CHESS-ENDGAME ADVICE: A CASE STUDY IN COMPUTER UTILIZATION OF KNOWLEDGE

by

M.H. van Emden

Research Report CS-80-05

Computer Science Dept.
University of Waterloo
Waterloo, Ontario, Canada

January 1980

Introduction

The intended contribution of this paper is the development of tools, in the form of a computer language, for the automatic utilization of knowledge. The field of study, having as objective to make such a utilization possible, is called "knowledge engineering". One proposed application is the development of a new breed of textbooks: such a textbook would have the form of an interactively usable computer program and database. The source code would be readable as a textbook, although more systematic, explicit, and precise than the ones we are used to. In interactive use the computerized textbook would solve problems in the application domain of the embedded knowledge. In this mode the user can be ignorant of everything except perhaps the broad outlines of the textbook's contents.

The program for the rook end game in chess, which is the centrepiece of this paper, is usable in the same way: for a human familiar with the rudiments of logic programming, a part of the source code is useful for learning the endgame for himself; but also, a computer equipped with a logic interpreter is able to use this knowledge to play itself the endgame against an independent opponent.

According to Michie's strategy for research in knowledge engineering, chess is used as test domain. Why chess? In the first place, a great deal of knowledge about chess has been recorded in the literature and is being used: advanced chess players study a lot. Also, among computer applications, chess has a relatively long history with several interesting reversals in the attitude of its practitioners. In the late fifties, when the programming work required to carry out the main outlines of Shannon's original proposal [15] first became feasible, the workers involved were very optimistic about the prospect of programs surpassing best human performance. In the late sixties, when a considerable effort using greatly improved processing power and programming tools had failed to yield anything even remotely approaching human top performance, the earlier optimism had changed to pessimism. In the late seventies, after another decade of steady improvement, a change to guarded optimism could be observed.

Roughly speaking, chess performance draws upon two distinct resources: look-ahead, that is, exploration of the game tree, and "knowledge" (including, or assisted by, "intuition", "experience", etc.). Human players rely almost exclusively on knowledge; this is probably necessary, given the peculiar limitations of their information processor. Computer players rely almost exclusively on look-ahead; this is probably not necessary, but caused by the primitive state of development of knowledge engineering.

Michie has proposed to express knowledge about chess endgames as "advice". This representation of knowledge is readable and useful to prospective human chess players and can also drive a computer equipped with an advice language interpreter to play in an expert fashion the endgames covered by the advice. Michie and his coworkers have formulated advice for several endgames [2, 3,12]; at least one interpreter has been implemented.

The purpose of the work reported in this paper is to investigate whether logic as a programming language [8] can take the place of the advice language proposed by Michie and perhaps, more generally, for expressing knowledge for purposes of knowledge engineering. The strategy followed in this paper is to introduce basic notations of game playing and logic programming by means of the game of Nim. These introductory sections conclude with a logic program for Nim. Subsequently the basic notions are refined and extended to be applicable to subgames of the rook endgame of chess.

We assume the reader to be familiar with the main outlines of the clausal form of logic and the notion of a logic program. A thorough treatment of logic programming is Kowalski's Logic for Problem-Solving [9] for which neither logic nor programming is a prerequisite.

2. The game of Nim

A position of the game of Nim can be visualized as a set of heaps of matches. Two players, called US and THEM, alternate making a move. As soon as a player whose turn it is to move, is unable to make a move, the game is over and that player has lost; the other player is said to have won. A move consists of taking at least one match from exactly one heap.

A two-person game such as Nim can be analyzed by means of a game tree. The nodes of a game tree are positions of the game, together with an indication which player is to move. In the game tree a node x is a descendant of node y if and only if y can be reached in one move from position x. The game tree shown below contains all positions reachable from the position where there are three heaps of matches, one with two matches and two with one match, and where US is to move.

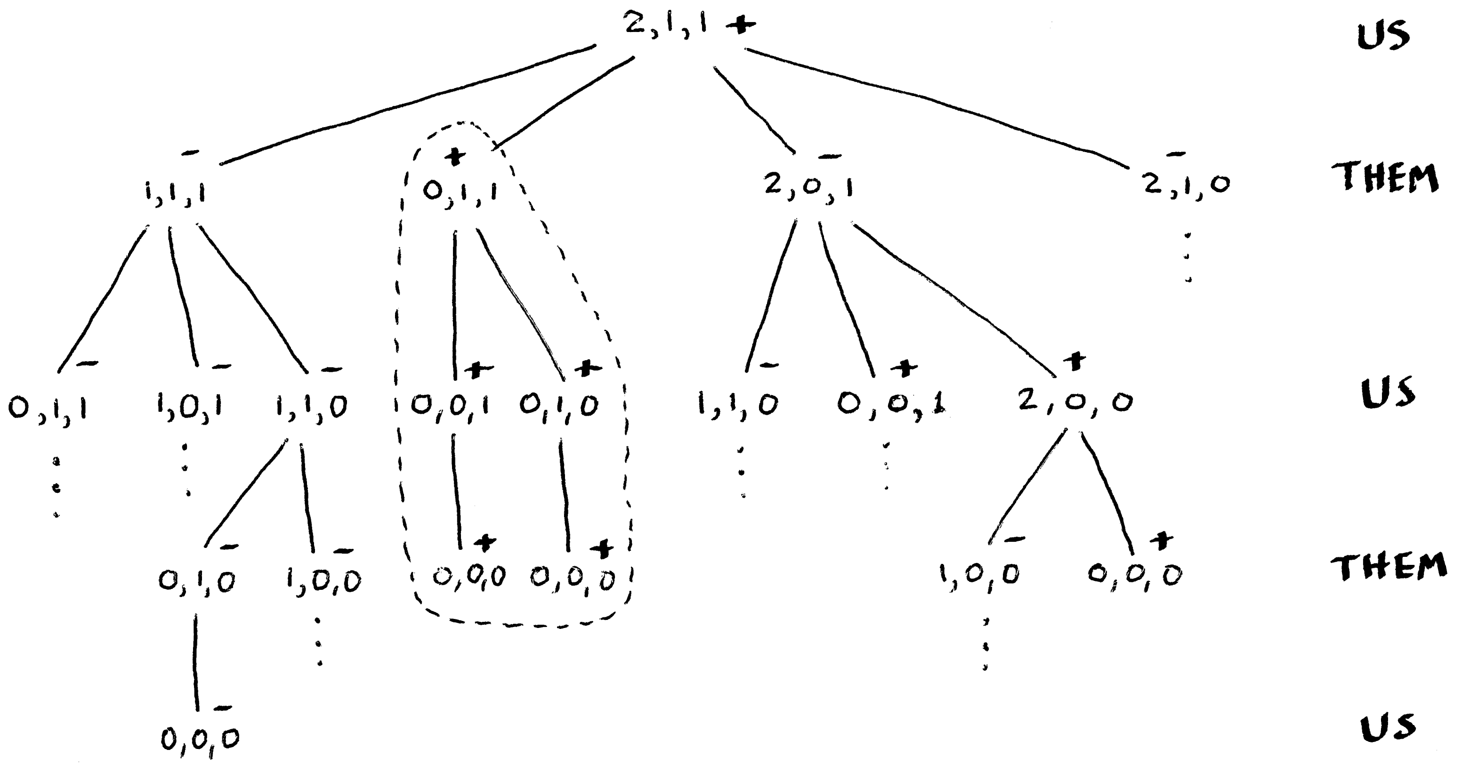

Fig. 2.1: Minimaxed game tree for a Nim position. The outlined subtree is the "forcing tree".

