OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

# Improving Problem Solving Performance by Example Guided Reformulation of Knowledge

Nicholas S. Flann
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

88-30-04

# Improving Problem Solving Performance by Example Guided Reformulation of Knowledge

Nicholas S. Flann
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
flann@cs.orst.edu

## Abstract

This paper introduces a method that improves the performance of a problem solver by reformulating its domain theory into one in which functionally *relevant* features are explicit in the syntax. This method, in contrast to previous reformulation methods, employs sets of training examples to constrain and direct the reformulation process. The use of examples offers two advantages over purely deductive approaches: First, the examples identify the exact part of the domain theory to be reformulated. Second, a proof with examples is much simpler to construct than a general proof because it is fully instantiated. The method exploits the fact that what is *relevant* to a goal is *syntactically explicit* in successful solutions to that goal. The method first takes as input a set of training examples that "exercise" an important part of the domain theory and then applies the problem solver to explain the examples in terms of a relevant goal. Next, the set of explanations is "clustered" into cases and then generalized using the induction over explanations method, forming a set of general explanations. Finally, these general explanations are reformulated into new domain theory rules. We illustrate the method in the domain of chess. We reformulate a simple declarative encoding of *legal-move* to produce a new domain theory that can generate the legal moves in a tenth of the time required by the original theory. We also show how the reformulated theory can more efficiently describe the important *knight-fork* feature.

# 1 Introduction

One solution to the important problem of constructing a usable knowledge base for a knowledge based system is to employ systems that automatically reformulate a given inefficient knowledge base into an efficient one. These systems are initially given an epistemologically adequate knowledge base and through experience or analysis, construct a more effective 'expert' knowledge base. The main advantage these systems have over the traditional knowledge engineering approach is that the initial knowledge that must be supplied to the system is much easier to formalize and encode in a computer.

Two principle methods have been proposed and applied with limited success to this problem: Explanation based learning and problem reformulation.

Explanation based learning (EBL) is a method by which a system improves its performance though analyzing successful (and failed) solutions. Given an example problem that is solved by the system, the solution trace is analysed and generalized to form a rule that will solve the same and similar problems faster the next time. (see Mitchell, Keller & Kedar-Cabelli, (1986) for a complete description of the method). EBL has been successful in some small domains but there are serious limitations. First, it is often the case that learning can diminish performance rather than improve it (Minton, 1985). Second, an EBL system does not benefit much from training examples because it is very inflexible in what it learns from each example (Flann & Dietterich, 1988). Third, the rules learned that extend the knowledge base are simply *syllogisms* of existing rules. This means that the systems do not go beyond the initial vocabulary used in problem solving. They simply cache sequences of implications that exist in the initial knowledge base. Because of these limitations, the method does not supply a general solution to the problem above.

Problem reformulation methods overcome some of the problems of EBL because more powerful changes are made in the initial knowledge base than simple syllogisms. These methods aim to transform the representation of a problem into one in which the solution is more easily found, often generating a new vocabulary. However, there are also problems associated with this method. First, the techniques have been successful only with small toy problems such as Missionaries and Cannibals, and Towers of Hanoi (Korf 1980, Amarel 1982). Second, logic based methods, such as those presented in Subramanian & Genesereth (1987), required computationally expensive first order proofs. Third, although many useful transformations have been identified, there is little understanding of how to control the application of these transformations.

Neither of these methods supply a solution to the general problem of automatically transforming a knowledgeable novice to an expert. EBL methods lack powerful transformations while reformulation techniques lack guidance on when to apply transformations.

In order to understand the difference between a knowledgeable novice and an expert, and identify the kinds of change we are interested in achieving automatically, let us consider a simple chess problem. Figure 1 shows a typical mid-game position with black to play.

A novice's knowledge of chess is comprised of the rules of the game and the ability to recognize a win or loss. When such a novice is faced with this position, she will perform a limited search analyzing a few alternatives and come up with a move such as moving the knight on `e4` to `c5`. This is not the best move in this position, in fact there is a way the black side can take the white queen. However, this sequence of moves is 6 ply deep and therefore cannot be seen by a novice.

The queen is captured by the following sequence of moves: first the black bishop on `f2` is moved to `c5`, white moves the king out of check to `f7`, black now moves the knight to `d6` checking the king. The only alternative is for white to move the king out of danger allowing black to capture the queen with the knight.

An expert will see this solution. First the expert may notice that the knight on `e4` can both

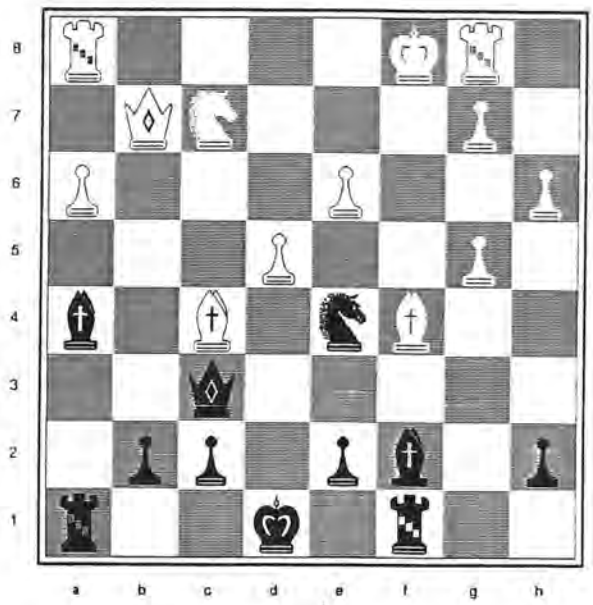


Figure 1: Example of difficult chess problem, black to play

directly threaten the queen and potentially threaten the king if it moved to `f7`. The expert now looks for ways to force the king into `f7` where it can be forked with the queen. The expert knows that one way to force a king to move is to put it in check and identifies the check threat by the queen from `c3` to `a3` as a suitable candidate. However, this does not complete the solution because a precondition of a fork is that the king cannot avoid moving, and in this case there is an option for the opponent to take the knight on `d6` with the bishop on `f4`. A subgoal is now created to find a move that can prevent the bishop from taking the knight and simultaneously check the king. The expert knows that there are three ways to prevent a move: take the piece, block the move or pin the piece. The bishop's move from `f2` to `c5` that creates a pin is found and verified as a suitable checking move. The solution has been found.

