AN ABSTRACT OF THE THESIS OF

Nicholas S. Flann for the degree of Master of Science in Computer Science presented on May 1, 1986.

Title: Learning Functional Descriptions From Examples

Redacted for Privacy

Abstract approved: ______________________

Thomas G. Dietterich.

The task of inductive learning from examples places constraints on the representation of training instances and concepts. These constraints are different from, and often incompatible with, the constraints placed on the representation by the performance task. This incompatibility is severe when learning functional concepts and explains why previous researchers have found it so difficult to construct good representations for inductive learning—they were trying to achieve a compromise between these two sets of constraints. This thesis addresses this problem, and takes a different approach. Rather than designing a compromise representation we employ two different representations: one for learning and one for performance.

The system developed learns concepts in chess and checkers. Training instances are presented in the "performance representation" as simple board positions, then converted to the "learning representation" via a search process that builds an explanation of the outcome of the position. Inductive generalization is performed over these explanations to form descriptions of the concepts in terms of the moves and goals involved. Finally the concepts are translated back into the "performance representation" to support efficient recognition of future instances.

The advantages of this "two representation" approach are (a) many fewer training instances are required to learn the concept, (b) the biases of the learning program are very simple, and (c) the learning system requires virtually no "vocabulary engineering" to learn concepts in a new domain.

Learning Functional Descriptions From Examples

By N. S. Flann

A Thesis submited to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Master of Science

Completed May 12, 1986

Commencement June 1986.

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented May 1, 1986

ACKNOWLEDGEMENTS

I am indebted to Tom Dietterich for his help and guidance throughout this work. I have benefited greatly from his rigorous approach and clear understanding of the issues.

I am grateful to Colin Gerety for many interesting discussions and critical comments during the development of Wyl. He also provided valuable feedback on early drafts of this thesis. I appreciate the careful reading of the thesis and useful comments by Dan Corpron.

I thank my friends for their support and forbearance, especially Kevin Krefft, Mike Gross and Rob Cline.

I take this opportunity to thank my parents, Jean and John Flann, for supporting and encouraging me throughout my education.

This work was partially supported by Tektronix Contract No. 530097 and by the National Science Foundation under grant numbers IST-8519926 and DMC-8514949.

Table of Contents

1 Introduction **1**

1.1 Inductive learning and the representation of concepts 3

1.2 Choosing the most appropriate representation for learning functional concepts 7

1.3 An overview of Wyl 9

1.4 A guide to the Thesis 11

2 Representation and Process in Wyl **13**

2.1 An Overview of Wyl 13

2.2 Representation of Knowledge 16

2.2.1 Structural Knowledge 17

2.2.2 Functional Knowledge 23

2.3 Process in Wyl 35

2.3.1 Learning the Functional Concepts 35

2.3.2 Applying the Functional Concepts 50

2.3.3 Summary of Wyl's Interpreters 64

2.3.4 Compaction 65

2.4 Chapter Summary 82

3 Examples of Wyl Learning **83**

3.1 Lesson 1: Checkers concept *trap* 84

3.2 Lesson 2: Checkers concept *trap-in-2-ply* 89

3.2.1 A dialogue with Wyl 93

3.3 Lesson 3: chess concept *knight-fork* 99

3.4 Lesson 4: chess concept *Skewer* 104

4 Analysis and Conclusions **108**

4.1 Summary 108

4.1.1 The Effectiveness of Wyl 109

4.1.2 Why Wyl works 111

4.2 Future research 113

4.2.1 Extensions to Wyl 113

4.2.2 General Issues 115

4.2.3 Directions in the underlying architecture 118

4.3 Contributions of the Thesis 118

5 References **119**

A Chess Domain Specification **124**

A.1 Search schema 124

A.2 Actions and Goals 127

A.3 Structural Language 129

B Checkers concept trap structural concept langauge **131**

B.1 Relational terms 131

B.2 Descriptive terms 133

C Trap structural concept description **136**

List of Figures

1.1 An example of the Checkers concept *Trap* 8

1.2 Representations in Wyl 10

2.1 Checker board numbering scheme 18

2.2 Example of *trap*, red to play 20

2.3 Trap position 2, red to play 30

2.4 Preference of recognizable states for checkers 36

2.5 Functional instance of *trap* position in Figure 2.3 42

2.6 Generalized functional instance of *trap* position Figure 2.3 45

2.7 Second training example for concept *trap*. 46

2.8 Generalized functional instance of *trap* position Figure 2.7 48

2.9 Generalized functional Concept of *trap* 49

2.10 *Trap* position (state1) with red to play 52

2.11 Reasoning tree formed during generation of *trap* in Figure 2.3 58

2.12 Checker board numbering scheme 72

2.13 Tax identifies first *focus* set 75

2.14 Tax finds second a new term, over the relative-direction attribute . . 76

2.15 Tax, ready to repartition to find trapped-square terms 79

3.1 Structural concept terms for checkers *trap* 89

3.2 Training Instance 1: Checkers concept *trap-in-2-ply* 90

3.3 Training Instance 2: Checkers concept *trap-in-2-ply* 90

3.4 Test Instance 1: Checkers concept *trap-in-2-ply* 92
3.5 Test Instance 2: Checkers concept *trap-in-2-ply* 92
3.6 Functional Concept *trap-in-2-ply* 100
3.7 Training Instance 1: Chess concept *knight-fork* 100
3.8 Training Instance 2: Chess concept *knight-fork* 101
3.9 Test Instance 1: Chess concept *knight-fork* 102
3.10 Test Instance 2: Chess concept *knight-fork* 103
3.11 Test Instance 3: Chess concept *knight-fork* 104
3.12 Training Instance 1: Chess concept *skewer* 104
3.13 Training Instance 2: Chess concept *skewer* 105
3.14 Test Instance 1: Chess concept *skewer* 106
3.15 Test Instance 2: Chess concept *skewer* 106
3.16 Test Instance 3: Chess concept *skewer* 107

A.1 Chess board notation 125

Chapter 1

Introduction

Concept learning from examples has been studied for over 15 years within the AI community and is perhaps the best understood facet of machine learning. Systems that learn from examples consist of two parts: a learning component that takes the training instances and forms a general concept description, and a performance component that applies the newly learned descriptions to some task, usually recognition. One of the earliest such systems was Winston's (1975) ARCH program. Here, descriptions of arches are presented to the system in a simple language of observable structures and features such as *block*, *touching*, *standing*, *on-top-of* etc. The learning component constructs a general concept description written in the same structural language by comparing the current concept description with the new instance and letting any differences drive generalization and specialization. The performance component applies this description of an arch to classify future instances by a matching process similar to that used for learning.

In Winston's work, the structural representation of the concept was suitable for both the learning and recognition tasks. Its suitability for the learning task is evident because relatively few training instances were required to learn the concept and the concept language was easy to design. The representation is suitable for recognition because instances were classified quickly via a simple matching opera-

tion.

Many other successful learning systems employ this approach of using only one representation to capture the concepts of interest for both the learning and performance tasks. Michalski (1980) has applied the A^q algorithm to numerous learning tasks including soybean disease classification. In this domain, the single structural representation of observable leaf features, such as *black spots, curled end* etc. was employed to describe the recognition rules learned from examples.

However, when Quinlan (1982) attempted to pursue this approach in his work on learning chess end-game concepts, he encountered difficulties. His representation for high-level chess features was effective for the task of recognizing end-game positions, but it introduced many problems for the learning task. First, the concept language was very difficult to design. Quinlan spent two man-months iteratively designing and testing the language until it was satisfactory. The second problem was that it took a large number of training instances (334) to learn the concept of *lost-in-3-ply* completely. These problems illustrate that the approach of employing the same representation for learning and for performance was inappropriate for this domain.

In this thesis, we show that inductive learning places constraints on the representation for training instances and concepts and that these constraints can conflict with the requirements of the performance task. Hence, the difficulty that Quinlan encountered can be traced to the fact that the concept *lost-in-3-ply* is an inherently *functional* concept that is most easily represented and therefore learned in a functional representation. However, the performance task (recognition) requires a structural concept representation. The vocabulary that Quinlan painstakingly constructed was a compromise between these functional and structural representations.

In this thesis, we propose an alternative approach to learning functional concepts that avoids the problems Quinlan encountered. Rather than employ a single representation and design a concept language that can only partially satisfy the

two conflicting constraints, we use two distinct representations, one for learning, one for performance. The learning representation satisfies the constraints of inductive learning, while the performance representation satisfies the constraints of recognition. By using two representations, the learning system becomes effective at both learning and recognition: few examples are needed to learn the concepts, the concept language is easy to design, and finally, recognition is fast and simple.

The remainder of this chapter is organized as follows. First, we describe the constraints that the task of inductive learning places on the representation of concepts and instances. Second, we relate these constraints and the form of the concept (functional or structural) to choosing the most appropriate representation. Third, we present an overview of an implemented system, Wyl, that learns functional concepts in checkers and chess from structural training instances by first mapping them into a functional representation, generalizing them there, and converting the learned concepts back into a structural representation for efficient recognition. Finally, we give a guide to the rest of the thesis.

1.1 Inductive learning and the representation of concepts

The goal of an inductive learning program is to produce a correct definition of a concept after observing a relatively small number of positive (and negative) training instances. Gold (1967) cast this problem in terms of search. The learning program is searching some space of concept definitions under guidance from the training instances. He showed that (for most interesting cases) this search cannot produce a unique answer, even with denumerably many training instances, unless some other criterion, or bias, is applied. Horning (1969), and many others since, have formulated this task as an optimization problem. The learning program is given a preference function that states which concept definitions are a priori more likely to be correct. The task of the learning program is to maximize this likelihood subject

to consistency with the training instances.

This highly abstract view of learning tells us that inductive learning will be easiest when (a) the search space of possible concept definitions is small, (b) it is easy to check whether a concept definition is consistent with a training instance, and (c) the preference function or bias is easy to implement. In practice, researchers in machine learning have achieved these three properties by (a) restricting the concept description language to contain few (or no) disjunctions, (b) employing a representation for concepts that permits consistency checking by direct matching to the training instances, and (c) implementing the bias in terms of constraints on the syntactic form of the concept description.

Let us explore each of these decisions in detail, since they place strong constraints on the choice of good representations for inductive learning.

Consider first the restriction that the concept description language must contain little or no disjunction. This constraint helps keep the space of possible concept definitions small. It can be summarized as saying "Choose a representation in which the desired concept can be captured succinctly."

The second decision—to use matching to determine whether a concept definition is consistent with a training instance—places constraints on the representation of training instances. Training instances must have the same syntactic form as the concept definition. Furthermore, since the concept definition contains little or no disjunction, the positive training instances must all be very similar syntactically. To see why this is so, consider the situation that would arise if the concept definition were highly disjunctive. Each disjunct could correspond to a separate "cluster" of positive training instances. With disjunction severely limited, however, the positive training instances must form only a small number of clusters.

In addition to grouping the positive instances "near" one another, the representation must also allow them to be easily distinguished from the negative instances.

This is again a consequence of the desire to keep the concept definition simple. The concept definition can be viewed as providing the minimum information necessary to determine whether a training instance is a positive or a negative instance. Hence, if the concept definition is to be short and succinct, the syntactic differences between positive and negative instances must be clear and simple.