The nodes are also marked with their value, which is a + or a –, as determined by "minimaxing", a process which may be described as follows. A node 0,0 has value "+" when THEM is to move, then US has won. Similarly, a node 0,0 has value "–" when US is to move. The values of the other nodes are determined as a consequence of the assumption that, when there is a choice of move, both US and THEM will make the choice so as to be least favourable to the other player. That is, a node where THEM (US) is to move has as value the minimum (maximum) value of a descendant node, where "+" is regarded to be greater than "–".

These rules make it possible to determine the value of the root node of the tree by starting at the terminal nodes and working upwards, and this determines the "value of the game", that is, it determines whether or not US can win, starting at the root node. The rules apply to arbitrary games trees and apply also if there are more values than just two, such as in chess. It follows that it is in principle possible to determine the value of the initial position of chess, but it seems that this has not been done. Certainly the size of the game tree makes it impossible for the value to be determined by minimaxing the entire game tree.

We now state a dual pair of useful properties of game values. The properties can be regarded as a result of rephrasing our previous description of the minimaxing process.

Property 1: US can win starting in a position x, with US to move, (i.e. the value of x is "+"), if there exists a move transforming x to y such that it is not the case that US cannot win starting in y (i.e. the value of y is "+").

<u>Property 2</u>: US cannot win starting in position x, with THEM to move (i.e. the value of x is "-"), if there exists a move transforming x to y such that it is not the case that US can win starting in y (i.e. the value of y is "-").

In these properties, "US can win" means that there is a choice of moves by US which makes it impossible for THEM to win. "US cannot win" means that there is a choice of moves by THEM which makes it impossible for US to win.

These properties have been selected from other possible ones because it is easy to see that they are true and because they are simply expressed in logic with a result that can be used as a computer program for computing the value of a game (provided that the game tree is small).

3. The states and moves of NIM expressed in logic

In logic, objects are represented by terms. A term is either a constant, or an expression of the form $f(t_1,\ldots,t_k)$ where f is a function symbol and $t_1,\ldots,t_k$ are terms. This recursive definition allows the existence of terms of arbitrarily great complexity to be built up from few and simple constants and function symbols. As our first example we represent as terms the heaps of matches which occur in Nim positions.

An empty heap is represented by the constant 0. If a heap is represented by a term t, then the heap with one match more is represented by successor(t). Thus, for example, a heap with two matches is represented by the term successor(successor(0)). For actual use we prefer a shorter function symbol and we have chosen "+" instead of successor (no connection with the game value denoted by the same symbol), so that we get +(+(0)) for the heap of two. Using postfix instead of prefix notation allows us to write t+ for the heap with one more than the heap represented by t. So we write 0++ for the heap of two.

A position is not in general a single heap, but a set of heaps, which we will regard for convenience as a sequence. The empty sequence is represented by the constant nil. The sequence resulting from concatenating a heap x onto a sequence y is represented by conc(x,y). For example, the initial position (2,1) of the game tree shown earlier is represented by

conc(0++,conc(0+,nil)).

Again we replace the function symbol conc by something more compact; in this case a single dot. This gives

•(0++,•(0+,nil))

Using infix notation instead of prefix notation further compacts this to

$0{+}{+}\cdot(0{+}\cdot nil)$ with the understanding that plus has higher precedence than dot.

We now have the notational apparatus for positions. To be able to express properties of moves we need atomic formulas and clauses. For example the atomic formula Append(x,y,z) expresses the relation between sequences x,y, and z where z is the result of appending y to x. In general, an atomic formula has the form $P(t_1,\ldots,t_n)$ where P is a predicate symbol and $t_1,\ldots,t_n$ are terms. A useful property of the Move relation is expressed by the clause

$$\mathrm{Move}(x,y) \leftarrow \mathrm{Append}(u,v_1\cdot w,x)\ \&\ \mathrm{Takesome}(v_1,v_2)\ \&\ \mathrm{Append}(u,v_2\cdot w,y).$$

Here x,y,u, and w are positions, i.e. sequences of heaps; v_1 and v_2 are heaps. The clause should be read as

"For all x and y,

"y is the result of a move from x if there exist u,v_1,v_2, and w

"such that x is the result of appending $v_1\cdot w$ to u and v_2

"is a result taking some (matches) from v_1 and y is

"the result of appending $v_2\cdot w$ to u.

To say that, for given x, there exist u,v, and w such that append(u,v.w,x), can be interpreted as saying that v is an arbitrary heap occurring in sequence x. If we accept this way of saying "occurs in", then we see that the above clause says:

"For all x and y,

"y is the result of a move from x if y differs from x only in

"a heap v_2 which is the result of taking some matches from

"the corresponding heap v_1 in x.

We will use the following properties of the append relation.

append(nil,y,y).

This says that for all y,y is the result of appending y to the empty list.

append(u•x,y,u•z) ← append(x,y,z)

This says that, for all u,x,y, and z,u•z is the result of appending y to u•x if z is the result of appending y to x.

The "take some" relation has the following useful properties:

Takesome(x+,x)

This says that some have been taken from a heap if the result is one less.

Takesome(x+,y) ← Takesome(x,y)

This says that some have been taken from a heap if the result is the result of taking some from a heap containing one less. That is, to take some, take one, and then some.

Logic programming consists of collecting a set of clauses, each expressing a useful property of the relation to be computed (such as Move) and of relations auxiliary thereto (such as Append and Takesome). The following is a PROLOG program for computing moves.

```
OP(+,SUFFIX,101).

APPEND(NIL,*Y,*Y).
APPEND(*U.*X,*Y,*U.*Z) <- APPEND(*X,*Y,*Z).

MOVE(*X,*Y) <- APPEND(*U,*X1.*V,*X) & TAKESOME(*X1,*X2)
               & APPEND(*U,*X2.*V,*Y).

TAKESOME(*X+,*X).
TAKESOME(*X+,*Y) <- TAKESOME(*X,*Y).
```

Fig. 3.1: Listing of PROLOG program for Nim moves

In Figure 3.1 we see a declaration for the postfix operator "+" with precedence 101. (The operator "•" is built in, with precedence 100). Variables are distinguished from constants (such as nil) by identifiers beginning with an asterisk.

4. The value of a game expressed in logic

We obtain a logic program for playing a game by expressing in logic the Properties 1 and 2 of section 2. Initially we do not stay within the clausal form of logic.

A straightforward expression of Properties 1 and 2 is:

$$\forall x.\ \mathrm{Uscanwin}(\mathrm{ustomove},x) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ \neg\,\mathrm{Uscannotwin}(\mathrm{themtomove},y).$$
$$\forall x.\ \mathrm{Uscannotwin}(\mathrm{themtomove},x) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ \neg\,\mathrm{Uscanwin}(\mathrm{ustomove},y).$$

Apparently, we only need Uscanwin with ustomove as first argument. This first argument is therefore superfluous: we use a one-argument predicate S instead. A similar consideration causes us to replace Uscannotwin by a one-argument predicate T.