There are some important observations to make concerning the expert players problem solving compared with that of the novice. The most important factor to note is that the expert is employing a vocabulary of *functional* features such as *incheck-by-knight*, *remove-check-by-capture*, *remove-threat-by-block*, *prevent-move*, *threaten-piece*, *knight-fork* and *pin-piece*. These features play a critical role in the problem solving in two ways:

- The vocabulary of functional features acts as a strong source of focus for the search. For example, once the potential fork is identified, two sub-goals are created and pursued, one getting the king to the "forkable" square (`f7`), the other freeing the forking square (`d6`) from the threat of capture.

- The functional features define a smaller and more pertinent search space for the expert to search. These features hide many of the irrelevant details such as the positions of other pieces that do not play a role in the current goals. For example, when the knight move was identified that would fork the queen and king, each individual state resulting from the possible king moves were not considered. Rather, all such states were treated as a single functionally defined state, one in which the king moves "out of check." The structural distinction of the destination square of the king was ignored because it is irrelevant to achieving the current goal.

This brief comparison between a novice and a chess expert identifies a way of improving problem solving performance that we will refer to as *problem-solver reformulation.*

This paper introduces a method to achieve problem-solver reformulation. In particular, the problem we are interested in solving is:

Given:

- An epistemologically adequate domain theory[1].
- A simple problem solver that can apply the domain theory in a search intensive way to achieve its goals.

Find:

- A reformulation of the domain theory cast in terms of new functional features.
- A problem solver that can apply these functional features to focus and reduce its search to achieved its goals more effectively.

In this paper we address the first component of the solution: reformulating the domain theory. The second component is an area of current research.

The remainder of this paper is organized as follows: Section 2 presents our approach to this problem. Section 3 details our method with an example from chess. Section 4 presents some empirical results that demonstrate improved performance through reformulation. We conclude in Section 5 where we compare the method with other approaches and suggest a reason for its success.

## 2 Approach

We view the goal of reformulation as the process of making functionally relevant knowledge explicit and directly usable by the problem solver.

Hasse (1986), and Lenat and Brown (1984) view such reformulation as collapsing the semantics into the syntax. Distinctions that were only apparent through extensive search become explicit in the syntax of the vocabulary of the problem solver. Consider a simple example from chess. When the king is in check by a knight, there are only two options available, either the king must be moved or the knight taken. This constraint is buried in the initial semantics of the chess domain theory; each time the situation arises, the initial problem solver, after extensive computation, will always identify moves that fall into one of the two cases. By reformulating the domain theory, these two options can be made explicit in the syntax.

Such a reformulated domain theory can considerably improve the problem solving performance. First, when the king is recognized as being in check from a knight (i.e., recognize the functional feature *incheck-by-knight*), there is no wasted work pursuing illegal moves. Second, and of more importance, the explicit options (i.e., *move-out-of-check* and *remove-check-by-capture*) can be used as functional features and direct the search. In the example problem above (Figure 1), the expert recognized the potential *incheck-by-knight* threat and it then explored the two options *move-out-of-check* and *remove-check-by-capture*. Seeing that if white chooses *move-out-of-check* the queen will be captured, the problem solver focuses the search on thwarting a *remove-check-by-capture* by the opponent.

With such potential benefit coming from reformulation it is surprising that it has had so little application in machine learning. One reason for the absence of success with reformulation techniques is that they are currently very unconstrained and computationally intensive. For example,

[1]We will use domain theory and knowledge base as synonyms

in Subramanian and Genesereth (1987) a logic of irrelevance is presented that can identify irrelevances in a domain theory that suggest reformulations. However, this process involves constructing complex proofs in a first order language—a semi-decidable problem. Other techniques such as those presented by Amarel (1982) and Korf (1980) are equally costly since they require extensive search.

In this proposal we present a new approach to this problem that overcomes the computational complexity:

> *In our approach, we employ a carefully chosen set of training examples supplied by a teacher to constrain and direct the reformulation process.*

Examples offer two principle advantages. First, the examples identify the exact part of the domain theory to be reformulated. Second, the proof with examples is much simpler to construct than the general proof (i.e., one using variables) because it is fully instantiated.

The reformulation method employs a sequence of training instance sets (called lessons), each more complex than the last. In this way the method learns *incrementally*—it applies previously learned features to simplify the current learning task. For example, in chess the first lessons concern enumerating the important cases of legal moves, such as moving when in check by a knight, or moving when in check by a bishop, queen or rook. In this later case, the moving player has the potential to *block* the check threat and thus can learn the functional feature *remove-check-by-blocking*. The follow up lessons will include more advanced features such as *pins*, *forks* and *skewers*.

Through this technique it is intended that a knowledgeable chess novice that cannot solve the initial problem given in Figure 1 under some resource bound (both time and space), can through instruction, come to solve the problem under the same resource bound.

## 3 Reformulation Method

In this section we detail the reformulation method and demonstrate through an extended example how some of the functional features introduced earlier are learned.

Let us assume the initial domain theory is written in Prolog and consists of rules and facts that describe the rules and goals of the problem. More formally, the domain theory $DT$, consists of a set of rules of the form $H_i$:-$P_1, \ldots, P_n$, where each $P_l$ $(1 \leq l \leq n)$ either unifies with some $H_j$ or some fact $F$. We call the set of $P_l$ that exclusively unify with facts, primitive predicates.