The third decision—to implement bias in terms of constraints on the syntactic form of the concept description—makes the choice of concept representation even more critical. Recall that the function of bias is to select the correct, or at least the most plausible, concept description from among all of the concept descriptions consistent with the training instances. Typically, the bias is implemented as some fixed policy in the program, such as "prefer conjunctive descriptions" or "prefer descriptions with fewest disjuncts." The bias will only have its intended effect if conjunctive descriptions or descriptions with fewest disjuncts are in fact more plausible. In other words, for syntactic biases to be effective, the concept description language must be chosen to make them true. The net effect of this is to reinforce the first representational constraint: the concept representation language should capture the desired concept as succinctly as possible.

Now that we have reviewed the constraints that inductive learning places on the representation of training instances and concepts, we can explain why some machine learning systems have been more successful than others. Consider Winston's ARCH program. In his structural representation, the positive examples of arches are all very similar (three objects that are restricted in shape and arrangement). Negative examples—non-arches—are all easily distinguished by some simple observable features such as *touching* or *standing*. This explains why the ARCH system was so successful.

In contrast, consider Quinlan's work on the chess concept *lost-in-3-ply*. When positive and negative examples of *lost-in-3-ply* are represented as simple board

positions, there are no obvious distinguishing features. Moving a piece one square often changes the classification of an instance. When we consider that there are 1.8 million distinct training instances, it is clear that inductive learning in this low level representation would require that the concept description include vast numbers of disjuncts. It is not surprising that Quinlan chose to design a higher-level vocabulary for describing his training instances.

In his (1982) article, Quinlan showed how one could evaluate the correctness of an inductive learning program by asking how many training examples are required for the program to discover a concept of a given complexity (i.e., with a given number of disjuncts). His definition of a perfect learning program was one that required only one positive training example for each disjunct in the concept definition. Such a learning program would possess a perfect bias.

We can turn this analysis around and use it to evaluate the combined appropriateness of the bias and the representation language. If a program requires few training instances to discover a concept, then the combination of the bias and representation is working well to select the proper concept definition.

By this measure, the ARCH program has an excellent bias and representation language, since very few training instances are required. On the other hand, even after Quinlan carefully engineered the representation language so that it included high level structural and functional terms, his system required 334 training instances to learn the concept *lost-in-3-ply*. This indicates that the representation language and the bias were still not very appropriate for learning this concept.

We can summarize this section by stating the following constraints on the choice of representation languages for inductive learning. First, the language should be able to represent the desired concept succinctly (i.e., conjunctively or as a short disjunction). Second, the training instances should have the same form as the concept definition. Third, the representation should capture semantic similarities among

the positive training instances directly in syntactic forms. Fourth, the representation should capture semantic differences between the positive and negative training instances syntactically.

1.2 Choosing the most appropriate representation for learning functional concepts

Now that we have reviewed the constraints that inductive learning places on the representation, we must consider how to satisfy those constraints in a given learning task. In particular we look at the task of learning end game concepts in chess and checkers. It should be clear that we want to select the representation that captures the concept most "naturally."

The "natural" representation is the one that formalizes the underlying reason for treating a collection of entities as a concept in the first place. A concept (in the machine learning sense anyway) is a collection of entities that share something in common. Some entities are grouped together because of the way they appear (e.g., arches, mountains, lakes), the way they behave (e.g., mobs, avalanches, rivers), or the functions that they serve (e.g., vehicles, cups, doors). Occasionally, these categories correspond nicely. Arches have a common appearance and a common function (e.g., as doorways or supports). More often, though, entities similar in one way (e.g., function) are quite different in another (e.g., structure).

The performance task for which a concept definition is to be learned may require a structural representation (e.g., for efficient recognition), a functional representation (e.g., for planning), or a behavior representation (e.g., for simulation or prediction). When we review the successes and failures of machine learning, we see that difficulties arise when the representation required for the performance task is not the natural representation for the concept.

Winston's *ARCH* program was successful because the natural representation—

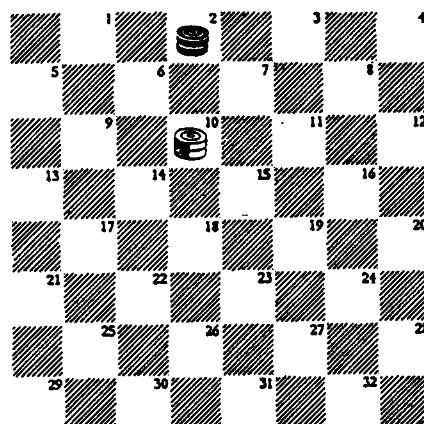


Figure 1.1: An example of the Checkers concept *Trap*

structural—was also the performance representation. Quinlan's difficulties with *lost-in-3-ply* can be traced to the fact that this concept is naturally defined functionally, yet the performance task required a structural representation. All board positions that are *lost-in-3-ply* are the same, not because they have the same appearance, but because they all result in a loss in exactly 3 moves. The representation in which *lost-in-3-ply* can be captured naturally (and hence the one which is most appropriate for learning), is a functional representation that includes operators (such as *move*) and goals (such as *loss*).

This functional representation of operators and goals is the most appropriate for end game concepts in general. End game concepts describe particular ways in which goals of the game are achieved. *Lost-in-3-ply* describes board positions which in which the goal *loss* is achieved following 3 legal moves. Consider the checkers end game concept *trap* illustrated in Figure 1.1. In this example the red king in square 2 is trapped by the white king in square 10. No matter what move the red king makes, the white king can take him. *Trap* is like *lost-in-3-ply* in that it describes a *loss* in a fixed number of moves (ie. *lost-in-2-ply*), yet it is different because it includes additional constraints on the operators. All *lost-in-2-ply* positions are not

traps. Only if the operators are of particular types (the first *normal-moves*, the second *take-moves*) and interact in certain ways (the second move jumps over the destination square of the first move), can the position be considered a *trap*. Hence, the functional representation must not only be capable of describing concepts as operator sequences and goals, but it must be able to capture both particular kinds of operators and interelations between them.

Now that we have presented a justification of the use of a rich yet simple functional language to learn end game concepts, we give a brief overview of the implemented system that exploits such a representation.

1.3 An overview of Wyl

We have developed a learning system named Wyl [1] that applies two representations, one for learning and one for performance, to learn concepts in board games such as checkers and chess. We have chosen this domain because there are simple and complete "domain theories" available (the rules of the games) and there are many interesting concepts that are naturally functional (e.g., *trap*, *skewer*, *fork*, *lost-in-2-ply*) and yet have complex structural definitions. Wyl has been applied to learn definitions for *trap* and *1-move-to-trap* in checkers and *skewer* and *knight-fork* in chess.

The performance task of Wyl is recognition. Given a board position, represented simply in terms of the kinds and locations of the playing pieces, Wyl must decide whether that position is, for example, a *trap*. To perform this task, the *trap* concept must be represented in a structural vocabulary that permits efficient matching against the board positions. However, as we have noted above, concepts such as *trap* are most easily learned in a functional representation.

In addition to requiring a structural representation for performance, a structural

[1] Named after James Wyllie, checker champion of the world from 1847 to 1878.

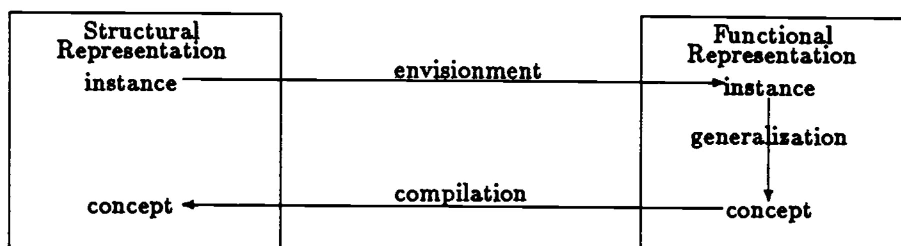


Figure 1.2: Representations in Wyl

representation is also needed for the training instances. To teach Wyl checkers and chess concepts, we want to simply present board positions that are examples of those concepts.

The organization of Wyl is shown in Figure 1.2. Wyl learns from positive instances only. These training instances are board positions, represented in an instance language of simple structural features—namely, the kinds and locations of the playing pieces.

Wyl converts these training instances into a functional representation through a process of envisionment. The purpose of envisionment is to determine how a given board position relates to the known goals of the players (e.g., *loss* or *win*). Wyl knows the rules of checkers (and chess), so it is able to conduct a forward minimax search to see what outcomes the given board position might lead to. Once it has related the board position to some known goal, it constructs an AND/OR graph that explains the relationship (along the lines of Mitchell, et al.,1986). We call this AND/OR graph a *functional training instance.* Wyl employs functional training instances to conduct inductive inference in the functional representation. This results in a functional concept definition that captures the desired concept.

The final task is to convert this functional definition into an equivalent structural description that can support efficient recognition. This is accomplished through a

compilation process that generates, as a side-effect, a generalized structural vocabulary for representing the concept.

The initial knowledge given to Wyl takes four forms. First, there is the structural instance langauge for board positions. Second, there is a representation for each of the legal operators in the game (e.g., *normal-move* and *take-move*). Third, Wyl is given the rules of the game, represented as a recursive schema that describes what moves are legal at what points in the game. Finally, Wyl is given definitions of the important goals of the game, such as *loss*, *win*, and *draw*. For chess, Wyl is also told that *lose-queen* is an important goal.

These given goals are the key to Wyl's learning ability. Wyl learns new functional concepts as *specializations* of these known concepts. For example, we know that the checkers concept *trap* is a specialization of *loss*. Once Wyl learns a recognition predicate for *trap*, it is added to the pool of known concepts, where it may be specialized further to form some future concept. Hence Wyl learns new concepts that fit into a generalization/specialization hierarchy under the presupplied goals.

This completes our overview of the Wyl system and the information that it is initially given. Chapter 2 gives a detailed description of Wyl.

1.4 A guide to the Thesis

In Chapter 2, we describe the control and representation used in the program Wyl. First both the initial structural and functional languages are described and illustrated with examples of representations of board positions and concepts. Next each method involved in the learning process is presented to a detail sufficient for others to reproduce the work.

In Chapter 3, we present traces of Wyl learning four concepts. First we present Wyl learning in checkers with *trap* and *trap-in-2-ply*. Then we present Wyl learning in chess with *skewer* and *knight-fork*. With each learning session we give the

training instances, test instances, the dialogue with Wyl and both the machine representation and first order logic description of the learned concept.

Finally in Chapter 4, we present the summary and conclusions of this thesis.

Chapter 2

Representation and Process in Wyl

In this chapter we present Wyl, a concept learning program working in the domains of chess and checkers. The chapter begins with an overview of Wyl that describes the main learning loop and the knowledge transformation methods employed at each stage. Next a comprehensive description of the representations of both the structural and functional spaces is given. There follows a description of the methods that interpret and transform the knowledge. Finally the process that constructs new structural descriptions representing the functional concepts learned, is given.

Wyl has successfully learned concepts in both chess and checkers. In this chapter, to aid in the explanation of Wyl, we use examples from checkers; in particular, we illustrate the representations and operation of Wyl with the checkers concept *trap*. In Chapter 3 we give traces of Wyl learning an additional concept in checkers and two concepts in chess along with representations of the training instances and concepts learned in functional and structural form.

