesis*), MIT-AI-TR 347, December 1975.

Moore77 Moore, Robert Carter, "Reasoning about Knowledge and Action", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, August 1977.

Moore79 Moore, Robert Carter, Reasoning about Knowledge and Action, (*dissertation*) Department of Electrical Engineering, Massachusetts Institute of Technology, *forthcoming* 1979.

Newell72 Newell, Alan, and Herbert A. Simon, Human Problem Solving, Englewood-Cliffs, New Jersey, Prentice-Hall, 1972.

Nilsson71 Nilsson, Nils J., Problem-Solving Methods in Artificial Inteligence, New York, McGraw-Hill, 1971.

Pratt77 Pratt, Vaughn, The Competance/Performance Dichotomy in Programming, MIT Artificial Intelligence Laboratory Memo 400, January 1977.

Prawitz65 Prawitz, Dag, Natural Deduction - A Proof-Theoretical Study, Stockholm, Almquist & Wiksell, 1965.

Prior57 Prior, Arthur N., Time and Modality, Oxford, Clarendon Press, 1957.

Prior68 Prior, Arthur N., Papers on Time and Tense, Oxford, Clarendon Press, 1968.

Rieger76 Rieger, Chuck and Milt Grinberg, The Causal Representation and Simulation of Physical Mechanisms, University of Maryland Computer Science Technical Report TR-495, November 1976.

Rieger77 Rieger, Chuck and Milt Grinberg, The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms, University of Maryland Computer Science Technical Report TR-513, March 1977.

Reiter72 Reiter, Raymond, The Use of Models in Automatic Theorem Proving, Department of Computer Science, University of British Columbia technical report, 1972.

Sacerdoti73 Sacerdoti, Earl D., "Planning in a Hierarchy of Abstraction Spaces", Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, August 1973.

Sussman75 Sussman, Gerald Jay, and Richard Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis", IEEE Transactions on Circuits and Systems, Vol CAS-22 No. 11, November 1975.

Thomas78 Thomas, Arthur J., Representation and Conception: An Essay in Computational Metaphysics, (*dissertation*) Special Graduate Program, Stanford University, 1978.

Weyhrauch77 Weyhrauch, Richard W., A Users Manual for FOL, Stanford Artificial Intelligence Laboratory Memo 235.1, July 1977.

Weyhrauch78 Weyhrauch, Richard W., Prolegomena to a Theory of Mechanized Formal Reasoning, Stanford Artificial Intelligence Laboratory Memo 315, December 1978.

Whorf56 Whorf, Benjamin Lee, Language, Thought, and Reality - Selected writings, John B. Carroll (ed), Cambridge, Massachusetts, MIT Press, 1956.

Winograd72 Winograd, Terry, Understanding Natural Language, New York, Academic Press, 1972.

Winograd75 Winograd, Terry, "Frame Representations", in Daniel G. Bobrow and Allan Collins (eds), Representation and Understanding -- Studies in Cognative Science, New York, Academic Press, 1975.

Index to Predconst and Opconst Declarations

Index to Axioms and Theorems