We are now in a position to define the inputs and outputs of the reformulation method:

**Given:**

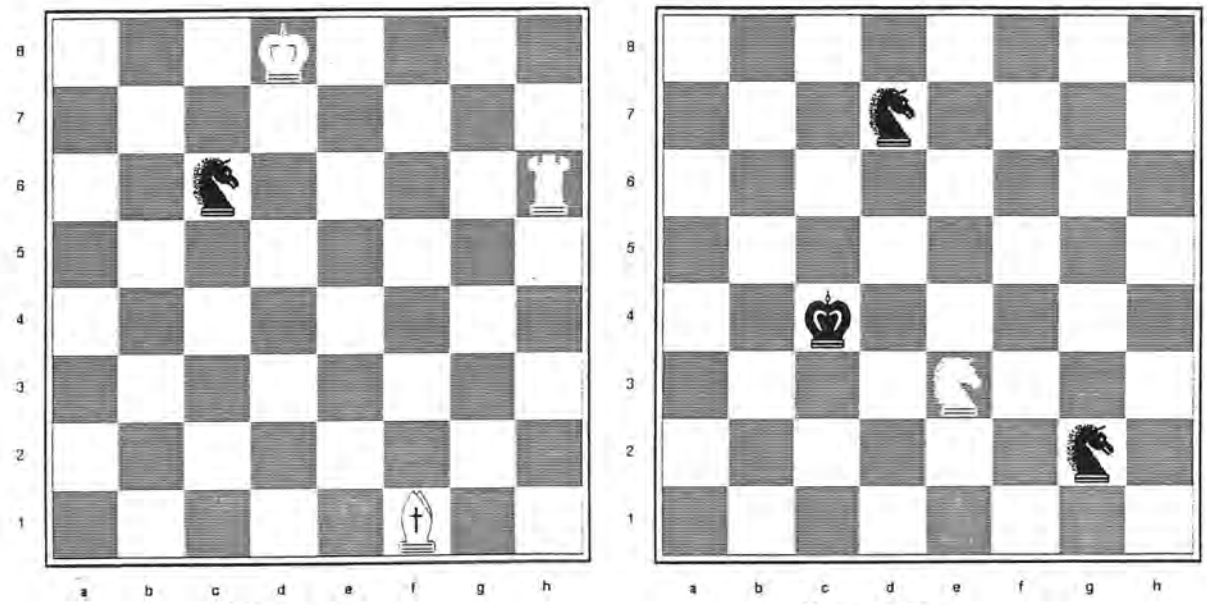
- A domain theory $DT_{old}$ that includes a single rule, $H_1$:-$P_1, \ldots, P_n$.
- A set of training examples described in terms of the primitive predicates that satisfy $H_1$.

**Find:**

- A reformulated domain theory $DT_{new}$ that includes a new rule of the form

$$H_1\text{:} -Pc\prime, !, (P\prime_1 \mid P\prime_2 \mid \ldots \mid P\prime_m),$$

  where $Pc\prime$ and $P\prime_j$ $(1 \leq j \leq m)$ are new predicates, $P$, defined as follows: $P$:-$Pn_1, Pn_2, \ldots, Pn_r$, where each $Pn_i$ $(1 \leq i \leq r)$ is either a $P\prime_j$, a $H_k$, or a primitive predicate.



(a) state1, White to play

(b) state2, Black to play

Figure 2: Lesson to reformulate `legalmove`

The new rule defining $H_1$ describes a special case of the original rule since it only applies when $Pc\prime$ is true. The predicates in the body of the new rule, $Pl_j$, explicitly enumerate disjunctive cases that hold for the original body $P_1, \ldots, P_n$ but are not explicit.

To clarify these definitions we give the inputs and outputs of the method when reformulating the legal move rule in chess.

The original rule that defines the legal moves (i.e., $H_1$) is given below:

```
legalmove(State,Newstate,Side):-
    possiblemove(State,Newstate,Side),
    not(incheck(Newstate,Side)).
```

A legal move in `State` for `Side` is one that is possible and does not lead to `Side` being in check. The `possiblemove` rule generates `Newstates` that result from possibly legal moves for `Side`. The `incheck` rule succeeds when there exists a possible move for the opponent that could take the king of `Side`. We include the definition of these and other goals in the appendix.

The lesson set in this case is two board positions (illustrated in Figure 2) both of which cover an important special case of generating legal moves—when the king is in check from a knight. In the white-to-play position, the king on `d8` is in check from the knight on `c6`. There are two legal classes of moves white can make in this position, either move the king or take the knight with the rook on `h6`.

The reformulated rule generated by the method from this lesson set is given below:

```
legalmove(State,Newstate,Side):-
    incheck-1(State,Side1,Side2,KnSq,KnPl,KSq,KPl), !,
    (legalmove-1(State,Newstate,Side1,Side2,KnSq,KnPl,KSq,KPl)
     | legalmove-2(State,Newstate,Side1,Side2,KnSq,KnPl,KSq,KPl)).
```

$Pc\prime$ in this case is `incheck-1(State,Side1,Side2,KnSq,KnPl,KSq,KPl)`, which is true when `Side1` is in check from a knight on square `KnSq`[2]. There are two cases; $Pl_1$ is `legalmove-1(...)`

[2]We will define the complete predicate later.

that generates moves by the king, while $Pl_2$ is `legalmove-2(...)` that generates moves that capture the knight on `KnSq`. We define `legalmove-1(...)` and `legalmove-2(...)` by additional rules given below.

This rule says that if in check by the knight then there are two options; either move the king out of check or capture the knight checking the king. Note the new predicates generated, `incheck-1`, `legalmove-1` and `legalmove-2` explicitly define the functional features *incheck-by-knight*, *move-out-of-check* and *remove-check-by-capture* introduced earlier.

Now we have given the inputs and outputs of the method, we describe the method in detail continuing with the `legalmove` example. The method has three stages: generate explanations, identify new definitions and finally, extract new domain theory rules.