2.1 An Overview of Wyl

The main learning loop for Wyl is given below:

```
(setq domain (get-domain?))
```

```
(while (continue-with-this-domain?)
        (setq conceptname (get-concept?))
        (setq conceptdefinition (lookup-concept-def conceptname))
        (while (continue-with-concept?)
                do (setq instance (get-instance?))
                   (if (test-concept conceptdefinition instance)
                      (if (not (example-of-concept?))
                         (ERROR))
                      (if (example-of-concept?)
                         (setq conceptdefinition (generalize (envisionment instance) concept))))
        (if (translate-to-structural?)
           (compact (generate-instances conceptdefinition)))
```

Wyl first takes a domain and enters the main concept learning loop. The main concept learning loop gets a name of a concept from the user (get-concept?), looks up the definition of the concept (conceptdefinition) and enters the training instance loop. The concept definition may be nil (new concept), a functional definition (continuing with a training session) or a structural definition (previously learned concept). Within this training loop, Wyl first gets a new training instance from the user (get-instance?), then tests it against the current concept definition. The result of the test (either *pass* or *fail*) is presented to the user who is asked for the correct classification (example-of-concept?). If the current concept classifies the instance correctly, no action is taken. If, on the other hand, the current concept definition is incorrect, two actions can be performed depending upon the kind of classification error. If the current concept requires generalization, (i.e. Wyl said *fail*, the user said *pass*) the structural training instance is first transformed into a functional instance by the envisionment stage. Next Wyl forms the maximally specific generalization of the current functional concept definition and the new in-

stance. If the current concept is too general, that is, it incorrectly classifies the instance positive, Wyl cannot do anything, since the learning component of Wyl is similar Vere's (1975) method of forming maximal unifying generalizations and performs no specialization.

The compilation stage is performed after all learning has taken place and the user decides that the current inductively learned concept is "correct" (translate-to-structural?). Forming the structural concept representation involves two stages. First the concept description is interpreted as a generator and employed to form the full set of positive structural instances by the function generate-instances. Second the set of structural instances is translated into a more compact form through the function compact. The function compact creates a new structual lanngauge in which to describe the original set of instances.

The various methods and processes in Wyl can be divided into three kinds; those that form the functional concept definition during learning, those that apply the functional concepts to the performance task, and finally the compaction stage that transforms expressions in the structural space.

The methods employed to form the functional concepts are:

Envisionment: Takes a structural training instance (board position) and performs min/max search to form the functional training instance.

Generalization: Takes the functional instance and the current concept definition (possibly nil) and produces a more general concept definition that covers the new instance.

The methods employed to apply the functional concepts are:

Test: The functional concept definition is employed as a test of structural instances.

Generator: The functional definition is employed as a generator of all the structural instances that satisfy the definition.

Wyl is implemented as an interpreter in LISP operating on top of MRS, the logic programming language. MRS is semantically and syntactically similar to first order logic. Syntactically, MRS uses prefix notation and denotes universally quantified variables as atoms beginning with $. The most important semantic departure from first order logic is the implicit closed database assumption, exploited to solve a variety of representation and reasoning problems including the frame problem (Reiter 1980). In the writing below we give brief descriptions, where needed, of the primitives and representations of MRS. The reader should refer to Russell (1985) for more information.

2.2 Representation of Knowledge

In this section we describe the functional and structural languages used in Wyl. Wyl is domain independent, in the sense that general structural and functional languages are supplied in which the domains of chess and checkers are specified. A domain is specified by the following;

Structural Language: A language describing the board and playing pieces written in MRS.

Functional Language: A language describing the moves, goals and search schema:

- **Actions** The legal moves of the game specified as rules in MRS (which is extended to deal with state variables).
- **Goals** The definition of a *win*, *loss*, and any other goals of interest in MRS.
- **Search Schema** This specifies to Wyl, how to used the move and goal defintions during search (see below).

First we discuss the representation of structural knowledge and functional knowledge. In both discussions we give examples of instances and concepts. Following

this discussion of representation we turn to the three processes of Wyl. First, how Wyl learns concepts. Second, how Wyl applies the functional concepts as both tests and generators. Third, how Wyl constructs the new compact structural langauge.

2.2.1 Structural Knowledge

One of the goals of this thesis is to demonstrate a learning system which does not require a highly engineered structural language. The initial structural language is very simple, one which could be formed by a vision system viewing a checker or chess board. Descriptions are formed by conjunctions of directly observable features, such as the color or type of a playing piece. This language can only describe structural *instances*, that is, board positions. Initially, the only way to describe a concept in structural terms is as a disjunction of all the positive instances. Later, after a functional concept definition is learned, Wyl will use it to construct a better structural language for representing *concepts*. This improved structural language includes new structural terms that are defined from the original structural primitives. We illustrate both the initial instance language and the constructed concept language with examples of the checkers concept *trap*.

The structural instance language

The instance language consists of descriptive and relational terms, were each term represents an observable feature and has a fixed, bounded, number of values. The ontology of the observable features needs to represent the relationship between playing squares on the board and the contents of squares (whether empty or occupied by a playing piece). In Figure 2.1 we show the standard numbering scheme for checker boards.

A board is described by a set of relational terms defining the relationship between playing squares. In checkers this takes the form of 98 facts asserting all the directions

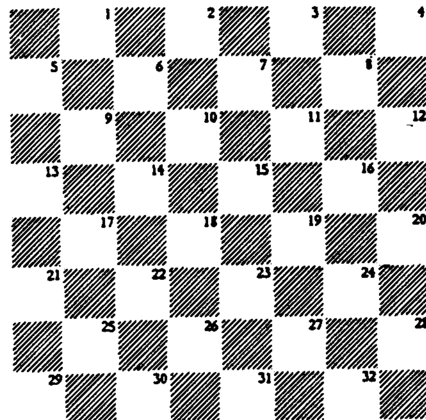


Figure 2.1: Checker board numbering scheme

in which the playing squares are connected. Below we give a sample of the facts:

```
(connected s1 s5 sw)
(connected s5 s1 ne)
(connected s1 s6 se)
(connceted s6 s1 nw)
```

The first fact states that square s1 is connected to square s5 in the sw direction. The second fact states the reverse relationship(s1 is ne of s5), and so on.

Each playing piece is a unique object with two features, the side, which can have the value red or white, and the type, which can have the value king or man. For example the red king is described as

```
(type rk1 king)
(side rk1 red).
```

The facts that refer to the contents of squares are time dependent; their truth changes nonmonotonically during state search. The analytic frame problem is solved by a situation calculus scheme (McCarthy 1958) and the archeological model (Waldinger 1977). Each occupied fact has a state variable, for example:

(occupied state-0 s1 empty)
(occupied state-0 s2 empty).

The empty board is state-0, represented as 32 occupied facts asserting the playing squares are empty. Original states created by Wyl, such as training instances, are described as changes to state-0. For example, a positive training instance of the concept *trap* is given in Figure 2.2. To represent this board position as state-1, the following facts are asserted:

(occupied state-1 s15 rm1)
(occupied state-1 s23 wm1)
(nextstate state-0 state-1).

New states created by Wyl during state search are described as changes to current state. Each applicable operator in state state-m, creates a new state variable state-n, asserts the new occupied facts in state-n, and extends the search tree appropriately by asserting (nextstate state-m state-n). Any unification of occupied facts during interpretation selects the most recent binding by searching back in time via nextstate relationships to state-0. By reifying playing pieces and treating empty as an object, all the forms of queries, including state unbound, are covered.

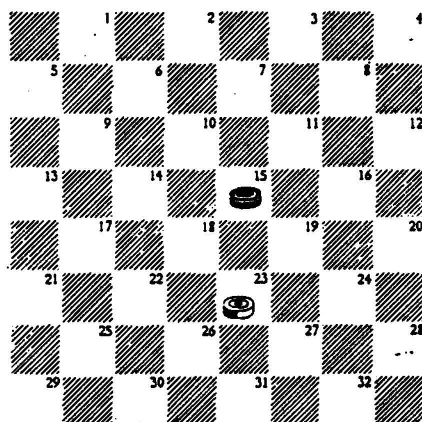


Figure 2.2: Example of *trap*, red to play

The structural concept langauge

The structual instance language initially given to Wyl can describe the complete space of structural instances. Hence, this language can describe *all* concepts extensionally, as sets of instances. Once Wyl has learned a concept and formed a description of it in this instance language, the compaction stage (discussed in Section 2.3.4) creates a new concept language in which the concept can be captured more succinctly.

The concept language is made up of both descriptive and relational terms that are defined from the primitives of the original instance language. The complete concept langauge created for the checkers concept *trap* is given in Appendix B. A concept in this language is represented as a disjunction of conjunctions of the new terms.

The checkers concept *trap*, for example, is captured in a disjunction of 12 such descriptions, each covering a subset of the instances in the original extensional description. All the 12 descriptions are given in Appendix C, here we give one

(simplified) example:

```
(if (and (occupied $state $square-1 $piece1)
         (side $piece1 red)
         (type $piece1 man)
         (sq $square-1 center)
         (occupied $state $square-2 $piece2)
         (side $piece2 white)
         (type $piece2 anytype)
         (connected $square-1 $square-2 south-2-square)
         (new-name $name trap))
   (terminate-state $name trap $state red)).
```

This concept description covers the set of instances of *trap* where a red man is trapped by either a white king or man in the center of the board. The instance of *trap* illustrated in Figure 2.2 is included in this set and hence, is covered by this description.

The predicate says that *state* is a trap for red if red to move and

- There exists a square *square*1 that is occupied by a red man and is located in the center of the board, and
- the square that is two rows directly south of *square*1 is occupied by either a white king or man.

The descriptive terms are either the primitives of the instance language or are named disjunctions of values of the primitives. For example, consider the feature type given previously. The instance language allows pieces to have a type value of man or king. In the examples of trap covered by the description above, the red man is trapped by either a man or a king. It was convenient for Wyl to name this disjunction (man $\vee$ king), anytype. The definition for anytype is given below. We

employ implication to define generality: If X and Y are terms describing the same feature then, $X \supset Y$, states that Y is more general than (or equivalent to) X.

```
(if (or (type $playing-piece king)
        (type $playing-piece man))
    (type $playing-piece anytype))
```

In general, after descriptive term construction, the values of each feature will be arranged in a hierarchy of named disjunctions. Descriptive terms are defined recursively as sets of instance values or terms. The sq primitive encodes the hierarchy of terms describing sets of squares on the board. Initially the sq primitive is defined for all the squares, (sq s1 s1), (sq s2 s2) .... and can be employed as a generator or test of all the squares. For example, the term center used above is defined as a disjunction of 10 squares.

```
(if (or (sq $square s23)
        (sq $square s7)
        (sq $square s11)
        (sq $square s22)
        (sq $square s19)
        (sq $square s14)
        (sq $square s10)
        (sq $square s15)
        (sq $square s18)
        (sq $square s6))
    (sq $square center))
```

Relational concept terms are formed by a conjunction of the relational primitives of the instance language. The terms are similar to macros; they define a path

through the board by a composition of smaller paths. For example the relational term south-2-squares is defined as follows:

```
(if (and (connected $square-1 $square-3 sw)
         (connected $square-3 $square-2 se))
   (connected $square-1 $square-2 south-2-squares))

(if (and (connected $square-1 $square-3 se)
         (connected $square-3 $square-2 sw))
   (connected $square-1 $square-2 south-2-squares))
```

Note that the representations given above can support both forward (data driven) reasoning and backward (goal driven) reasoning.

2.2.2 Functional Knowledge

This section consists of two parts: First, we describe how a *domain* is specified in the functional language using the example of checkers. Second, we explain how *functional concepts* are described in this language using examples from the concept *trap*.

Representation of Domain Knowledge

In order to learn concepts in a domain, Wyl must be provided with a *domain specification* that provides the basic rules of the domain. A domain specification consists of the definition of the goals, actions and search schema of the domain. Each is described below and illustrated with the domain of checkers.