The resulting formulas are

$$\begin{aligned}&\forall x.\ S(x) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ \neg\,T(y).\\ &\forall x.\ T(x) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ \neg\,S(y). \qquad \ldots.(4.1)\end{aligned}$$

$S(x)$ means: for a suitable choice of moves, US will win starting in position x with US to move, whatever the choice of moves by THEM.

$T(x)$ means: for no choice of moves, US can win starting in position x with THEM to move, against a choice of moves by THEM which is least favourable to US.

However, S does not tell us _how_ we can win: it could at least incorporate the position resulting from the first suitable move. Hence we prefer

$$\forall x.\ S(x,y) \leftarrow \exists y.Move(x,y) \& \neg \exists z.T(y,z)$$
$$\forall x.\ T(x,y) \leftarrow \exists y.Move(x,y) \& \neg \exists z.S(y,z). \quad \ldots.(4.2)$$

S(x,y) means:	S(x), and, moreover, a first suitable choice is from x to y.
T(x,y) means:	T(x), and, moreover, a first choice least favourable to US is from x to y.

However, in order to determine that US can win, more information must have been obtained than just the first move, unless it immediately ended the game. In general, for each of the countermoves by THEM, US must have determined a suitable response. All information required by US to win the game against any defence by THEM is contained in a subtree of the game tree called the _forcing tree_. The subtree outlined in figure 2.1 is an example of a forcing tree.

To obtain a program that also computes the forcing tree, we first modify (4.1) to as to avoid negations. Let TT(x) mean that THEM must lose against best play by US. Then we have

$$\forall x.\ S(x) \leftarrow \exists y.Move(x,y)\ \&\ TT(y)$$
$$\forall x.\ TT(x) \leftarrow (\forall y.Move(x,y) \rightarrow S(y))$$

Only the first implication is in a form suitable for being handled by current interpreters for logic programs. We reformulate the second implication so as to obtain a conjunction in the condition.

$$\forall x.\ S(x) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ TT(y)$$

$$\forall x.\ TT(x) \leftarrow \exists z.\mathrm{Moves}(x,z)\ \&\ \mathrm{All}(z)$$

z is a list containing all moves possible (for THEM) from position x. $\mathrm{All}(z)$ means that all positions u in list z have the property that $S(u)$:

$$\mathrm{All}(\mathrm{nil})$$

$$\forall u,v.\mathrm{All}(u \cdot v) \leftarrow S(u)\ \&\ \mathrm{All}(v)$$

We now have to decide how to represent a forcing tree as a term. The format for the forcing tree is as follows

$$\mathrm{ourmove} \cdot ((\mathrm{theirmove}_1 \cdot ft_1) \cdot \ldots \cdot (\mathrm{theirmove}_n \cdot ft_n)) \cdot \mathrm{nil}$$

where each of $ft_1,\ldots,ft_n$ are themselves forcing trees.

Incorporating the forcing tree into the relations S and TT we get

$$\forall x,z.S(x,y \cdot z) \leftarrow \exists y.\mathrm{Move}(x,y)\ \&\ TT(y,z)$$

$$\forall y,u.TT(y,u) \leftarrow \exists z.\mathrm{Moves}(y,z)\ \&\ \mathrm{All}(z,u)$$

$$\mathrm{All}(\mathrm{nil},\mathrm{nil})$$

$$\forall u,v,w.\mathrm{All}(u \cdot v,(u \cdot w) \cdot z) \leftarrow S(u,w)\ \&\ \mathrm{All}(v,z)$$

Each of the above formulas are clauses and can be handled by PROLOG interpreters. Additional clauses are needed to specify the Move relation, as in figure 3.1. A valuable feature of logic programming is that one only has to specify what the admissible moves are; it is not necessary to program a move generator.

In current PROLOG systems there are difficulties in reconciling this attractive way of specifying Move with a specification of the Moves relation, where the second argument is a list of all moves. We can do it in logic, but not in our PROLOG implementation. Therefore we do not compute forcing trees and work from the version (4.2) which only specifies the first move by US.

A complete PROLOG program for Nim using game trees is given in figure 4.1. Even for less than a dozen matches the cost of running this program can be prohibitive. This is not caused by excessive slowness of PROLOG: in spite of its elegance the efficiency of this program is within an order of magnitude of what is possible when exhaustively traversing game trees.

It is well-known [1] that, if Nim can be won then it can be won by a simple calculation that can be done in human head for positions that would defeat the fastest minimaxing programs on the fastest computers: select a move such that the resulting position has the property that the digital sum of the sizes of the heaps is zero.

It seems plausible that such a short computation of the value is only possible for games of great simplicity and that the shortest possible computation for a game like chess is considerably longer, although still much shorter than minimaxing. Perhaps for the game of Kalah there exists a computation for the value of the game which is not much longer than computing the digital sum usable in Nim.

```
S(*X,*Y) <- MOVE(*X,*Y) & ~T(*Y,*Z).
T(*X,*Y) <- MOVE(*X,*Y) & ~S(*Y,*Z).

OP(+,SUFFIX,101).

APPEND(NIL,*Y,*Y).
APPEND(*U.*X,*Y,*U.*Z) <- APPEND(*X,*Y,*Z).

MOVE(*X,*Y) <- APPEND(*U,*X1.*V,*X) & TAKESOME(*X1,*X2)
             & APPEND(*U,*X2.*V,*Y).

TAKESOME(*X+,*X).
TAKESOME(*X+,*Y) <- TAKESOME(*X,*Y).
```

fig. 4.1

A Nim-playing program in PROLOG

5. An outline of advice-driven endgame programs in logic

As we saw, an all-powerful chess-playing program can, in theory, be very simple: one just supplements the logic program for a universal game player, namely,

$$S(x,y) \leftarrow Move(x,y) \ \& \ \neg T(y,z)$$
$$T(x,y) \leftarrow Move(x,y) \ \& \ \neg S(y,z)$$

with a program for Move expressing the rules of chess. Such a theory, however, would have to be unaware of complexity issues, because this program would have to explore the entire game tree. For all of chess this tree is so enormous that a computer of a size near the limits imposed by the size of the universe and operating at speeds near the limits imposed by fundamental physical constants, would still require eons to determine the value of the game.

Even in the rook endgame the game tree is too large to make its minimaxing feasible: most initial positions require more than 10 moves by each side, where the number of possibilities for the rook side often exceeds 20 (almost all of these are usually idiotic). Yet one can teach a motivated, averagely gifted child of 9 years old how to play the rook endgame. The explanation of this difference that humans use 'knowledge', or have acquired certain 'skills' to play a game such as this and rely very little on game trees. If look-ahead is used at all, it is typically based on small, error-ridden fragments of game trees.

Our basic hypothesis is that the utilization of knowledge, or whatever it is that allows humans to play the game, is an information processing task and as such in principle executable by computer. Analogs of the old biological controversy of mechanism versus vitalism crop up in several other disciplines. In each of these one's position has to be stated anew. The above hypothesis states our position here.