### 3.1 Generating Explanations

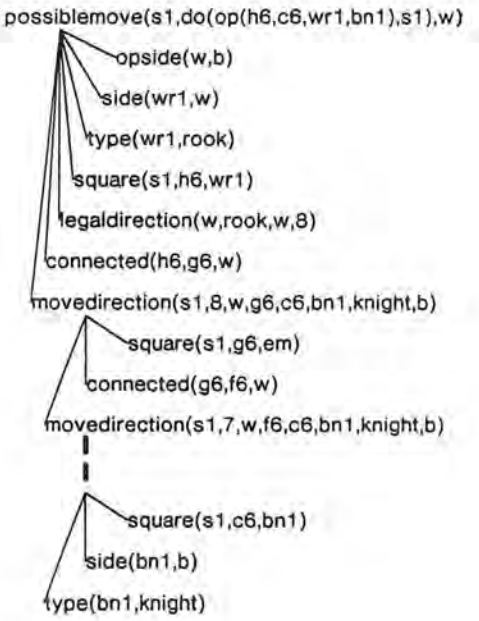
The first stage of reformulation is to apply the domain theory to analyze the training examples. The goal of this stage is to determine how these examples satisfy the current domain theory. Here, the rule for `legalmove` is used to find the set of legal moves for each example. During analysis a cache of the computation involved in generating the moves is made. This cache forms a set of proofs or explanations that demonstrate each legal move is indeed legal. Note that in each example, some possible moves (such as moving the white bishop in Figure 2) turn out not to be legal moves because the king is still in check following the move. Even though this analysis does not result in any legal moves, it is included in the set of explanations. We include fragments of four such explanations in Figure 3: two legal move fragments (1a) and (1b), and two illegal move fragments (2a) and (2b) (see appendix for description of the primitive predicates). In (1a) we show the `possiblemove` proof of the rook move from `h6` that captures the knight on `c6` in `state1`. In (2b) we show a failed not-incheck proof that proves the black move from `d7` to `b6` in `state2` is illegal because the king is still in check from the knight on `e3`.

### 3.2 Identify New Definitions

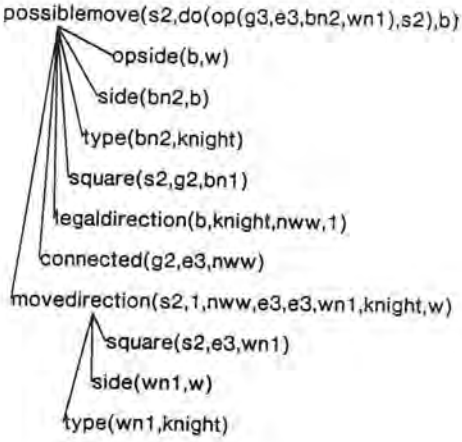
In the second stage, the set of explanations are syntactically compared and generalized. The goal of this stage is to identify fragments of the explanations that define the intended functional features or components of the reformulated rule (such as $Pc\prime$). Here, we identify three generalized explanations that define the functional features *incheck-by-knight*, *move-out-of-check* and *remove-check-by-capture*.

First, we identify the condition $Pc\prime$ by empirically determining a reason for the failed explanations. To do this we compare and generalize the successful explanations, producing a general explanation that describes moves that do not result in check. We similarly compare and generalize the failed explanations, this time producing a general explanation that describes illegal moves that result in check by the knight. To identify the condition, we compare the two general explanations and search for a syntactic difference that would account for the failure. The explanation fragment that defines the check by the knight is identified and proposed as the condition.

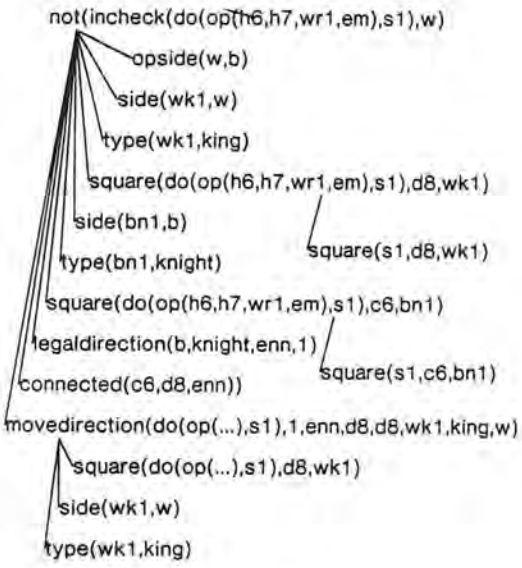
We use the induction-over-explanations (IOE) method (described in Flann & Dietterich, 1988, 1986; Dietterich & Flann, 1988) to generalize among the explanations. IOE syntactically generalizes a set of explanations and forms a single generalized explanation that represents the maximally specific common generalization of the input explanations. The generalized proof is formed by a combination of a simple constants to variable bias that is employed over the syntactic structure of the explanations and the pruning of dissimilar explanation sub-trees among the instances. IOE is used in preference to the more familiar EBG generalization method because EBG is too aggressive



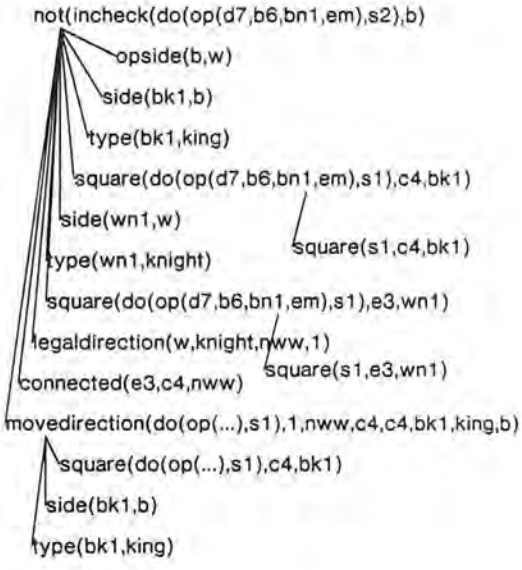

**(1a) Successful move white-to-play**




**(1b) Successful move black-to-play**




**(2a) Failed move white-to-play**




**(2b) Failed move black-to-play**

Figure 3: Explanations for `possiblemove` and `incheck`

in its generalization policy and ignores important commonalities among the examples. IOE, because it uses a more conservative generalization policy, is able to retain such commonalities and thereby extract more information from the training examples (Flann & Dietterich, 1988).

We illustrate this in Figure 4(2c), where we give the result of generalizing the `incheck` fragment of the failed explanations. Note that most of the constants existing in the input explanations (in Figure 3(2a) and (2b)) have been replaced by variables. However, the explanation is not completely variablized and retains the constraint that the checking piece is a `knight` (in `type(KnPl,knight)`). In other words, this generalized explanation defines the feature *incheck-by-knight*. We retained the important `knight` constraint because it was common among all the instances. An EBG generalization policy would have simply variablized everything and thus lost this constraint.