Representation of Search Schema

The search schema represents to Wyl how the forward search is to be performed, when the termination condition is to be checked, and which operators are legal at

each state. The schema is an expression written in a simple programming language. The schema language, being a programming language, has primitives and composition methods for constructing larger expressions out of those primitives. The language is best explained by an example. Below we give the schema for checkers:

```
(o-or (single-terminate terminate)
      (multiple-recursive take-move)
      (multiple-recursive normal-move)).
```

During forward search in checkers, Wyl must first check if the current state is a recognized goal such as *loss* or *trap* etc. This is indicated in the schema by (single-terminate terminate)). If no termination goal is satisfied, Wyl must find the legal moves available. In checkers, one important restriction on moves is that if "take" moves exist they *must* be taken in preference to "normal" moves. Hence, the recursive schema specifies that Wyl first check if there are take moves available (by (multiple-recursive take-move)), if so they are taken, otherwise normal moves are explored (by (multiple-recursive normal-move)).

The schema encodes this search knowledge by directing the behavior of Wyl during interpretation. The schema is interpreted by three different interpreters: envisionment, test, and generation. A detailed description of the interpreters is given in Section 2.3, here we give a brief overview of the commonalities of the interpretation process. Each interpreter performs forward search through the space of board positions. At each point in the search, the schema is interpreted to determine the actions to be taken. Information is carried through the search tree by means of binding lists, which are updated and passed down the recursion. Two important bindings are the current state name, which is held on the variable **$state**, and the current side to move, which is held on the variable **$side**. All the interpreters update these bindings at each recursive invocation. The **$side** binding is automatically

switched between red and white at each depth of the search, while the $state binding is changed to the new state if new operators are applicable.

The language has two primitives, single-terminate and multiple-recursive, and two connectives, o-or and o-and .

First consider the primitives. Each primitive takes one argument, which is the name of a unit (or frame) with predefined slots. One of the slots is named mrsform, and its value is an MRS formula representing a consequent of some rule in the MRS data base. The consequent represents either the goal or the actions in the domain. For example, the frame terminate is the argument of the single-terminate primitive above and specifies a test for termination of search. Hence, the MRS formula stored on the mrsform slot of terminate unifies with the consequences of the rules defining the goals of the domain. We give the frame terminate below:

(terminate mrsform (terminate-state $name $result $state $side))).

Upon interpretation of the (single-terminate terminate) expression, Wyl looks up the MRS formula, plugs any bindings in, including $state and $side, and calls the backward chaining theorem prover of MRS, truep, on the form. For example, if the board position illustrated in Figure 2.2 were analyzed with the binding list (($side . red) ($state . state-1)), the interpreter will call the following:

(truep (terminate-state $name $result state-1 red)).

The resulting binding list will determine whether the current state is a goal state. In this case the current state is an example of a *trap* with red to play, and if Wyl had already learned the concept, then the backward chaining would succeed (on the structural rule given above in the description of the structural concept space), with $result bound to *trap* and $name bound to a unique name.

The single-terminate primitive determines if there exists a condition, defined through the mrsform, for the current state in the search. If so, the recursion is terminated and the new bindings are returned.

The multiple-recursive primitive, on the other hand, determines all the solutions to the supplied MRS formula, and continues the state search for each solution. For example, the value of the mrsform slot of frame normal-move is an MRS formula that unifies with the consequent of the rule defining legal non-take moves in checkers. This frame is illustrated below, (the move rule is given in the subsection on actions later):

(normal-move mrsform (move $name $state $newstate $type $from $to $side)).

The MRS primitive **trueps** is employed on the form after **$state** and **$side** have been plugged in. **Trueps** returns a list of lists of bindings, each of which is processed by the current Wyl interpreter and used to continue the search. To ensure the correct binding for **$state**, the binding for **$newstate** returned from **trueps** is reassigned to the **$state** variable at each recursive invocation. This process is similar to renaming variables during recursive invocation of a rule in logic programming languages.

The connectives of the schema language operate on the binding lists returned from the primitives in different ways depending upon the current interpreter.

First, consider the connective o-or under the envisonment interpreter. In this mode, o-or is rather like a LISP or, in that the computation continues until a non-nil value is returned from one of the primitives. The clauses are evaluated in left-to-right order. For the checkers schema given above, during envisionment Wyl will first determine if the current state is any of the known goals (such as *loss* or *trap*). If so, evaluation will cease and, because the primitive is single-terminate, the search will terminate. If the state is not recognized (i.e., single-terminate returns nil), Wyl determines if take moves exist, if so the o-or will terminate and, because the primitive is multiple recursive, the search will continue in the new states. If none of the above applies, the normal moves will be evaluated. If none of them apply then the o-or terminates with nil.

The connective o-and is used in the specification of the search scheme for *chess* and is given in Appendix A. During interpretation by the envisonment interpreter o-and is like the LISP and—computation continues until a nil is encountered.

Representation of Goals

Goals are defined by rules written in MRS. Initially, as part of the domain specification, the standard goals of *loss* and *win* are defined. Other goals can be defined. For example, in chess, a predicate is supplied that recognizes when a queen is lost. During learning, the system will add new definitions of goals such as *trap* above.

The consequents of the rules defining goals must unify with the form used in the specification of the search schema discussed above. For example, *loss* in *checkers* is given below:

```
(if (and (unprovable (move $name $state $newstate $type $from $to $side))
         (unprovable (take $name $state $newstate $type $from $over $to $side)))
    (terminate-state $name loss $state $side)).
```

Note that the goal is not specified structurally, in fact it is very difficult to structurally define *loss* correctly in checkers. The correct definition of a *loss* is when a player has no legal moves available as defined above. A fair approximation to a *loss* is when the side to play has no legal playing pieces on the board, but this is not the correct definition and fails when pieces exist but are blocked.

The full power of MRS stepping outside of first order logic is exploited here. The modal operator unprovable is used to determine if there exist no legal moves for the side $side in state $state.

Representation of Actions

The actions of a domain are defined by supplying rules which compute the legal moves available given a state and a playing side. The operator specifications must

conform with those definitions given on the mrsform slot of the frames for the multiple-recursive primitives discussed earlier. In addition the forms must include the variable $newstate, which is bound to the state formed by applying the operator to the originally supplied state.

In keeping with one of the stated goals of this thesis, the operators are defined in terms of the simple structural language—no special features are introduced. Each operator in checkers is defined by the squares and type of piece involved. For normal moves, only the *from* and *to* squares are specified, while take moves require an additional *over* square.

To illustrate the simplicity of the operator specification, we give the rule defining normal moves in checkers. The rule has two parts, one that creates legal moves and one that makes the moves on the board.

First the main rule:

```
(if (and (find-move $state $type $from $to $player $side)
         (make-move $state $name $from $to $player $newstate))
    (move $name $state $newstate $type $from $to $side))
```

The consequent unifies with the MRS formula stored on the normal-move frame of the search schema. The first subgoal find-move acts as a generator of legal normal moves, while the second subgoal is side-affecting and creates the new name ($name) and state ($newstate).

```
(if (and (side $player $side)
         (type $player $type)
         (occupied $state $from $player)
         (connected $from $to $direct)
         (legal-direction $side $type $direct)
         (occupied $state $to empty))
    (find-move $state $type $from $to $player $side))
```

The first part of the move predicate above finds the legal moves by determining if the playing pieces of the side under consideration can move into adjacent empty squares connected in a direction which is legal for the piece concerned.

```
(if (and (newname $from $to $name)
         (new-state $newstate)
         (add-info ( (occupied $newstate $to $player)
                     (occupied $newstate $from empty)
                     (nextstate $state $newstate))))
    (make-move $state $name $from $to $player $newstate))
```

The second part includes all the side-affecting code. First a new name for the move is created, then a new state name. The add-info predicate asserts the given formulae into the MRS database to define the new state as described in Section 2.1.

Representation of Functional Concepts

Functional concept descriptions are formed by combining specifications of action sequences with goals. Initially the domain knowledge and the general search procedure characterize a functional description of the most general concept—any legal board position. Specified functionally as any number and kind of legal operators resulting in any known goal. Wyl, as a result of learning, forms descriptions representing particular sets of board positions as specializations of this general definition. For example, *trap* (discussed below) describes a particular way to lose a game of checkers and is therefore a specialization of the concept *loss*.

In this section we first describe a typical functional concept, *trap* in checkers. We next describe how Wyl represents such concepts in its functional language.

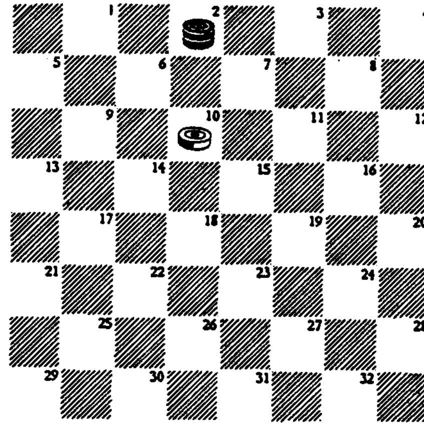


Figure 2.3: Trap position 2, red to play

A functional concept

In Figure 2.3 we give an example of the checkers concept *trap*. It is red's turn to move. The red king on square s2 is trapped by the white man on square s10. No matter what move the red king makes the white man can take him.

Below we give a definition of *trap* in logic.

$$\forall\, sq1\ sq2\ state1\ side1$$

$$TRAP(state1\ side1) \Leftrightarrow$$

$$\forall\, move1\ type1\ to1\ nextstate1$$

$$legalmoves(state1\ side1\ move1) \supset$$

$$normalmove(move1\ state1\ nextstate1\ type1\ sq1\ to1\ side1)$$

$$\wedge \exists\ \ state2\ nextstate2\ move2\ to2\ type1$$

$$takemove(move2\ state2\ nextstate2\ type1\ sq2\ sq1\ to2\ side2)$$

$$\wedge Terminatestate\ (\mathrm{loss}\ \ nextstate2\ side1)$$

The definition of *trap* above consists of three descriptions, two describing the

sequence of operators, the other describing the goal. Generally, concept definitions consist of a fixed number of operator descriptions and a known goal.

A position is a *trap* for *side*1 iff the following are true:

- All legal moves available to *side*1 are normal and involve moving from square *sq*1 to square *to*1 forming state *nextstate*1.
- At *nextstate*1 there exists a move for the opponent from square *sq*2 that captures the piece on *to*1 leading to state *nextstate*2.
- The state *nextstate*2 is an recognized *loss* for side *side*1.

There are two important aspects of the concept description. The first aspect is the quantification over the operators and how this is employed to specify forced moves. The second aspect is how the goals and the operators are described. We discuss each aspect below:

Quantification of operators. The first move defined for *side*1 in *trap* is forced. The moves available are restricted to be from square *sq*1 and *all* lead to a *loss*. On the other hand, the moves for the opponent are not forced; a position is a *trap* if *there exists* a move taking the piece on square *to*1. This alternating quantification is a product of the "adversary game" domain. In general the quantification is dependent upon whether the outcome is favorable to the current player. To prove that a position is unfavorable to player (e.g. *side*1 in *trap* above) we must prove that *all* the moves available lead to an unfavorable result. To prove a position is favorable (e.g., *side*2 in *trap* above), we need only prove that *there exists* a move leading to a favorable result.

Operator and goal description The operators in the description of *trap* are restricted to be certain kinds (*normal* first *take* second) and interact in certain ways (the take move jumps over *to*1, the destination square of the *normal* move). The final goal is restricted to be a *loss*.