Michie has proposed to express knowledge about how to play a chess endgame in terms of "advice". An algorithm in a typical high-level programming language is not a suitable form for expressing such knowledge, being both extremely hard to write and to read. This judgment is based on the one example known to us: the rook endgame program in Algol 60 by C. Zuidema [18]. At another extreme we have treatments of endgames in such standard texts as Fine's Basic Chess endings [7]. Although these do contain some pieces of advice, no attempt has been made at completeness or explicitness. Advice in Michie's sense lies between these two extremes: although sufficiently complete and explicit to be followed automatically, it is also learnable by a human for playing the game herself. By "learnable" we mean that a human can memorize the advice and use it without reference to its text, something which is inconceivable with Zuidema's Algol-60 program.

Michie and his co-workers have compiled advice for several chess endgames. We use Bratko's [2] advice table for the rook endgame. Its English form can be given as follows:

1. Look for a way to mate the black *) king in two moves, trying checking moves first. The depthbound is 2.
2. If the above is not possible, then squeeze, that is, look for a way to decrease the area of the board to which the black king is confined by the white rook, considering rook moves only. The depthbound is 1.
3. If the above is not possible, then approach, that is, look for a way to move, preferring diagonal moves, the white king closer to the black king so as to help the rook to squeeze in a later move. The depthbound is 1.
4. If the above is not possible, then keep room, that is, look for a way of maintaining the present achievements in the sense of 2 and 3. This can always been done with a move by the white king; diagonal moves are to be tried first. The depthbound is 1.
5. If the above is not possible, then divide, that is, move the white rook into such a position that it divides the kings either vertically or horizontally. The depthbound is 3.

Each of the rules 1, 2, 3, 4, and 5 advises white to achieve a certain goal in the game. The rules are in order of decreasing ambitiousness. In a typical endgame we have to settle initially for the least ambitious goal, and progress towards more ambitious ones. For this reason the goals are called better goals. The rook endgame is subdivided into a sequence of subgames, each defined by a rule of advice, and each with its own criterion of winning, namely one of the better goals. The criterion for losing is not to achieve the

*) We assume that US has white and has the rook.

holding goal. For each of the subgames the holding goal is to avoid losing the rook and to avoid stalemate. Note that this is not a loss according to the rules of chess, but only according to the subgame we define for the purpose of our advice-driven endgame program.

The depthbound given with each rule is the greatest depth to which the game tree of the subgame defined by the rule is to be explored.

The rules typically also contain move constraints, that is, advice about the type of move to try first, or to try at all. This greatly reduces the number of possibilities to explore, facilitating the task both for a human and for a computer. We formalize the existence of move constraints which exclude certain moves, by defining the subgame to have this different set of moves.

Thus each rule of advice determines a different (sub)game. Each subgame has been chosen in such a way that is is computationally feasible to explore its complete game tree, and this is what an advice-driven program for the rook endgame does.

In order to obtain a logic program for subgames we have to generalize the two-line game player

$$S(x,y) \leftarrow Move(x,y) \ \& \ \neg\, T(y,z)$$

$$T(x,y) \leftarrow Move(x,y) \ \& \ \neg\, S(y,z)$$

used up till now. It assumes identical move-sets for both players; to accommodate move constraints we distinguish S-moves (by US) and T-moves (by THEM). It also assumes that the criterion for winning for one player is that the other is to move but cannot move; to accommodate better goals and holding goals, we introduce explicit criteria for a win by US (Swon) and a win by THEM (Twon). Hence the following elaboration of the two-line game player.

$$S(x,x) \leftarrow \mathrm{Swon}(x)$$
$$S(x,y) \leftarrow \mathrm{Smove}(x,y) \ \& \ \neg\, T(y,z)$$
$$T(x,x) \leftarrow \mathrm{Twon}(x)$$
$$T(x,y) \leftarrow \mathrm{Tmove}(x,y) \ \& \ \neg\, S(y,z)$$

Several other elaborations are needed. Better goals often depend not on a single position, but on the two last positions. For example, when the better goal is to squeeze, the amounts of room for the black king in the last two positions have to be compared, to see if there was a decrease. Therefore Swon has to have two successive positions as argument, as in the next version of the game player:

$$S(x,y) \leftarrow \mathrm{Smove}(x,y) \ \& \ S_1(x,y)$$
$$S_1(x,y) \leftarrow \mathrm{Swon}(x,y)$$
$$S_1(x,y) \leftarrow \neg\, T(y,z)$$
$$T(x,y) \leftarrow \mathrm{Tmove}(x,y) \ \& \ T_1(x,y)$$
$$T_1(x,y) \leftarrow \mathrm{Twon}(x,y)$$
$$T_1(x,y) \leftarrow \neg\, S(y,z)$$

For the sake of symmetry Twon was also given two arguments, although in our program only one is used.

The above game player only plays one of the subgames defined by a rule of advice. In order to avoid writing a different game player for each rule, the predicates are given an extra parameter, taking a rule as value. Another extra parameter is for the depthbound. These refinements are incorporated in the following version of the game player:

$$
\begin{aligned}
S(x,y,rule,n) &\leftarrow Gt(n,0)\ \&\ Diff(n,1,n1)\ \&\\
&\quad Moveconstr(rule,mc)\ \&\ Smove(mc,x,y)\ \&\\
&\quad S_1(x,y,rule,n1)\\
S_1(x,y,rule,n) &\leftarrow Swon(x,y,rule)\\
S_1(x,y,rule,n) &\leftarrow \neg\, T(y,z,rule,n)
\end{aligned}
$$

$$
\begin{aligned}
T(x,y,rule,n) &\leftarrow Tmove(x,y)\ \&\ T_1(x,y,rule,n)\\
T_1(x,y,rule,n) &\leftarrow Twon(x,y,rule)\\
T_1(x,y,rule,n) &\leftarrow \neg\, S(y,z,rule,n)
\end{aligned}
$$

The following additional explanations are necessary. Gt and Diff are pre-defined relations: Gt(x,y) holds if x and y are integers and $x > y$; Diff(x,y) holds if x,y, and z are integers and $x-y=z$. Moveconstr will be defined in such a way that Moveconstr(x,y) holds if x is a rule and y is the move-constraint component of x.

We have to formalize also the overall process of trying to find most ambitious rule of advice that works in a given position. In the overall process US first decides which rule is applicable (i.e., which subgame to play), determines the appropriate move, and waits for a counter move. At this level a one-person game is played, where each move is an exchange of moves at the lower level.

```
Win(pos)  ← Mated(pos)
Win(pos1) ← Theirmove(pos1, pos2) & Ourmove(pos2,pos3) & Win(pos3)
```

This says that there exists a sequence of exchanges leading to a win if THEM is Mated, or if an exchange can bring about a position (pos3) from which such a sequence can be found.