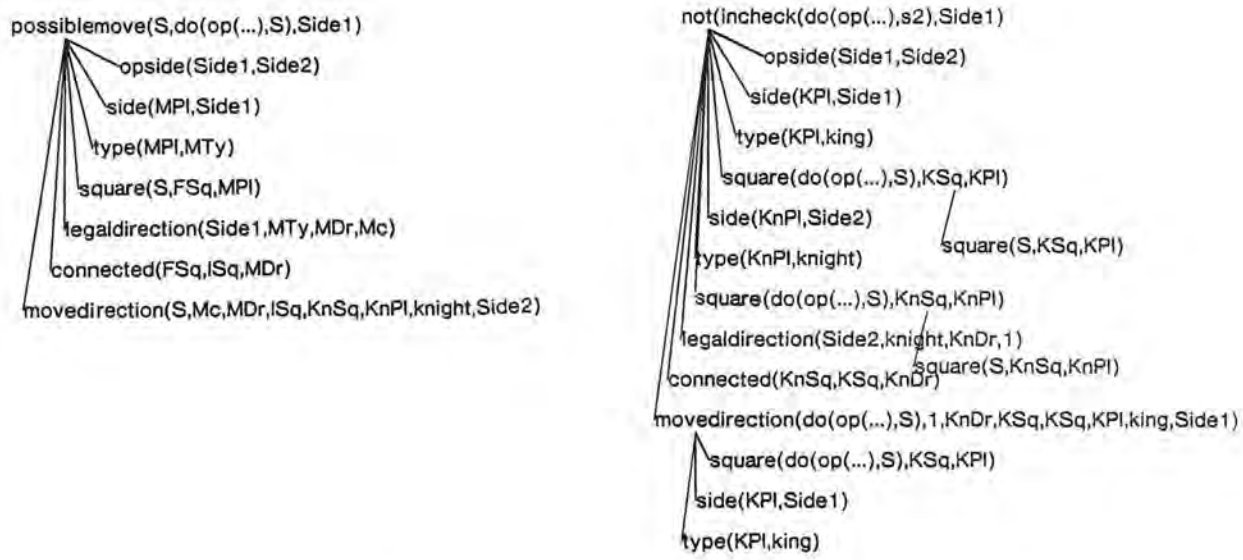
The explanation in Figure 4(2c) is formed by recursively descending the explanation trees starting at the root, merging the explanations to form a single general explanation. This merging step takes a list of ground predicates $[p(x1_1, x2_1, \ldots, xn_1), \ldots, p(x1_m, x2_m, \ldots, xn_m)]$ and returns a generalized predicate $p(x1, x2, \ldots, xn)$, where $xj$ equals $xj_1$ if $xj_1 = xj_2 = \ldots = xj_m$ and equals a variable if any two $xj_i \neq xj_k$. For example, when generalizing Figure 3(2a) and (2b), the list of predicates `[type(bn1,knight), type(wn1,knight)...]` will be merged resulting in `type(KnPl,knight)` being included in the general explanation (upper case letters are variables).

Note that the explanation includes many repeated variables. For example, the variable `KPl` that represents the king, occurs both in the `square(S,KSq,KPl)` predicate and `type(KPl,king)` predicate. These repeated variables are important because they encode the shared variable constraints present in the domain theory rules. Without them, the generalized explanation would no longer define a check, because for example, the piece potentially captured would no longer have to be a king (i.e., the `KPl` in `type(KPl,king)` and `square(S,KSq,KPl)` would be different). We preserve the shared variable constraints from the domain theory by what we call the "no coincidences" bias: if ever the same set of constants is merged we assign them the same variable. When generalizing `[type(bk1,king), type(wk1,king)...]` we merge {`bk1`,`wk1`} and when generalizing `[square(state1,c4,bk1), square(state2,d8,wk1)...]` we also merge {`bk1`,`wk1`}. Because these sets are the same, we assign the same variable `KPl`.

We have described how the condition part of the new reformulation is identified. We now turn our attention to the identification of the other functional features and components of the reformulation.

The goal of this stage is to identify distinct cases that make up the possible solutions to the rule being reformulated. For the legal move rule we wish to identify the two cases: either move the king out of check or take the checking piece. This is accomplished by first applying clustering techniques (such as those described by Fisher, 1987) over the successful explanations. The two sets identified in our example are the set of all moves by the king and the set of all moves that take the knight. We then apply IOE to form two generalized explanations.

We illustrate the generalized explanation for the knight capture moves in Figure 4(1c). For this explanation to correctly describe the *remove-check-by-capture* feature, the piece captured must be the same piece that is checking the king. This is an important *semantic* constraint, for if it were not the case, the move would be illegal. We correctly find this constraint because it is *explicit* as a repreated pattern in the *syntax* of the successful explanations. We identify this pattern though the use of the "no coincidences" bias introduced earlier. The merging of the destination squares of the capturing moves (found in the `movedirection` predicate in Figures 3(1a) and (1b)) produces the same set of constants {`c6`,`e3`}, as the merging of the originating square of the check threat (found in the `square` predicate in Figures 3(2a) and (2b)). Since both sets are equal, we assign them the same variable `KnSq`.




**(1c)** *remove-check-by-capture* **generalized explanation**

**(2c)** *in-check-by-knight* **generalized explanation**

Figure 4: Generalized Explanations for *in-check-by-knight* and *remove-check-by-capture*

This stage has identified three generalized explanations that describe the three functional features *incheck-by-knight*, *remove-check-by-capture* and *move-out-of-check*. The final stage describes how the reformulated domain theory rules are extracted out of these explanations.

### 3.3 Extracting New Domain Theory Rules

Extracting the new domain theory rules involves three parts: First, we find variables that are shared among different parts of the explanations and "promote" these variables to the renamed heads of the rules used in the explanations. Second, we walk down the explanations extracting new domain theory rules for each head in the explanations. Finally, we simplify the code by removing redundant predicates and reorder tests for increased efficiency (using techniques introduced by Smith, Genesereth & Ginsberg, 1986).