Representation in Wyl

The representation of *trap* in Wyl consists of three descriptions that correspond to the descriptions given above. The first describes the *normalmove*, the second describes the *takemove* while the third describes the *loss* goal. Each of the descriptions is represented in a uniform way as instantiations of the general search schema given earlier. The descriptions (termed concept states) denote a set of constraints on the primitives in the general schema. The first concept state of *trap* specifies constraints for the multiple recursive primitive normal-move, the second for the multiple recursive primitive take-move, and finally the third specifies constraints on the single-terminate primitive terminate-state.

The three concept states are given below:

```
(checkers/trap-0
    normal-move
        tstate-0 ( ($quantification . for-all)
                   ($next-concept-state . nextfc-2)
                   ($type . $type1)
                   ($from . $sq1)
                   ($to . $to1)
                   (t . t)) )
(checkers/trap-0
    take-move
        nextfc-2 ( ($quantification . there-exists)
                   ($next-concept-state . nextfc-3)
                   ($type . $type1)
                   ($from . $sq2)
                   ($over . $to1)
                   ($to . $to2)
```

```
                    (t . t)) )
    (checkers/trap-0
        terminate-state
            nextfc-3 ( ($result . loss)
                       (t . t)) )
```

A concept state is a four tuple; the first item is the concept name, the second is the primitive the constraints apply to, the third is the concept state name, and the last gives the actual constraints on the primitive. All concepts begin at tstate-0, the remaining states are found by following $next-concept-state values.

The constraints are defined as bindings for each of the variables found in the MRS form attached to the corresponding primitive of the general schema. The binding values can be constants or variables. An example of a constant parameter is the $result value for the primitive terminate-state of nextfc-3, which is loss. This constrains the terminating states to be losses.

Binding values that are variables are used to express the operator descriptions given above. The variables can express unrestricted constraints on MRS form (the $type of piece can be a king or man) by giving a unique variable as a binding. For example, the $type binding for the first operator is given as $type1, which appears no where else in the concept description.

In addition, by using variables as binding values we can capture constraints between operators. In *trap* the second take move must jump over the square the first operator moved into. This is captured in the description of trap above by giving the binding of the $to variable for the first concept state (the normal-move) the same value ($to1) as the $over variable of the second concept state (the take-move).

To see how this technique enforces the constraint between the operators, some understanding of the test interpreter is needed. The test interpreter is fully described in Section 2.3.2.1, here we give only a brief overview. The test interpreter

performs forward search passing bindings down the tree in similar manner to the other interpreters, it differs on how the primitives are interpreted. First, at each recursive invocation during search the current concept state (i.e., tstate-0) is updated and used to access the constraints information. Only those primitives that have constraints specified are interpreted. Before calling the required MRS function, such as **trueps**, for the first operator, the constraint bindings are plugged in. The resulting bindings are then passed down the search. During interpretation at the second level, therefore, the binding lists will contain a binding for $to1, say *square1*. The constraints for the take move will be looked up and plugged in the mrsform, replacing $to of the (take ...) form with $to1. Next the bindings passed down the tree are plugged in, further replacing the $to1 variable with *square1*. The resulting form,

(takemove $name *state2* $newstate $type1 $sq1 *square1* $to2 *red*).

is backchained on, and will therefore return only those take moves existing on the board which jump over *square1*.

The constraints of the operators (the multiple-recursive primitives) have an additional binding named $quantification. This is used by the test interpreter to check for forced moves when the value is for-all and non-forced moves when the value is there-exists.

In addition, the scope of the variable bindings can be specified as part of a concept definition. In the beginning of this section, we gave a first order logic description of the concept *trap*. Note that the two variables *sq*1 and *sq*2 have global scope over the whole concept description. It is important to capture this information in Wyl's concept descriptions, as the scope of variables is needed to correctly describe concepts. For example, there are many positions which are similar to *trap* but involve the first players piece being captured by either of two opposing pieces (i.e., *sq*2 has only local scope). Wyl, with its expressive power of scope, is

able to differentiate between the two versions of trap. During concept formation, Wyl identifies variables that have global scope. The *trap* concept definition includes the following facts which express this global scope:

(global $sq1)
(global $sq2).

2.3 Process in Wyl

The processes in Wyl are summarized in three sections. First the processes of envisonment and generalization, which inductively learn the functional concepts, are described. This is followed by a description of the processes that apply the learned functional concepts to the performance tasks of test and generation. Finally the procedures that construct the structural concept language are given.

2.3.1 Learning the Functional Concepts

Wyl inductively learns concepts in the functional space, yet it is supplied with simple structural instances, that is, board positions. The first task of the learning component, therefore, is to translate the supplied instance to a functional form ready for generalization.

This section first describes the envisionment interpreter that performs this knowledge transformation. This followed by a description of the inductive generalizer.

The Envisonment Interpreter

The envisionment interpreter takes as input the supplied structural instance and the specification for the domain and produces as output the equivalent functional

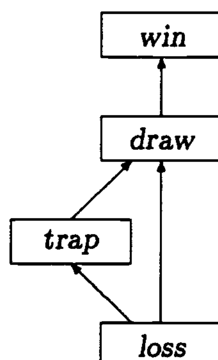


Figure 2.4: Preference of recognizable states for checkers

instance. To form the functional instance (refered to as FI), Wyl applies two procedures. The first constructs a search tree while computing the outcome of the structural instance (SI) by performing forward min/max search looking for recognizable states. The second traverses the resulting search tree and extracts from it only those operators and termination states that were relevant to the proof of outcome.

Min/max search determines the outcome of a position by applying knowledge about the main goals of the players concerned. To correctly predict the outcome, it is important therefore to not only be capable of recognizing termination states, but to know which of them are *preferable* to the players. This preference ordering is represented in Wyl as facts asserting (better-than *state1 state2*), when *state1* is preferable to *state2*. The ordering is illustrated in Figure 2.4 as a directed graph. *Trap* is not initially part of this ordering, but it is added as the result of the learning process. (Note that *trap* is considered better than *loss* as it delays the end of the game.)

First, consider the min/max search procedure. The procedure searches forward

depth first, interpreting the general domain schema at each recursive invocation.

During the search down the tree, the current state and side to play are passed down in the manner previously described. The primitives of the schema are evaluated left to right. If the state is recognized, the search terminates and binding for $result and $side are returned. A depth limit is imposed on the search that, when reached, terminates the search and returns a result of *draw*. Otherwise all legal operators are found and explored. At each invocation of the general schema, a cache is made of the computation to form the search tree. Each node in the tree represents a complete instantiation of the general schema. First each primitive of the schema which was interpreted is saved. In the case of checkers, because the connective is o-or, only one can apply. For *trap* the cache of the first state includes the primitive (multiple-recursive normal-moves). The MRS form of the normal-moves slot will generally have many instantiations, one for each operator available. Each is cached in the tree.

On the return from recursion, the envisionment interpreter determines the outcome for each state in the standard min/max manner. The outcome for *state*1 with *player*1 to play is either a recognized goal or the most preferred goal according to *player*1, returned from applying all the legal operators. The outcome is then cached on the node.

The resulting tree will be the exhaustive forward search tree to the depth limit. Once the outcome is known, the parts of the tree relevant to that outcome can be distinguished from those parts irrelevant. Considering *state*, with *player* to play and result *outcome*, we can identify two cases: when *outcome* is advantageous to *player* (i.e., *outcome* > *draw*) or when the converse is true. First considering the advantageous case. Here it *sufficient* for there to exist one operator leading to *outcome* (out of the many possible operators) for the outcome to be better than a *draw*. It is this case where the full tree is pruned. Only the operator that leads to the

selected goal is included in the proof, as the others have no effect on the computed outcome. To encode the fact that there need only exist one operator to guarantee the outcome, the state is marked with $\exists$ as a quantification over the operator. The other case is where, when all the operators available were explored, the best result found was worse than a *draw*. No pruning can occur here because there must not have existed an operator which returned a better result, making all the operators from this state *forced*, and therefore *necessary* to the outcome. Another way of viewing these forced moves is counterfactually: if any other operator existed which returned a different result, the outcome of the whole proof would be changed. It is for this reason that the quantification over the operator from this state is marked $\forall$ to encode the fact that these moves are necessary to the outcome.

The operation of pruning and quantification of the nodes in the explanation tree can be viewed as incorporating tests into the instance that are implicit in the preference order and goals of the players used during the min-max search. This knowledge is used as tests by the test interpreter during classification to avoid a full exhaustive search (see Section 2.3.2.1).

The Generalizer

The generalizer takes a functional instance that is an example of the concept (formed by the envisonment interpreter) and the current functional concept description, and returns a more general functional concept description that covers the instance. Generalization is inductive and driven, as are all inductive generalizers, by a combination of syntactic similarities in the concept space and bias. The resulting concept is the maximally specific unifying generalization of the two inputs.

Generalization by simple syntactic matching and rewriting will only work when the concept and instance descriptions satisfy the constraints introduced in Chapter 1. The constraints can be summarized by saying the that most effective representa-

tion is the one in which the concepts can be described "naturally." These constraints are satisfied in Wyl since a functional representation is employed to learn functional concepts.

Bias is characterized by generalization operators that act on the syntactic forms of the instances and the concept. The biases of concept languages, by their very nature, are stated in vocabulary-specific terms. In Wyl, because of the simplicity of the expressions in the functional space, the biases used are straightforward and do not require any complex hierarchy of vocabulary terms, common in structural concept languages. Two simple biases are employed in Wyl:

Single line of play: Functional instances are trees of fully instantiated (i.e., ground) search schema describing particular operators and goals. One the other hand, functional concepts are single generalized sequences of search schema describing general operators and goals. To form the concept from the instance, the generalization step "compresses" the branches of the tree in the instance. The resulting single sequence is the maximally specific unifying generalization of the instance. The compression is achieved by applying the inverse enumeration rule given below:

$$\begin{aligned} &Operator_1(A_1\ B_1 ... Z_1)\\ &\wedge Operator_2(A_2\ B_2 ... Z_2) ....\\ &\wedge Operator_n(A_n\ B_n .... Z_n)\\ &\Longrightarrow \forall a\ b ... z Operator(a\ b ... z). \end{aligned}$$

Compression applies the inverse enumeration rule to the schema instantiations begining at the root and terminating at the leaves.

No coincidences: This particular bias applied to Wyl can be stated more concretely. If the same constant appears at different points in the same instance, it is asserted that these points are *necessarily* equal. It is this bias that

identifies and preserves the interrelation constraints found in concepts. The checkers concept *trap* demonstrates this bias. For the example illustrated in Figure 2.3, the proof tree has two branches corresponding to the choices available to the first player. On the branch where the player moved to s6, the take move of the second player jumped over s6. This bias says that the two occurrences of s6 are equal. On the other branch where the player moves to s7, the take move of the second player jumped over s7. Again the bias says these occurrences of s7 are equal. When the two branches are matched against each other during generalization, the squares s6 and s7 will match in two places, as *to* squares of the first operator and *over* squares of the second operator. This bias says that these two sets {s6 s7} are equal and hence, are generalized to the *same* variable. This bias also applies to matches of variables to constants, where the same variable may match against the same constant.

Before taking a more detailed look at the generalization procedures, it is necessary to describe the representation employed in expressing functional instances and concepts. The representations employed for functional instances and concepts in the generalizer differs from that presented in Section 2.2.2.2; a more uniform representation is adopted to simplify the generalization process. The envisionment interpreter builds functional instances in this representation and the final generalized concept is similarly represented. To build the expressions given in Section 2.2.2.2 representing concepts used by both the test and generator interpreter, the final form is traversed.

An important characteristic of representations employed in inductive generalizers is their simplicity and uniformity. One of the most universal, powerful but simple representations is the frame, with slots and values. Trees are captured in this representation by allowing slot values to be frames themselves. Each schema instantiation is represented as a sub tree of frames, which are arranged in a tree,

mirroring the computation.