The following clauses provide the connection with the S defined earlier.

```
Ourmove(pos1,pos2) ← Advice(rule) & Depthbound(rule,n) &
                     S(pos1,pos2,rule,n).
Advice(mate      :  notrooklost  :  checkfirst   :  2)
Advice(squeeze   :  notrooklost  :  rookmove     :  1)
Advice(approach  :  notrooklost  :  kdmovefirst  :  1)
Advice(keeproom  :  notrooklost  :  kdmovefirst  :  1)
Advice(divide    :  notrooklost  :  anymove      :  3)
```

The terms in the arguments to Advice are constants; those in the other clauses are variables. In PROLOG source code variables are distinguished from constants by being preceded by an asterisk.

Each argument to Advice encodes a rule; its components are separated by the infix function symbol ":". The components denote successively the better goal, the holding goal, the move constraint, and the depth bound.

Backtracking plays an important role in the execution of a PROLOG program. For example, in the above definition of Ourmove, Advice picks up the first rule listed as such. If S subsequently fails to find a winning (in the subgame sense) move, then the interpreter backtracks back to Advice, which then invokes its next untried definition.

Sometimes, however, backtracking would be highly undesirable, such as in the two clauses for minmax in fig. 6.7:

```
minmax(x,y,x,y) ← lt(x,y) & /.
minmax(x,y,y,x),
```

where lt(x,y) if x is less than y. Minmax uses the first two arguments as input and outputs them in the last two arguments in increasing order of magnitude. Suppose this definition is used to solve minmax (3,4,x,y), then the first clause applies and x is set to 3, y to 4. Some later failure should never cause the system to backtrack and use the second clause. The slash "/" in the first clause inhibits such backtracking.

Figures 6.1,...,6.7 list the source code of the advice-driven PROLOG program for the rook endgame. The top level of this hierarchically structured program has been discussed so far. We hope to have made familiar the part in figures 6.1 and 6.2. Figure 6.6 provides a definition of the moves.

6. An advice table for the rook endgame

The figures below shows Michie's format for an advice table for the rook endgame. The table is from Bratko [2] with some slight modifications. It should be noted that this advice is uncharacteristically simple. Moreover, advice in Michie's sense consists of several cooperating advice tables.

RULE NUMBER	NAME OF BETTER GOAL	MATED	NEWROOMLT	REXPOSED	OKCSNMDLT	RDIVIDES	RDIVIDES OR LPATT	OKORNDLE	ROOMGT2 OR NOT OKEDGE	NOT ROOKLOST	MOVE CONSTRAINT	DEPTHBOUND
1	MATE	y	-	-	-	-	-	-	-	y	CHECKFIRST	2
2	SQUEEZE	-	y	y	-	y	-	-	-	y	ROOKMOVE	1
3	APPROACH	-	-	y	y	-	y	-	y	y	KDMOVEFIRST	1
4	KEEPROOM	-	-	y	-	y	-	y	y	y	KDMOVEFIRST	1
5	DIVIDE	-	-	y	-	y	-	-	-	y	ANYMOVE	3

better-goal predicates (MATED … ROOMGT2 OR NOT OKEDGE)

holding goal (NOT ROOKLOST)

Each better goal is a conjunction of the predicates marked with Y (yes). Fig. 6.3 displays the PROLOG clauses for the advice rules and for the definition of the better goals. The rule only contains the name of a better goal, such as "squeeze"; below it is a clause defining the better goals as conjunctions of predicates. These in turn are defined and explained in figures 6.4, 6.5, and 6.7.

The typical format of a term denoting a board position is

$$\left.\begin{array}{l}\text{theirturn}\\ \text{ourturn}\end{array}\right\} \text{:*tk:*ok:*or.}$$

The first component indicates who is to move. The remaining components are the position of their king, our king, and the (our) rook, respectively. Each of these positions are given as the file number (i) paired by means of a dot to the rank number (j). With this detail included, the typical format of a position is

$$\left.\begin{array}{l}\text{theirturn}\\ \text{ourturn}\end{array}\right\} \text{:*itk•*jtk:iok•*jok:*ior•*jor}$$

The listing uses a tilde instead of the usual negation sign ¬. Also, we use A ← B | C (read: A if (B or C)) as a shorthand for the two separate clauses A ← B and A ← C.

```
/* THE TOP LEVEL: HERE ADVICE IS SELECTED.
   WIN(*X) MEANS THAT WE CAN WIN WITH THEM STARTING IN POSITION X.
*/

WIN(*POS) <- MATED(*POS).
WIN(*POS1) <- THEIRMOVE(*POS1,*POS2) & OURMOVE(*POS2,*POS3) & WIN(*POS3).

OURMOVE(*POS1,*POS2)
       <- ADVICE(*RULE,*POS1) & DEPTHBOUND(*RULE,*N) & WRITE(TRYING:*RULE) &
          S(*POS1,*POS2,*RULE,*N) &
          WRITE('NEW POSITION (TURN:YOUR KING:OUR KING:OUR ROOK) IS:') &
          WRITE(*POS2).

THEIRMOVE(THEIRTURN:*ITK1.*JTK1:*OK:*OR,OURTURN:*ITK2.*JTK2:*OK:*OR)
         <- WRITE('ENTER NEW POSITION AS I.J FOR YOUR KING:') &
            READ(*ITK2.*JTK2) &
            TEST(THEIRTURN:*ITK1.*JTK1:*OK:*OR,OURTURN:*ITK2.*JTK2:*OK:*OR).

TEST(*POS1,*POS2) <- TMOVE(*POS1,*POS2) & /.
TEST(*POS1,*POS2) <- WRITE('NOT A VALID MOVE; TRY AGAIN') & FAIL.
```

fig: 6.1

```
/* THIS DEFINES A SUBGAME DETERMINED BY A PREVIOUSLY SELECTED RULE */

S(*POS1,*POS2,*RULE,*N)
 <- GT(*N,0) & DIFF(*N,1,*N1) & MOVECONSTR(*RULE,*MC) &
    SMOVE(*MC,*POS1,*POS2) & S1(*POS1,*POS2,*RULE,*N1).

S1(*POS1,*POS2,*RULE,*N) <- SWON(*POS1,*POS2,*RULE) & /.
S1(*POS1,*POS2,*RULE,*N) <- ~T(*POS2,*,*RULE,*N).

T(*POS1,*POS2,*RULE,*N)
 <- TMOVE(*POS1,*POS2) & T1(*POS1,*POS2,*RULE,*N).
T1(*POS1,*POS2,*RULE,*N) <- TWON(*POS1,*POS2,*RULE) & /.
T1(*POS1,*POS2,*RULE,*N) <- ~S(*POS2,*,*RULE,*N).

SMOVE(*MC,OURTURN:*TK:*OK1:*OR1,THEIRTURN:*TK:*OK2:*OR2)
     <- TRIALMOVE(*MC,OURTURN:*TK:*OK1:*OR1,THEIRTURN:*TK:*OK2:*OR2) &
        ~(*OK2=*OR2) & ~KINGRULE(*TK,*OK2) &
        ~STALEMATE(THEIRTURN:*TK:*OK2:*OR2).

TMOVE(THEIRTURN:*TK1:*OK:*OR,OURTURN:*TK2:*OK:*OR)
     <- KINGRULE(*TK1,*TK2) & ~INCHECK(*:*TK2:*OK:*OR).

SWON(*POS1,*POS2,*RULE) <- BETTERGOAL(*RULE,*B) & TRUE(*B,*POS1,*POS2).
TWON(*POS1,*POS2,*RULE) <- HOLDINGGOAL(*RULE,*H) & ~TRUE(*H,*POS1,*POS2).

TRIALMOVE(CHECKFIRST,*POS1,*POS2)
         <- TRIALMOVE(ROOKMOVE,*POS1,*POS2) & INCHECK(*POS2).
TRIALMOVE(CHECKFIRST,*POS1,*POS2)
         <- TRIALMOVE(ANYMOVE,*POS1,*POS2).
TRIALMOVE(ROOKMOVE,OURTURN:*TK:*OK:*OR1,THEIRTURN:*TK:*OK:*OR2)
         <- ROOKRANGE(OURTURN:*TK:*OK:*OR1,*RANGE) &
            ROOKRULE(*OR1,*OR2,*RANGE) & ~HOPPED(*OK,*OR1,*OR2).
TRIALMOVE(KDMOVEFIRST,OURTURN:*TK:*OK1:*OR,THEIRTURN:*TK:*OK2:*OR)
         <- KINGRULE(*OK1,*OK2).
TRIALMOVE(ANYMOVE,*POS1,*POS2)
         <- TRIALMOVE(ROOKMOVE,*POS1,*POS2).
TRIALMOVE(ANYMOVE,*POS1,*POS2)
         <- TRIALMOVE(KDMOVEFIRST,*POS1,*POS2).

HOPPED(*I.*JK,*I.*JR1,*I.*JR2) <- BETWEEN(*JK,*JR1,*JR2).
HOPPED(*IK.*J,*IR1.*J,*IR2.*J) <- BETWEEN(*IK,*IR1,*IR2).
```