We generate the condition rule (*incheck-by-knight*) from the explanation fragment in Figure 4(2c) by re-naming the head `incheck-1` and promoting the shared variables `Side2, KnSq, KnPl, KSq, KPl` as new arguments. The final rule is generated by collecting the leaves (i.e., the primitive predicates) and simplifying. The rule defines `incheck-1` as true in state `S` when there exists a king of side `Side1` on square `KSq`, a knight of side `Side2` on square `KnSq` and `KnSq` and `KSq` are connected in a legal direction for the knight:

```
incheck-1(S,Side1,Side2,KnSq,KnPl,KSq,KPl):-
    opside(Side1,Side2),
    type(KPl,king), side(KPl,Side1), square(S,KSq,KPl),
    type(KnPl,knight), side(KnPl,Side2), square(S,KnSq,KnPl),
    legaldirection(Side2,knight,KnDr,1), connected(KnSq,KSq,KnDr).
```

The rule defining *remove-check-by-capture*, (`legalmove-2`) is similarly generated from the explanation in Figure 4(1c). The `move-2` rule (below) generates possible moves of player `MPl` of side `Side1` that can move to square `KnSq`. In this case, because the destination square `KnSq` will always be bound to a constant at run time, we simplify the recursive `movedirection` rule by removing the generator of bindings for `KnSq` (`square, side, type`). The rules are given below:

```
legalmove-2(S,do(op(...),S),Side1,Side2,KnSq,KnPl,KSq,KPl):-
    move-2(S,do(op(...),S),Side1,Side2,KnSq,KPl),
    not(incheck-3(do(op(...),S),Side1,Side2,KnPl,KSq,KPl)).

move-2(S,do(op(FSq,KnSq,Pl,KnPl),State),Side1,Side2,KnSq,KPl):-
    side(MPl,Side1), MPl\==KPl,
    type(MPl,MTy), square(S,FSq,MPl),
    legaldirection(Side1,MTy,MDr,Mc),
    connected(FSq,ISq,MDr),
    movedirection2(S,Mc,MDr,ISq,KnSq).
```

This concludes our description of the reformulation method. Next we briefly describe how the reformulated chess theory can benefit future learning.

### 3.4 Learning Knight-fork

We demonstrate how the functional features just learned, *incheck-by-knight*, *move-out-out-check* and *remove-check-by-capture* aid in the learning of an efficient definition of the feature *knight-fork*. Recall that the solution to the initial problem in Figure 1 involved a knight-fork: following the check threat and king move, the knight (moved to `d6`) simultaneously checked the king on `f7` and threatened the queen on `b7`. The king was forced to move out of check allowing the king to capture the queen.

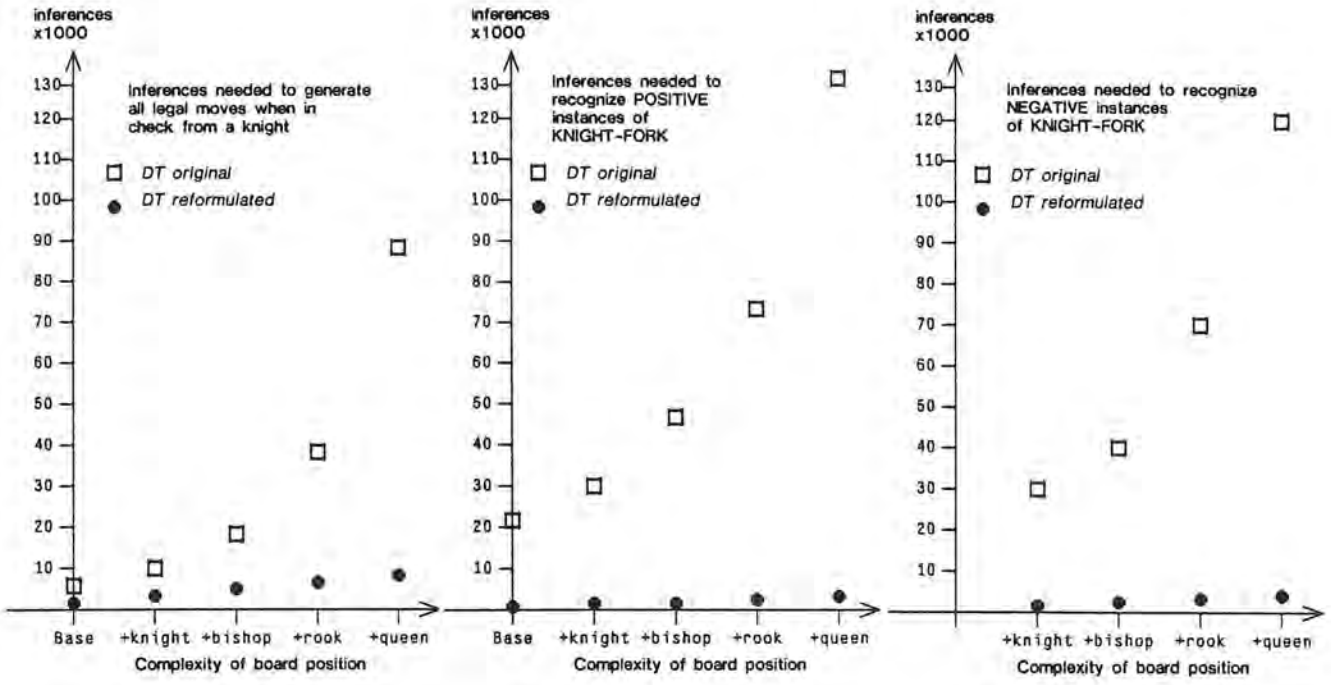
To learn the functional feature *knight-fork*, two board positions are presented, one where the white side is in a knight-fork, the other where the black side is in a knight-fork.

The reformulation process will proceed as before by generating explanations in terms of the current domain theory. In this case, because of the previous reformulation, the explanations will include the satisfied *incheck-by-knight* feature, the satisfied *move-out-of-check* feature (describing the king moves) and the unsatisfied *remove-check-by-capture* feature. The final rule generated is given below:

```
goodgoal-1(S,exchange(empty,queen),Side1):-
    incheck-1(S,Side1,Side2,KnSq,KnPl,KSq,KPl), !,
    (legalmove-3(S,Newstate1,Side1,Side2,KnSq,KnPl,KSq,KPl),
     not(legalmove-2(S,Newstate2,Side2,Side1,KnSq,KnPl,KSq,KPl))).
```

The final rule recognizes the *knight-fork* feature as a combination of existing and new features: A *knight-fork* exists in `S` with `Side1` to play if `incheck-1` is true in the state (i.e., *incheck-by-knight*) and there exists a move that can take the queen, `legalmove-3`, and there must not exist a `legalmove-2` (i.e., *remove-check-by-capture*) for the opponent.