The functional instance created by the envisonment interpreter (when it is given the structural instance of a *trap* shown in Figure 2.3), is illustrated in Figure 2.5.

The boxed nodes are frames, representing two levels of instantiation of the general schema. The beginning frame and those others called state name each recursive instantiation of the general schema and coincide with the board states created during search. Each instantiation or concept state, represents a cache of the computation performed during the particular invocation. For example the first concept state records the fact that at this point in the search the (multiple-recursive normal-move) primitive was invoked. The primitives also have invocations, that represent the solutions found to the mrsform attached to the primitive's unit. When the primitive is multiple-recursive there will usually be many such invocations, while single-terminate primitives only have one invocation. Each primitive invocation is named as a separate frame whose slot values are the variables found in the relevant mrsform. For example the instantiations of the first normal-move are named, s2-s6-5, s2-s7-3 which represent the legal operators available to the first player and have slots such as to, from, type giving the bindings values found for the mrsform. When the primitive is multiple-recursive, additional quantification information is included as described in the previous section.

The simple frame slot value representation needs some extension to enable the generalization procedure to uniformly traverse the tree. Each frame is associated with an additional value, which represents the kind of invocation it describes, whether it is an invocation of part of the schema such as normal-move or particular instantiations of a primitive. The tree structure is represented implicitly by describing each individual fact as a four tuple in the MRS data base, for example the following describe the top of the instance:

(normal-move state-6 quantification for-all)

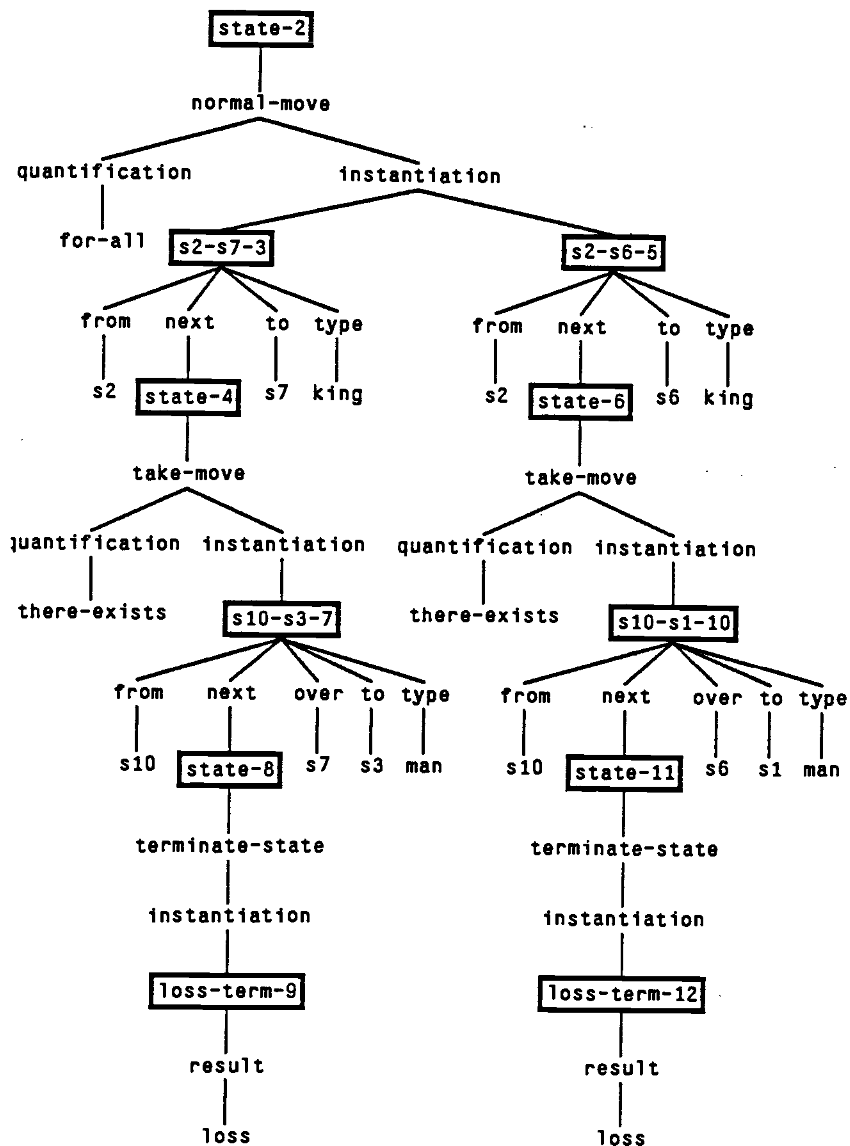


Figure 2.5: Functional instance of *trap* position in Figure 2.3

(normal-move state-6 instantiation s2-s6-5)

(normal-move state-6 instantiation s2-s7-3)

(mrs-invocation s2-s6-5 from s2)

(mrs-invocation s2-s6-5 to s7).

The general form of these four tuples is (instantiationkind framename slot value)[1].

The algorithm that generalizes the functional instances and concepts is relatively straightforward. It basically traverses the tree from root to leaves, compressing instantiations of frames at each level to single frames.

The algorithm recursively invokes a procedure compress, which takes a *framename*, representing many invocations, and compresses it to one general instantiation by applying the single line of play bias. Compress first obtains a list of sets of frame values that need generalization in order to reduce the frame to a single instantiation. Each member of the list is the set of value's whose framename is equal to *framename* and whose instantiationkind and slot are unique. Considering the example illustrated in Figure 2.5, when *framename* is state-2 one of the sets of values will be {s2-s6-5, s2-s7-3}, (for the instantiation slot), another {s6, s7} (for the to slot). To form a single instantiation each slot must have only one value, therefore each of the value sets must be reduced to a singleton. This is achieved by a procedure maximally-specific-generalization that returns a unique name representing the input set. In the example, the set {s2-s6-5, s2-s7-3}, is generalized to instantiationfc0 and the set {s6, s7} is generalized to $tofc-0. All slot types other than instantiationkind are turned into MRS variables[2].

Next the tree is modified. The original facts (the 4 tuples introduced earlier) containing the values in value *set* are all replaced by a single fact involving the new

[1]The graph of the instance in Figure 2.5 does not included the MRS-invocation information for clarity and brevity.

[2]instantiationfc0 is a variable for the Wyl interpreters.

value returned from (maximally-specific-generalization *set*)[3]. When the values are frame names (e.g., s2-s6-5, s2-s7-3), another modification is required in addition to changing the value information—that is, changing the framename's themselves. How this forces the correct compression can be seen when we consider the next invocation of compress with instantiationfc. In this case, because the facts associated with the two frames {s2-s6-5, s2-s7-3} are now all associated with only the one frame instantiationfc, the list of value sets obtained will include sets of values from the original two frames. For example, the values relating to the slot over will be {s6, s7}. The process continues with each value set being mapped to a single variable by maximally-specific-generalization. The set {s6, s7} is mapped to the MRS variable $tofc-0, while the set of frame names, {state-8, state-11} is mapped to nextfc-1. The process terminates when the procedure maximally-specific-generalization creates no more new frames. That is, compress has reached the leaves of the tree. The generalized functional instance formed from compressing the FI in Figure 2.5 is illustrated in Figure 2.6.

Note that the generalization process is in no sense knowledge intensive. The only slot values treated any differently during compaction are those whose values are frames. There can be any number of other slots relating to objects in the domain such as from, to, toemptysquare, etc., but to compress they are all simple syntactic forms.

Biases are encoded in the procedure maximally-specific-generalization which we will now describe in some detail. If the set passed to the procedure is a singleton, it is returned, else a single variable representing the set is returned. The procedure treats sets of frames differently, in that the value returned is simply a new framename as in the case above. Otherwise the set represents instantiations of the MRS forms

[3] In fact no real modification takes place, changes are implemented by marking all facts with situation variables and using the STRIPS assumption and current situations to retrieve facts.

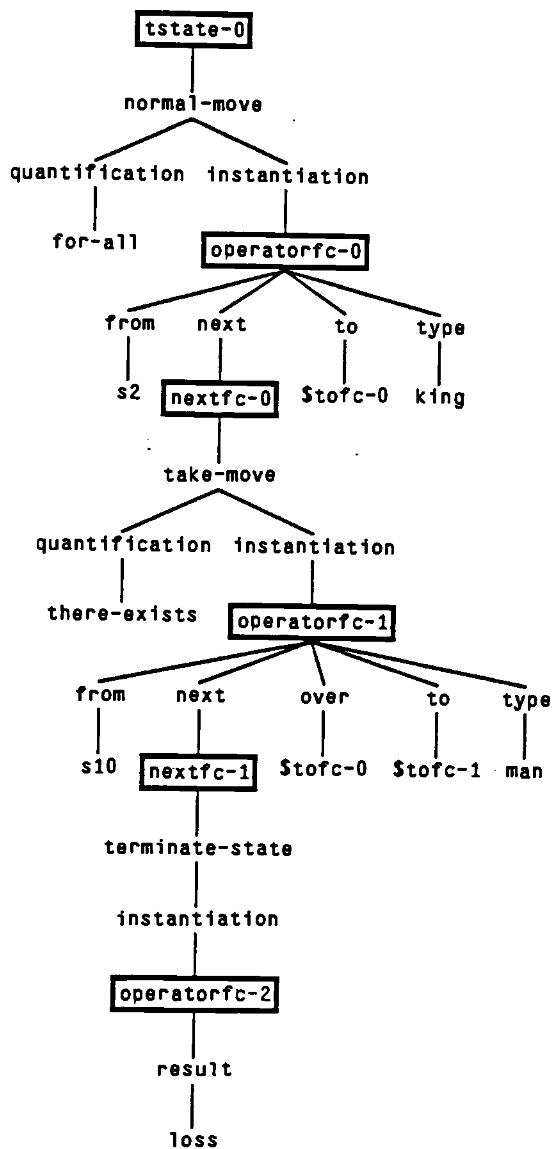


Figure 2.6: Generalized functional instance of *trap* position Figure 2.3

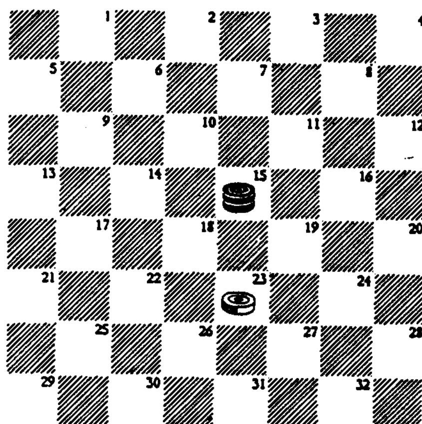


Figure 2.7: Second training example for concept *trap*.

and is treated in a more complex manner to correctly implement the biases.

Consider the "no coincidences" bias. When the procedure is called with a set, say *set*1, it first determines whether is has been assigned a variable previously. If so that variable is returned. Otherwise a new variable is formed, cached, and returned. For example when *set*1 = {s6. s7} is first seen during compaction of the first operators, the new variable $tofc-0 is created and returned. Next time, during the compress of the second level, the input set {s6. s7} will be compared with known sets and assigned the same variable.

Once the system has formed a generalized functional instance, it is generalized further against the existing concept definition (if any). In the *trap* case above, we reviewed Wyl with its first training example. Hence, the functional instance illustrated in Figure 2.6 is the *trap* functional concept (referred to as FC) after the one example. It is too specific since it describes *trap* in exactly one location on the board.