Fig. 6.2

```
/*************************************************************************

    THE ADVICE TABLE; THE FORMAT IS:
       BETTERGOAL:HOLDINGGOAL :MOVECONSTRAINT:DEPTHBOUND                  */

ADVICE(MATE       :NOTROOKLOST :CHECKFIRST    :    2     ,*POS) <- CLOSE(*POS).
ADVICE(SQUEEZE    :NOTROOKLOST :ROOKMOVE      :    1     ,*POS).
ADVICE(APPROACH   :NOTROOKLOST :KDMOVEFIRST   :    1     ,*POS).
ADVICE(KEEPROOM   :NOTROOKLOST :KDMOVEFIRST   :    1     ,*POS).
ADVICE(DIVIDE     :NOTROOKLOST :ANYMOVE       :    3     ,*POS).

/* SELECTORS FOR ADVICE COMPONENTS */

BETTERGOAL(*X:*:*:*,*X).                HOLDINGGOAL(*:*X:*:*,*X).
MOVECONSTR(*:*:*X:*,*X).                DEPTHBOUND (*:*:*:*X,*X).

/* BETTER GOALS */

MATE(*,*POS) <- MATED(*POS) & WRITE('MATE').

SQUEEZE(*POS1,*POS2) <- NEWROOMLT(*POS1,*POS2) & ~REXPOSED(*POS2) &
                        RDIVIDES(*POS2) & WRITE('SQUEEZE').

APPROACH(*POS1,*POS2)
       <- OKCSNMDLT(*POS1,*POS2) & ~REXPOSED(*POS2) &
          RDIVIDESORLPATT(*POS2) &
          ROOMGT2ORNOTOKEDGE(*POS2) & WRITE('APPROACH').

KEEPROOM(*POS1,*POS2)
       <- RDIVIDES(*POS2) & ~REXPOSED(*POS2) &
          OKORNDLE(*POS1,*POS2) & ROOMGT2ORNOTOKEDGE(*POS2) &
          WRITE('KEEPROOM').

DIVIDE(*POS1,*POS2) <- ~REXPOSED(*POS2) & RDIVIDES(*POS2).

/* HOLDING GOAL */

NOTROOKLOST(*POS,*) <- ~ROOKLOST(*POS).

/* ADVICE CONDITIONS */

CLOSE(*:*TK:*OK:*) <- MDIST(*TK,*OK,*D) & LT(*D,4) & EDGE(*TK).

/*************************************************************************/
```

Fig. 6.3

```
/* ADVICE TABLE PREDICATES */

MATED(*POS) <- INCHECK(*POS) & ~TMOVE(*POS,*).

/* NEW ROOM (OF THEIR KING) LESS THAN */
NEWROOMLT(*POS1,*POS2)
         <- ROOM(*POS1,*R1) & ROOM(*POS2,*R2) & LT(*R2,*R1).

/* ROOK EXPOSED: ROOK TOO CLOSE TO THEIR KING */
REXPOSED(THEIRTURN:*TK:*OK:*OR)
        <- DIST(*TK,*OR,*TD) & DIST(*OK,*OR,*OD) & LT(*TD,*OD).
REXPOSED(OURTURN:*TK:*OK:*OR)
        <- DIST(*TK,*OR,*TD) & DIST(*OK,*OR,*OD) &
           DIFF(*OD,1,*OD1) & LT(*TD,*OD1).

/* OUR KING (TO) CRITICAL SQUARE (HAS) NEW MANHATTAN DISTANCE LESS THAN */
OKCSNMDLT(*:*ITK1.*JTK1:*OK1:*IOR.*JOR,*:*:*OK2:*IOR.*JOR)
         <- DIFF(*ITK1,*IOR,*DI) & SIGN(*DI,*SDI) & SUM(*IOR,*SDI,*ICS) &
            DIFF(*JTK1,*JOR,*DJ) & SIGN(*DJ,*SDJ) & SUM(*JOR,*SDJ,*JCS) &
            MDIST(*OK1,*ICS.*JCS,*DIST1) & MDIST(*OK2,*ICS.*JCS,*DIST2) &
            LT(*DIST2,*DIST1).

/* ROOK DIVIDES (THE KINGS) */
RDIVIDES(*:*ITK.*JTK:*IOK.*JOK:*IOR.*JOR)
        <- BETWEEN(*IOR,*ITK,*IOK) | BETWEEN(*JOR,*JTK,*JOK).
           (LT(*JTK,*JOR) & LT(*JOR,*JOK)) | (GT(*JTK,*JOR) & GT(*JOR,*JOK)) .

/* ROOK DIVIDES OR L-PATTERN */
RDIVIDESORLPATT(*POS) <- RDIVIDES(*POS).
RDIVIDESORLPATT(*POS) <- LPATT(*POS).

/* OUR KING (TO) OUR ROOK NEW DISTANCE LESS (OR) EQUAL */
OKORNDLE(*:*TK1:*OK1:*OR1,*:*TK2:*OK2:*OR2)
        <- DIST(*OK1,*OR1,*DIST1) & DIST(*OK2,*OR2,*DIST2) &
           LE(*DIST2,*DIST1).

/* ROOM GREATER THAN 2 OR NOT OUR KING (ON) EDGE */
ROOMGT2ORNOTOKEDGE(*POS) <- ROOMGT2(*POS).
ROOMGT2ORNOTOKEDGE(*POS) <- ~OKEDGE(*POS).

STALEMATE(*POS) <- ~INCHECK(*POS) & ~TMOVE(*POS,*).

ROOKLOST(THEIRTURN:*TK:*OK:*OR) <- KINGRULE(*TK,*OR) & ~KINGRULE(*OK,*OR).
```