This brief sketch of the reformulating method has demonstrated how the initial domain theory can be incrementally reformulated to one written in terms of relevant functional features.




(a) Generate all legal moves

(b) Recognize Positive Knight Forks

(c) Recognize Negative Knight Forks

Figure 5: Results for evaluating performance efficiency

# 4 Empirical Study of Reformulation

The principle goal of reformulating the domain of a problem solver is to improve its performance. In this section we present a brief empirical study that compares the performance of a simple chess problem solver using either the original or reformulated domain theory.

Currently, because the reformulator is not integrated within a practical problem solver, we measure performance as efficiency—the number of Prolog inference steps (Lis) required for the domain theory rules to generate the answer. In particular, we compare the number of logical inferences need by the original legal-move rules and the reformulated rules to generate all the legal moves in a given position. We also compare the classification costs of using a definition of knight-fork expressed in the original domain theory (given in Flann & Dietterich, 1988) with a definition expressed in the reformulated theory.

The results are shown in Figure 5. The vertical axis of the graphs is a count of the logical inferences needed for the performance task. In graph (a) this task is to generate all the legal moves for a given position. In graphs (b) and (c) the task is to classify board positions using the concept *knight-fork*, with graph (b) identifying positive instances and graph (c) identifying negative instances.

The horizontal axis represents a board position of increasing complexity. In graph (a) the Base position includes only a king and knight of *side1*, and a knight of *side2* (with *side1* side to play). The knight of *side2* is checking the king and can be captured by the knight of *side1*. There are eight legal moves in this position: seven by the king moving out of check and one by the knight taking the threatening knight. The +knight position is the Base position with two added knights (one of each side) placed on the board in such a way as not to affect the legal moves available. The +bishop is the +knight position with two bishops added similarly. The +rook and +queen represent further increases in complexity without affecting the legal moves available.

The graphs (b) and (c) are similarly set up for the knight-fork concept. In graph (b), the Base

position includes only a king and queen of *side1*, and a knight of *side2* (with *side1* side to play). The knight forks the king and the queen. In graph (b), the additional pieces are added such that they do not affect the concept definition. In graph (c), each position is like that for (b) but one of the pieces has been moved such that it can take the threatening knight. Hence, this graph demonstrates the performance of the definitions with negative instances. The results demonstrate that the reformulated theory ($DT_{new}$) performs much better than the original theory ($DT_{old}$). $DT_{new}$ requires significantly less resources to generate moves and classify knight fork positions. It is also interesting to note the insensitivity of $DT_{new}$ to irrelevant complexities in board. The search for a knight fork concept is much more constrained with $DT_{new}$ than with $DT_{old}$. If fact, recognizing *knight-fork* in the +**queen** position is approximately 60 times faster with $DT_{new}$ (2177 Lis) compared with $DT_{old}$ (130430 Lis).

We also anticipate a great benefit arising from this reformulation when the system is embedded within a chess problem solver. We expect the functional features will considerably constrain the game search (as described in the introduction) and lead to a significant improvement in overall performance.

# 5 Analysis

In this section we relate the approach to previous reformulation work that has stressed the importance of relevance and irrelevance in the domain theory and offer an explanation as to why the method works.

Amarel (1982) and more recently Subramanian and Genesereth (1987) view reformulation as identifying and exploiting irrelevances within a domain theory. If it can be shown that fact $f$ is irrelevant to proving fact $g$ in domain theory $M$ then $f$ can be removed. Subramanian gives an example of reformulating a family tree domain theory under the condition where the only queries will concern whether two people are in the same family. The original domain theory predicates describing *father* and *ancestor* are determined to be irrelevant and the domain theory is reformuated to include only *foundingfather* relations.

In the chess example we see the same use of irrelevance: when generating legal moves under the condition of a check by a knight, computation that generates moves other than moving the king or capturing the knight is irrelevant. The reformulated domain theory tests for the condition, and if true sanctions only moves that take the checking knight or move the king. In other words, the reformulated theory explicitly represents only what is *relevant* to generating moves when in check by a knight (i.e., it ignores what is irrelevant). The empirical results clearly demonstrate this. The reformulated theory is almost completely insensitive to the additional pieces since they are irrelevant to the goal.

In general then, we can view reformulation as a two stage process: First, given a domain theory $DT$, we identify parts of a domain theory $DT_I$ that are irrelevant when solving a goal $G$ under some condition $C$. Second, we reformulate $DT$ such that when it is used by a problem solver to solve $G$ under the condition $C$ it will use only what is relevant (referred to as $DT_R$), that is, ignore $DT_I$.

Under this view, the important problem of reformulation becomes identifying $DT_I$ and $DT_R$ in the domain theory. In Subramanian's formalism presented in (1987), $DT_I$ corresponds to fact $f$, $G$ to fact $g$ and $M$ to $DT$. Here, once $f$ is found, the reformulated theory simply becomes $M - f$. This formalism captures only a simple case of irrelevance since what is irrelevant, fact $f$, can be explicitly represented in the $DT$. In this case, the $DT$ requires little reformulation since the distinction between what is irrelevant ($f$) and relevant ($\neg f$) is already explicit in the syntax.

A more interesting case is when $DT_R$ and $DT_I$ are not explicit in the DT. For instance, in our chess example $DT_R$ is "the set of all king moves or the set of all moves that capture the checking knight." Although this can be described in the domain theory vocabulary, it is not an explicit predicate in $DT$.

The key idea in the method introduced in this paper is to represent $DT_R$ as sets of *generalized explanations.* This reduces the reformulation problem to: 1) identifying generalized explanations that express $DT_R$ and 2) compiling generalized explanations into efficient domain theory rules.