The second training instance is given in Figure 2.7 with white to play. This

structural instance looks very different from the first instance (Figure 2.3) and so has a very different structural representation. However, the resulting generalized functional instance, given in Figure 2.8, is very similar to both the previous instance and the current concept definition.

This second FI is generalized against the current concept in a similar manner to that used to generalize the alternating lines of play in an instance. The descriptions linked into a tree with one branch being the new FI and the other being the FC. The tree is traversed by compress applying the inverse enumeration rule. Consider generalizing the first FI (Figure 2.6) with the second FI, (Figure 2.8). The procedure maximally-specific-generalization simply renames variables when two match such as {$tofc-2, $tofc-0} to $tofc-4. Frame names are similarly renamed such as {operatorfc-0, operatorfc-3} to operatorfc-6. Real generalization takes place when there are two constants that match. For example, while compressing the first operator the constants {s12, s15} match and are generalized to $fromfc-0. Here, because this generalization is over concepts, the new variables created at this stage are given global scope over the concept. Hence, the generalizer asserts the following as part of the concept definition:

```
(global $fromfc-0)
(global $fromfc-1).
```

The final functional concept *trap* is given in Figure 2.9, after being show the two training board positions, Figure 2.3 and Figure 2.7. Note that in the final concept the globally scoped variables are boxed lightly. The concept description given in Section 2.2.2.2 is formed by traversing the tree representation of the concept in Figure 2.9.

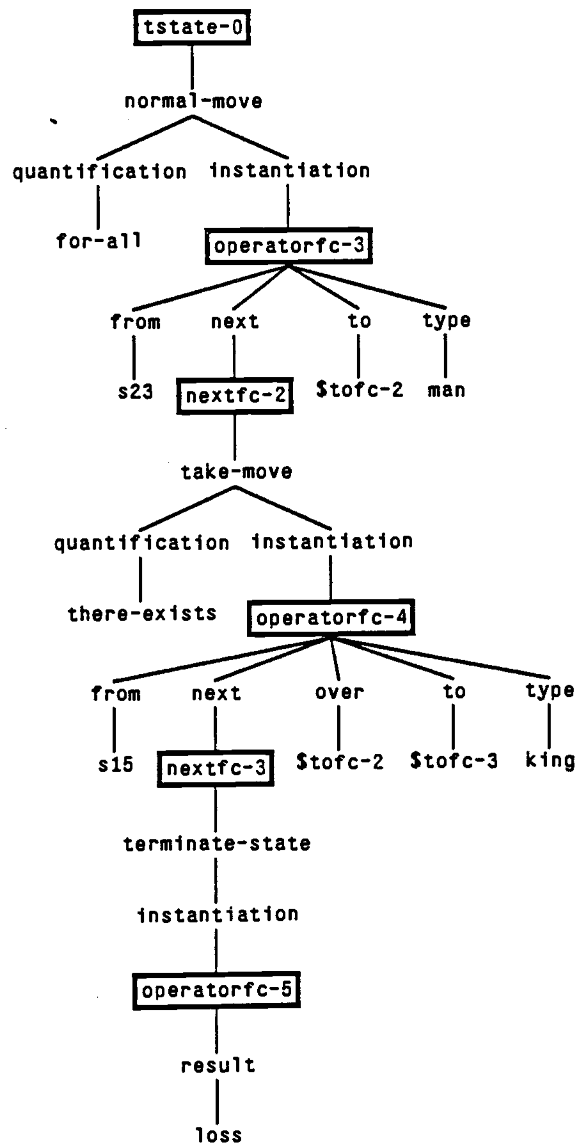


Figure 2.8: Generalized functional instance of *trap* position Figure 2.7

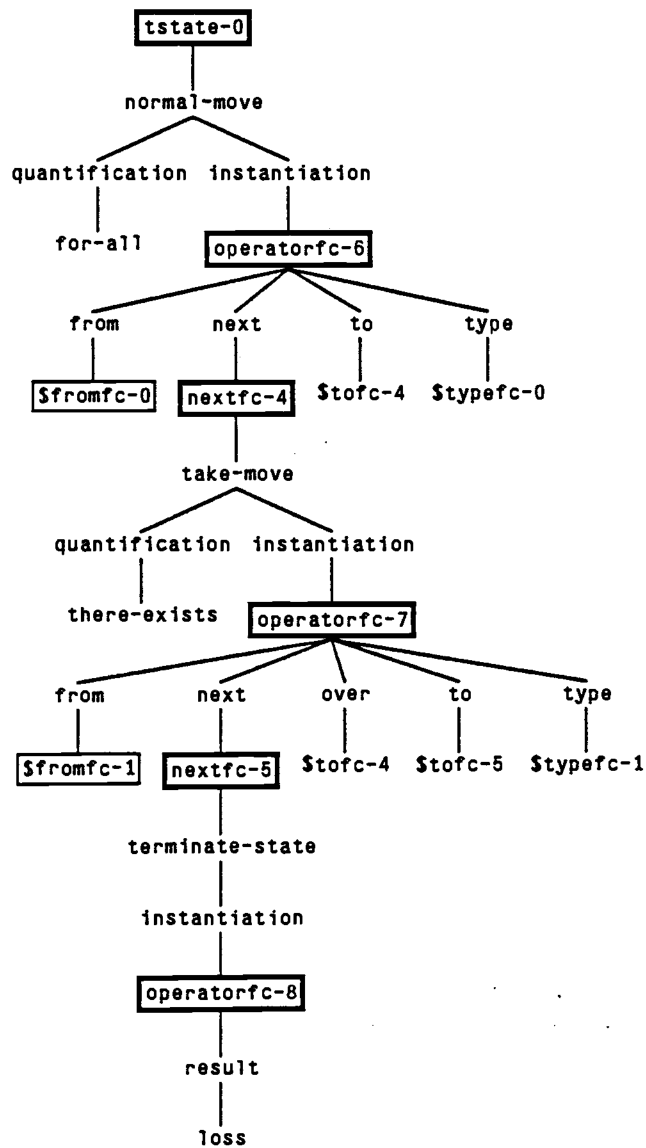


Figure 2.9: Generalized functional Concept of *trap*

2.3.2 Applying the Functional Concepts

Once a functional concept definition has been learned, Wyl applies it to two tasks. First, the functional concept is used to test whether new structural training instances are examples of the concept. Second, during compilation, the functional concept is used to generate all possible structural examples of the concept. The first task is performed by the "test interpreter," and is discussed in the first part of this section. The second task is performed by the "generation interpreter" and is discussed in the second half of this section.

Applying FC to Test Structural Instances

The test interpreter takes a functional concept, the domain specification, and a structural board position and returns T or NIL indicating whether the instance is an example of the functional concept. The main idea behind using a functional concept as a test is to view it as a proof procedure. Recall that the functional instances generated by envisionment are proofs of the outcome of the position. The functional concepts are generalized proofs, and specify a series of constraints on the outcome and operators of structural instances, which if satisfied, determine that the instance is an example of the concept.

More concretely, an instance is an example of a concept if it satisfies the following:

- The instance has the same outcome as the concept.
- The moves involved in the outcome are those that satisfy the constraints of the concepts operator descriptions.
- Any quantification constraints, such as globally scoped and shared variables, are satisfied by the instance.

The test interpreter is like the envisionment interpreter in that forward domain search is performed. It differs in the concept description applied during search: the test interpreter applies the particular concept under test (i.e., the concept's search schema instantiations), while the envisionment interpreter applies the general concept (i.e., the domain schema). The envisionment interpreter attempts to prove that outcome of the given instance is *any* one of the known goals. In contrast, the test interpreter attempts to prove that the given instance satisfies all the constraints in the given concept.

The test interpreter determines whether the board position under question can form a legal instantiation of the functional concept. To do this, the test interpreter performs forward search. Each concept state of the concept is re-instantiated with the values from the board position, rebuilding a functional instance. For example, if one of the original training instances were given, the resulting proof tree formed would be exactly the same as the functional instance constructed by the envisionment stage. If the instance can form a legal instantiation it is deemed an example: otherwise, it is not an example.

The search is done depth first and builds an **and/or** proof tree of schema instantiations. Each **and** node corresponds to a concept state that specifies a universally quantified multiple-recursive primitive, while each **or** node corresponds to an existentially quantified multiple-recursive primitive. The search terminates either when the current concept state specifies a single-terminate constraint to be satisfied or when the instance fails to satisfy a constraint. Each schema instantiation may involve an **and/or** subtree corresponding to the original structure of the connectives of the domain schema. In checkers, because the only connective is one or, there is no subtree, but in chess where both and and or are used, this must be considered[4].

The test interpreter is like the envisionment interpreter in that bindings lists are

[4]In chess the conective is an and because at each stage of the search the side to play must determine whether they are in check. See Appendix for more information.

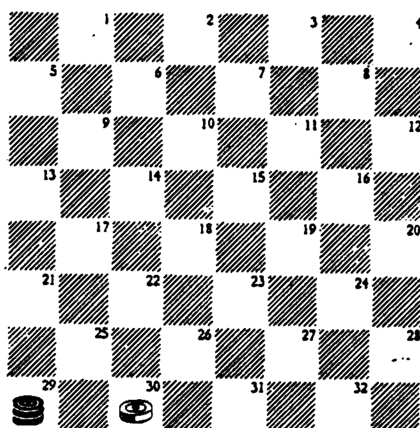


Figure 2.10: *Trap* position (state1) with red to play

used to carry information down the search, and the general domain schema is used to determine the primitives that could apply and to supply the mrsforms needed for backward chaining. It differs in that the concept states are used at each level of the search to determine how the general schema is to be interpreted, and in the way the binding lists are passed down.

The operation of the test interpreter is best explained by an example. Consider Wyl applying the test interpreter to the board position illustrated in Figure 2.10 after the *trap* concept given in Section 2.2.2.2 has been learned.

First, the usual $state and $side bindings of the envisonment interpreter are supplemented with a binding for &conceptstate which holds the current concept state name. The bindings for each level of the proof tree will hold the name of the concept state relevant for that level. In this case the initial bindings will include (($side . red)($state . state1)(&conceptstate . tstate-0)). At the next level the &conceptstate binding will be nextfc-2, which is obtained by following the $next-

concept-state binding given on the concept state[5].

The concept state information is used to access the constraints for each primitive in the general schema. First, for each primitive in the general schema, any constraints are looked up. Only those primitives with constraints are interpreted. In the example (Figure 2.10) when analyzing state1, the (multiple-recursive normal-move) form is interpreted with those constraints illustrated in Figure 2.9. Before the mrsform specified on normal-move is used for backward chaining, the constraints are first plugged in, then the bindings on the binding list passed down from above are plugged in. The resulting form which trueps is called on is:

(move $name state1 $newstate $typefc-0 $fromfc-0 $tofc-0 red).

The MRS move rule that unifies with the form above will compute the legal normal moves available for red. Here, only one move is available, from s29 to s25. As the primitive is multiple-recusive, the interpreter will recur on each of the new bindings returned. The new binding list passed to the next invocation has &conceptstate. $side, and $state set as previously discussed. Where the test interpreter differs from the envisionment interpreter is that the other bindings, usually cached, are passed down the search tree. Therefore, the binding list of the next invocation of the search will include the following bindings: (($state . state2) ($side . white)(&conceptstate . nextfc-2) ($tofc-0 . s25) ...). This passing of bindings is the same process MRS and other logic programming languages use to solve conjunctive goals.