Fig. 6.4

```
/* PREDICATES AUXILIARY TO ADVICE TABLE PREDICATES */

INCHECK(*:*TK:*OK:*OR) <- KINGRULE(*OK,*TK) & /.
INCHECK(*:*TK:*OK:*OR) <- ~(*OR=*TK) & ROOKCHECK(*:*TK:*OK:*OR).

ROOKCHECK(*:*ITK.*JTK:*IOK.*JOK:*ITK.*JOR) <- ~(*IOK=*ITK) & /.
ROOKCHECK(*:*ITK.*JTK:*IOK.*JOK:*ITK.*JOR) <- ~BETWEEN(*JOK,*JTK,*JOR) & /.
ROOKCHECK(*:*ITK.*JTK:*IOK.*JOK:*IOR.*JTK) <- ~(*JOK=*JTK) & /.
ROOKCHECK(*:*ITK.*JTK:*IOK.*JOK:*IOR.*JTK) <- ~BETWEEN(*IOK,*ITK,*IOR).

BETWEEN(*X,*Y,*Z)
       <- (LT(*Y,*X) & LT(*X,*Z)) | (LT(*Z,*X) & LT(*X,*Y)).

RDIVIDESORLPATT(*POS) <- RDIVIDES(*POS) | LPATT(*POS).

/* L-PATTERN */
LPATT(THEIRTURN:*ITK.*JTK:*IOK.*JTK:*IOK.*JOR)
     <- DIFF(*IOK,*ITK,*IDIFF) & ABS(*IDIFF,2) &
        (SUCC(*JTK,*JOR) | PRED(*JTK,*JOR)).
LPATT(THEIRTURN:*ITK.*JTK:*ITK.*JOK:*IOR.*JOK)
     <- DIFF(*JOK,*JTK,*JDIFF) & ABS(*JDIFF,2) &
        (SUCC(*ITK,*IOR) | PRED(*ITK,*IOR)).

ROOM(*:*ITK.*JTK:*OK:*IOR.*JOR,*R)
    <- SIDE(*ITK,*IOR,*I) & SIDE(*JTK,*JOR,*J) &
       PROD(*I,*J,*R) & /.
ROOM(*:*:*:*,100).

SIDE(*X,*Y,*Z) <- LT(*X,*Y) & DIFF(*Y,1,*Z) & /.
SIDE(*X,*Y,*Z) <- GT(*X,*Y) & DIFF(8,*Y,*Z).

/* MANHATTAN DISTANCE */
MDIST(*I1.*J1,*I2.*J2,*D) <- DIFF(*I2,*I1,*I21) & ABS(*I21,*ABSI) &
                             DIFF(*J2,*J1,*J21) & ABS(*J21,*ABSJ) &
                             SUM(*ABSI,*ABSJ,*D).

/* ANOTHER KIND OF DISTANCE */
DIST(*I1.*J1,*I2.*J2,*D) <- DIFF(*I2,*I1,*I21) & ABS(*I21,*ABSI) &
                            DIFF(*J2,*J1,*J21) & ABS(*J21,*ABSJ) &
                            MINMAX(*ABSI,*ABSJ,*,*D).

ROOMGT2(*POS) <- ROOM(*POS,*AREA) & GT(*AREA,2).

OKEDGE(*:*:*OK:*) <- EDGE(*OK).

EDGE(1.*).  EDGE(8.*).  EDGE(*.8).  EDGE(*.1).
```

Fig. 6.5

```
/* RULES FOR MOVING THE PIECES */

KINGRULE(*I1.*J1,*I2.*J2) <- (SUCC(*I1,*I2) & SUCC(*J1,*J2)) |
                             (SUCC(*I1,*I2) & PRED(*J1,*J2)) |
                             (PRED(*I1,*I2) & SUCC(*J1,*J2)) |
                             (PRED(*I1,*I2) & PRED(*J1,*J2)) .
KINGRULE(*I1.*J1,*I1.*J2) <- (SUCC(*J1,*J2) | PRED(*J1,*J2)) .
KINGRULE(*I1.*J1,*I2.*J1) <- (SUCC(*I1,*I2) | PRED(*I1,*I2)) .

ROOKRULE(*OR1,*OR2,*N.*S.*E.*W)
        <- ROOKNORTH(*OR1,*OR2,*N) | ROOKSOUTH(*OR1,*OR2,*S) |
           ROOKEAST (*OR1,*OR2,*E) | ROOKWEST (*OR1,*OR2,*W) .
ROOKNORTH(*I1.*J1,*I1.*J2,*N)
         <- LT(*J1,*N) &
            (*J2=*N | PRED(*N,*N1) & ROOKNORTH(*I1.*J1,*I1.*J2,*N1)).
ROOKSOUTH(*I1.*J1,*I1.*J2,*S)
         <- GT(*J1,*S) &
            (*J2=*S | SUCC(*S,*S1) & ROOKSOUTH(*I1.*J1,*I1.*J2,*S1)).
ROOKEAST(*I1.*J1,*I2.*J1,*E)
        <- LT(*I1,*E) &
           (*I2=*E | PRED(*E,*E1) & ROOKEAST(*I1.*J1,*I2.*J1,*E1)).
ROOKWEST(*I1.*J1,*I2.*J1,*W)
        <- GT(*I1,*W) &
           (*I2=*W | SUCC(*W,*W1) & ROOKWEST(*I1.*J1,*I2.*J1,*W1)).

ROOKRANGE(*:*ITK.*JTK:*IOK.*JOK:*,*N.*S.*E.*W)
         <- MINMAX(*ITK,*IOK,*W,*E) & MINMAX(*JTK,*JOK,*S,*N).
```

Fig. 6.6

```
/* MISCELLANEOUS AUXILIARIES */

*X=*X.

SUCC(*X,*Y) <- LT(*X,8) &  SUM(*X,1,*Y).
PRED(*X,*Y) <- GT(*X,1) & DIFF(*X,1,*Y).
MINMAX(*X,*Y,*X,*Y) <- LE(*X,*Y) & /.           MINMAX(*X,*Y,*Y,*X).
ABS(*X,*Y)<- LT(*X,0) & DIFF(0,*X,*Y) & /.      ABS(*X,*X).
SIGN(*X,1) <- GT(*X,0) & /.  SIGN(0,0) <- /.    SIGN(*X,*Y) <- DIFF(0,1,*Y).

TRUE(*GOAL,*POS1,*POS2) <- CONS(*GOAL.*POS1.*POS2.NIL,*ATOM) & *ATOM.
```

Fig. 6.7

7. Concluding remarks

PROLOG programs should be written, in a first approximation at least, as true statements about the relations we want to have computed automatically by computer. This method is in general referred to as "logic programming"; PROLOG is one particular system, which supports the method to a greater degree than any other system. Logic programming has several features that make it an attractive method for knowledge engineering. Its basic unit, the clause, is a production rule, which has emerged, independently of logic programming, as the favorite formalism of knowledge engineering. Moreover, in logic programming the natural way of activating clauses is by pattern matching. Also, clauses are so general that they encompass as a special case the relational data base model [6]. As a result of this, clauses can always be regarded as specifying, explicitly or implicitly, a relational data base. In this way the usual distinction between program and database disappears, which is especially attractive for knowledge engineering.