We identify $DT_R$ by employing the following assumption:

*What is relevant to solving G will be present in successful proofs of G.*

Hence, by clustering and generalizing successful proofs of $G$ we identify commonalities among the solutions to $G$ that express what is relevant to $G$. In the chess example, syntactic commonalities among the solutions traces identified the two general cases of legal moves given above.

This approach offers two advantages: The first advantage is that we avoid computationally intensive and unfocussed proofs. Examples both avoid the computational cost, because the proofs are fully instantiated; and provide focus, because they are supplied by a teacher. The second advantage is that we can identify $DT_R$ when it is not explicit in the original domain theory. In fact, the language used to describe $DT_R$—generalized explantions—is very rich: we can represent any legal partially instantiated proof tree of some goal $G$.

In summary, the method works because it exploits the fact that what is relevant in the domain theory is expressed as syntactic commonalities and differences among problem solving traces. We have shown how this method can improve the efficiency of parts of a chess domain theory and presented a framework in which the performance of a problem solver could be significantly improved.

## Acknowledgments

Discussions with my advisor, Tom Dietterich, helped both in the development of the IOE method and in understanding the reason for its success. Discussions with Devika Subramanian helped understand the relationship between irrelevance and the method presented. Jim Holloway and Caroline Koff provided useful comments on earlier drafts of this paper.

This research was partially funded by the National Science Foundation under grant numbers IST-8519926 and DMC-8514949.

## 6 Bibliography

Amarel S., (1982) "Expert Behavior and Problem Representations," in *Artificial and Human Intelligence,* A. Elithorn and R. Banerji (editors).

Dietterich T. G., & Flann N., S., (1988) "An Inductive Approach to Solving the Imperfect Theory Problem," in *Proceedings of the AAAI Symposium on Explanation-Based Learning,* 1988.

Fisher D. H., (1987) "Knowledge Acquisition Via Incremental Conceptual Clustering," in *Machine Learning,* Vol. 2, No. 2, 1097.

Flann, N. S. & Dietterich T. G. (1986) "Selecting Appropriate Representations for Learning from Examples," in *Proceedings of the Fifth National Conference on Artificial Intelligence,* 1986.

Flann N., S., & Dietterich T., G., (1988) "Induction Over Explanations: A Method that Exploits Knowledge to Learn From Examples," submitted to *Machine Learning.* (also OSU tech. report No. 88-30-3)

Genesereth M. R., & Nilsson H. J. (1987) *The Logical Foundations of Artificial Intelligence,* Morgan Kaufmann Pub. Los Altos. 1987.

Hasse K., W., (1986) "Discovery Systems," AI Memo 898, M.I.T. 1986.

Korf R., E., (1980) "Toward a Model of Representation Changes" in *Artificial Intelligence*, No. 14, pp. 41-78. 1980.

Lenat D. B. & Brown J. S. (1984). "Why AM and Eurisko Appear to Work," in *Artificial Intelligence*, Vol. 23, No. 3, pp 269-94, August 1984.

Minton S., (1985) "Selectively Generalizing Plans for Problem Solving." In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 596-599, 1985.

Mitchell T., Keller R., & Kedar-Cabelli, S. (1986) "Explanation-Based Generalization: A Unifying View," in *Machine Learning*, Vol. 1, No. 1, pp. 47-80, 1986.

Smith D. E., Genesereth M. R., & Ginsberg M. L., (1986) "Controling Recursive Inference," in *Artificial Intelligence*, Vol. 30, No. 3 pp. 343-390, 1987.

Subramanian D., & Genesereth M., R., (1987) "The Relevance of Irrelevance," in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 416-422, 1987.

# Appendix

This section gives relevant parts of the chess domain theory that define the legal move predicate. First we cover the way boards are described.

Each board state can be denoted by a constant such as `state1`. The squares on the board are also denoted by constants, for example, `a8`, `b2`, and so on. Finally, the pieces are each given names such as `wr1` for the white rook and `bn1` for the black knight. Empty squares are represented by an imaginary piece called `em` (essentially a null value). For example, the description of the board position illustrated in Figure 2(a) includes

```
square(state1,h6,wr1).  square(state1,c6,bn1).  square(state1,g6,em).  ...
```

With this representation, a board configuration can be represented by 64 assertions.

In addition, the structure of the chess board must be represented. A basic representation that captures the topology of the squares is the following:

```
connected(a7,a8,n).  connected(a7,b7,e).  connected(a7,b8,ne).  ...
```

The constants `n`, `e`, `ne`, and so on represent the eight directions of the compass points. In all, 372 `connected` assertions are needed.

We identify certain pieces (e.g., `wn1`, `wr1`, and so on) as all being white pieces. Similarly, we define groups of pieces (e.g., `wn1`, `bn2`, and so on) as being of the same type, `knight`:

```
side(wn1,w).  type(wn1,knight).  side(bq1,b).  type(bq1,queen).
```

Using these definitions, it is possible to define legal moves for each piece. We begin by stating, for each piece, the direction and maximum number of moves it can make. (Knight is treated specially see Flann & Dietterich, (1988)). As an example, the rules for rooks are:

```
legaldirection(Side,rook,n,8).  legaldirection(Side,rook,e,8).  ...
```

Several rules are required in order to define legal moves. In the body of the paper we gave the definition for `legalmove` here we define `possiblemove` and `incheck`

```
possiblemove(State,do(op(From,To,Playerm,Playert),State),Side1):-
    opside(Side1,Side2),
    side(Playerm,Side1), type(Playerm,Type), square(State,From,Playerm),
    legaldirection(Side1,Type,Direct,Count), connected(From,Next,Direct),
    movedirection(State,Count,Direct,Next,To,Playert,Type2,Side2).
```

A move is described as an operator function `op` in the situation calculus (using techniques recommended in Genesereth & Nilsson, 1987). The function `op` takes four arguments: the source square, the destination square, the name of the piece moved, and the name of the piece taken (`em` if no piece is taken). This rule checks to see that the indicated player, `Playerm`, is located on the source square; that `Playert` is located on the destination square; and that the indicated direction and number of squares is legal for the kind of piece being moved. In particular, the `movedirection` predicate recursively decrements the `Count` as it traverses connected squares in the indicated direction. It checks that all intervening squares are empty.

In check is defined similarly, as a move that takes the king by constraing the destination square to be the king.