The PROLOG system is just one experimental realization of logic programming. Its main design consideration was speed. As a spin-off from resolution theorem-proving, logic programming was born under the bane of ineffectiveness; as a result speed was essential for a feasibility demonstration. In fact, PROLOG implementations exist which are approximately as fast as LISP [17].

However, the success of PROLOG should not obscure the fact that it is not the only possible realization of logic programming. We have taken advantage of its existence by performing the experiment reported in this paper. Let us discuss the conclusions with a view toward the design of future realizations of logic programming specifically suited for applications in knowledge engineering.

According to Michie's approach, advice tables are written in the AL/1 language and directly interpreted by a special-purpose system written in the POP-2 language. Given the implementation of AL/1, the remaining work is to construct the advice table and to program in POP-2 the required predicates and the move generator.

In our approach, the interpreter and the advice table are written in the same language. (PROLOG, run on the Waterloo implementation [13].) As programs they are to some extent separated, but not as much as they perhaps should be. Fig. 6.1 and the first half of fig. 6.2 perform some of the functions of the AL/1 interpreter. We should not lose sight of the fact that the advice table used in our example is uncharacteristically simple: in general advice tables have an entry condition (absent in our example) and several rules (what we have called "rules" should be referred as "subrules", as they constitute together a single rule of AL/1). Moreover, advice in AL/1 contains in general several advice tables.

On the other hand, our subrules may have conditions attached to them, which is not the case in AL/1. For example, the condition on the "mate" subrule is essential for avoiding gross wastage of computing effort. Without the condition, a mate would be attempted also in the beginning stage of the game. Because of the depth bound, the "mate" subgame is already the most expensive one. In the beginning of the game, when many rookmoves have to be considered, the game tree is especially large.

In our opinion the endgame program in Waterloo PROLOG [13] is, on the whole, compact and readable. Exceptions are, for example, conjunctions which have to replace arithmetic expressions. Arithmetic expressions do not pose a serious problem in logic programming. Several implementations [10,11,14,16] have in fact adequate support for this feature.

We find that each of the distinctive features of PROLOG systems namely being rule-based, using pattern matching, and using automatic backtracking, have been useful in this application. In connection with backtracking we felt the lack of an additional feature in Waterloo PROLOG. By relying on backtracking we did not have to program a move generator. The rules for the moves are merely specified and the backtracking mechanism generates a new move whenever one is needed. Sometimes, however, as when a forcing tree is required, we need all such generated moves simultaneously in a list. In most PROLOGs this effect can be fudged using built-in extralogical features. We preferred to build a game player without forcing trees. IC-Prolog [5], however, has a primitive specifically for this purpose. Our experience suggests that this is quite justified. Kowalski's [9] proposal to merge metalanguage and the object language of logic would also solve the problem encountered here.

Waterloo PROLOG is quite wasteful of storage. For example, in the last game reported below, the stack overflowed after the 28th move. The game was completed by restarting with the last position on an empty stack. Again the problem is not inherent in logic programming: several implementations [4 ,10,11,16] exist which have quite reasonable storage utilization.

Initial position format: turn:TK:ok:or	Number of moves by US before mate	Average CPU time in seconds per US move on an IBM 4331
theirturn:4.2:8.2:5.5	9	.48
theirturn:5.4:8.6:4.2	24	.80
theirturn:7.6:1.2:5.4	29	.72

Table 7.1: Data for three randomly selected them-to-move initial positions

To summarize: Waterloo PROLOG is probably a good (compared to other implemented alternatives) tool for this type of application, in fact surprisingly good for an early, experimental realization of logic programming. The three faults noted (wasteful storage management, lack of "set of solutions", lack of arithmetic expressions) have all been corrected in several other versions of PROLOG.

8. Acknowledgements

I am grateful to Donald Michie for introducing me to advice tables in an early stage of their development. It would have been impossible to develop this program without the use of the excellent PROLOG system implemented by Grant Roberts. My claim to the simplicity and naturalness of the endgame program is in spite of the fact that this is the third attempt, which was facilitated by version II, programmed by Helio Silva, which, in turn, was facilitated by version I, programmed by Nancy Meisner.

The National Science and Engineering Research Council financed supporting facilities, which I hereby gratefully acknowledge.

9. References

[1] Claude Berge, The Theory of Graphs and its Applications; Methuen, London and New York, 1962.

[2] I. Bratko, Proving correctness of strategies in the AL1 assertional language; Information Processing Letters 7 (1978), pp. 223-230.

[3] I. Bratko, D. Kopec, D. Michie, Pattern-based representation of chess end-game knowledge; Computer Journal, 21 (1978), 149-153.

[4] M. Bruynooghe, The memory management of PROLOG implementations; Logic Programming, K.L. Clark and S.A. Tarnlund (eds.), (to appear).

[5] K.L. Clark and F.G. McCabe, The Control Facilities of IC-PROLOG; Expert Systems in the Micro-electronic Age, Donald Michie (ed.), Edinburgh University Press, 1979.

[6] M.H. van Emden, Computation and Deductive Information Retrieval; E. Neuhold (ed.): Formal Description of Programming Concepts, North Holland, Amsterdam, 1978.

[7] Reuben Fine, Basic Chess Endings; David MacKay Company, New York, 1949, 1969.

[8] R.A. Kowalski, Predicate Logic as a programming language; Proc. IFIP 74, North Holland, 1974, 556-574.

[9] R.A. Kowalski, Logic for Problem-Solving; North Holland - Elsevier, 1979.

[10] F.G. McCabe, Tiny Prolog; Logic Programming, K.L. Clark and S.A. Tarnlund (eds.), to appear.

[11] C.S. Mellish, An alternative to structure-sharing in the implementation of a PROLOG interpreter; Logic Programming, K.L. Clark and S.A. Tärnlund (eds.), to appear.

[12] D. Michie, King and Rook against King: in M.R.B. Clarke (ed.), Advances in Computer Chess I; Edinburgh University Press, 1977.

[13] G.M. Roberts: An implementation of PROLOG; M.Sc. Thesis, Dept. of Computer Science, University of Waterloo, 1977.

[14] J.A. Robinson and E.E. Sibert, Logic Programming in Lisp; Department of Computer and Information Science, Syracuse University.

9. References - cont'd.

[15] Claude E. Shannon, Programming a computer for playing chess; Phil. Mag. 41 (1950), 256-275.

[16] D. Warren, An improved PROLOG implementation which optimizes tail recursion; Logic Programming, K.L. Clark and S.A. Tarnlund (eds.), to appear.

[17] D.H.D. Warren, L.M. Pereira, F. Pereira, PROLOG - the language and its implementation compared with LISP; SIGART/SIGPLAN Symposium on AI and Programming Language, U. of Rochester, August 1977. SIGPLAN Notices 12 (1977), 109-115, SIGART Newsletter 64, (1977), 109-115.

[18] C. Zuidema, Chess; how to program the exceptions? Report IW 21/76, Mathematical Centre, Amsterdam, 1974.