At this new state in the search, the same process will occur, with the current concept constraints applying to the (multiple-recursive take-move). The mrsform, after the constraint bindings and values bindings have been plugged in, is given below:

(takemove $name state2 $newstate $typefc-1 $fromfc-1 s25 $tofc-1 white).

[5]See Figure 2.9

Hence we see how Wyl captures the important relational constraints by sharing bindings. The take move must be one which jumps over the piece on square s25. The MRS rule defining the take moves will find the move for white from s30 to s21, creating a new state (state3) in which red has no playing pieces. The process continues for the next level where the &conceptstate nextfc-3 specifies single-terminate constraints. The resulting mrsform after binding is:

(terminate-state $name loss state3 red).

This will succeed.

We have now discussed how the test interpreter performs forward search. Now we consider how values are passed back up the proof tree and the conclusion of the proof, either T or NIL is computed.

First consider the schema language connectives. The o-or returns T if the primitive which has constraints specified returns T, while the o-and returns T if all the primitives return T.

The simplest primitive is single-terminate, illustrated above with the *loss* position. The interpreter uses the single solution MRS backward chainer **truep** to determine if the form is true for the supplied state. If so, T is returned, otherwise NIL is returned.

The primitive multiple-recursive is more complex, as the quantification assigned during envisionment must be taken into account. The quantification information is important to the concept definition, because it encodes the knowledge gained from the min/max search. The test interpreter need not perform full min/max search, because if the quantification constraints are met, the outcome is guaranteed to be that defined in the concept. The justification for this is the same as that presented during the envisionment stage. Here we demonstrate how the test interpreter employs this knowledge to prune its search.

For a multiple-recursive existentially quantified primitive to return T, there must

exist at least one instantiation of the constrained mrsform that returns T. In other words, the existentially quantified primitives form the or nodes of the proof tree. In the *trap* case above, this is not well illustrated, as there was only one move available. But in the *chess* concept *skewer* given in chapter 3 there are many moves available. These nodes apply to operators for the side that is gaining the advantage, in the *chess* case Wyl must show there exists a move that can capture the opponent's queen.

When the quantification is universal, the operators are defined as *forced*. The test interpreter, to return T, must show that two conditions are met: (1) there exist no operators other than those specified by the constraints and (2) for each of these operators, a T is returned from the corresponding subtrees. It is here that the search can be terminated prematurely with NIL when there exist operators that are not described by the constraints. The rationale behind this is that these universally quantified operators represent *forced* moves, ones which where taken in the original training instances because there was no other alternative. Therefore if the test interpreter can demonstrate that in this case there does exist an alternative, the current instance cannot share the same functional features of the concept examples and must not be an example of the concept. An example of this is given in the next chapter for the *chess* concept *skewer*, where the king is forced to move out of check, unavoidably exposing his queen to capture. The test interpreter is able to check if indeed this move is forced, by determining if any other moves exist. It is often the case that a board position *looks* like a *skewer*, (i.e., is structurally similar) but because there exists a way to block the check, the king is no longer forced to move, and the queen is safe.

The implementation of forced moves is straight forward. Recall (from Section 2.2.2.2) that a forced move is represented in logic as follows:

$$\forall\, state\ move\ nextstate\ type\ from\ to$$
$$legalmove(state\ side1\ move) \supset$$
$$normalmove(move\ state\ nextstate\ type\ from\ to\ side1).$$

This is true only when the set of *all* legal moves is equal to the set of constrained moves. The test interpreter generates all legal moves using the general mrsform from the current primitive and the legal moves from the constrained mrsform. If the sets are equal, T is returned. If they are not equal, NIL is returned.

Once the general operation of the test interpreter is understood, the method of dealing with the global variables can be explained. These variables complicate the interpreter, because they represent constraints across branches of the search tree. For example, in the *trap* cases illustrated previously, when the first player has two moves available, the second player's move must originate from the same square ($fromfc-1 in the concept given in Section 2.2.2.2) for both branches of the search tree. Therefore once a binding is established for $fromfc-1, it must be communicated to the other branch to ensure the constraint is met. This communication is achieved by global assertions of bindings of the variables in question and by having the test interpreter check whether any of the variables in a form are global. If so, the interpreter looks to see if any bindings have been posted. When alternate bindings are available, the system uses a very simple chronological backtracking algorithm. A much better way to efficiently solve this problem is by dependency directed backtracking.

Applying FC to Generate Structural Instances

The generation interpreter takes a functional concept and the domain specification and returns the set of all structural instances that are examples of the concept. The idea behind the generator is to use the functional concept as a *constructive* proof and run it "backwards" to generate all the instances that would return T if tested.

The procedure is computationally expensive[6].

The easiest way to implement such a generator is to create a procedure that generates arrangements of checkers or chess pieces on a board and test them with the procedure described previously. A better way is to incorporate some of the tests into the generator so the generator only produces instances that are examples of the concept.

The generator incorporates tests by incrementally assembling the structural instances of playing pieces that meet the constraints in the functional concept. It is similar to the test interpreter in the way the concept is used, but instead of testing a supplied structural instance at each level of search, the generator forms new instances that satisfy the constraints by making assumptions about squares being occupied by playing pieces. The assumptions are made and controlled by applying `residue` (Finger & Genesereth 1983). `Residue` is like normal backward chaining in that the top level goal is proved by decomposition into simpler subgoals that are eventually proved by direct binding with ground terms. It differs when a subgoal cannot be proved by simple monotonic means. Here, the interpreter can make assumptions under control of a set of "assumable rules."

The interpreter uses the MRS `residue(s)` in place of `truep(s)` to call on the bound mrsforms of the functional concept similar to the test interpreter. `Residue` will return sets of mrsform instantiations, each holding under different assumptions. These assumptions must be arranged in a context tree to ensure the remaining constraints of the functional concept are tested/satisfied under each assumption separately.

The operation of the interpreter is illustrated by considering the generation of one trap position given earlier in Figure 2.3. The resulting reasoning is given as a tree in Figure 2.11.

[6] Approximately 14 hours of unloaded VAX11/750 time for 72 trap red to play positions. It has therefore only been run on the checkers concept trap (see Section 2.2).

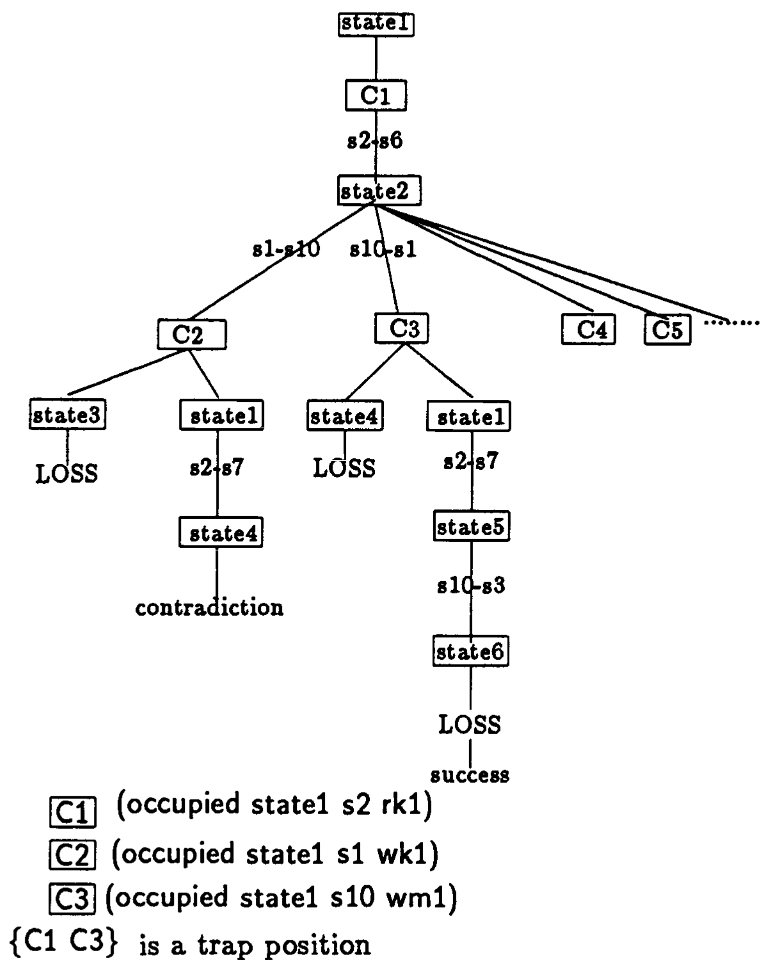


Figure 2.11: Reasoning tree formed during generation of *trap* in Figure 2.3

The generation interpreter is similar to the test interpreter in the way bindings are passed down the search tree and in the way the current concept state is used to build the mrsform for backward chaining. Initially the interpreter is given $side, $state, and &conceptstate bindings as before, with $state being bound to state1 (an empty board position) and $side bound to red[7]. During `residues'` analysis of the first mrsform,

(move $name state1 $newstate $type $from $to red),

64 assumptions are created. One assumption for each king or man on the 32 playing squares. In the example in Figure 2.11, C1 is the assumption that square s2 is occupied by a red king. The set returned from `residues'` analysis of the normal moves associated with this assumption will contain both the legal moves available, s2-s6 and s2-s7. These moves form a conjunctive goal for the interpreter, and they must both be proved correct for the assumption to represent a *trap* position.

First the s2-s6 is explored in state2 (refer to Figure 2.11). At this point the take move constraints will apply, and during the `residues` backward chaining, the interpreter will make assumptions for the white side capturing the red king. A total of 5 assumptions are made[8]:

(occupied state1 s1 wk1) {C2}

(occupied state1 s10 wm1) {C3}

(occupied state1 s10 wk1) {C4}

(occupied state1 s9 wk1) {C5}

(occupied state1 s9 wm1) {C6}

Each of these assumptions satisfies the constraints in the take move and allows the legal capture of the red king. The assumption of a white king in square s2

[7] Only instances for one side are generated— the other side can be computed through symmetry.

[8] All assumptions are made in state1 and framed forward.

is excluded because, in C1, state1 has square s2 occupied by the red king. These restrictions to the assumptions permissable are encoded in the **assumable** rules.

The system continues the forward search under the assumption set {C1 C2} and tests the current board position (state3) against the final constraints of the concept. This succeeds because the current board position is a *loss* for red. The remaining goal of the conjunction, exploring s2-s7, is now invoked under {C1 C2}. The system makes the move s2-s7 forming state4, which has a white king on s1 (from C2) and a red king on s7 (from C1 and s2-s7). The constraint for determining whether there exists a take move over s7 for white is now explored. This fails because of the constraints imposed on global variables. Recall that the variable denoting the square the takemove moves from ($fromfc-1) is globally scoped and can have only one binding. In other words, the trapped piece must be taken by the same piece whichever move it makes. The assumption C2 gave a binding to $fromfc-1 of s1, but the piece on this square cannot take the piece on s7. Hence, the take move constraint fails, and the assumption set {C1 C2} cannot be a *trap*.

A simple chronological backtracking scheme is used to recover from contradictions. Reasoning begins again under C1 with the square s2 occupied by a red king. The next assumption is explored, C3, that places a white man in square s10. The *loss* constraint is satisfied, and when the alternative move s2-s7 is explored, the take move for white is satisfied with the single binding for $fromfc-1 of s10. Thus, the generator has found an assumption set {C1 C3} that represents an example of *trap*.

It can be seen that the generation interpreter reverts to generate and test when dealing with constraints that must be satisfied between different concept states (such as shared variables and global variables). Here the variables are bound in one state and tested in another. There appears to be little that can be done about this, since the tests are separated from the point of generation and, therefore, the

generated results must be propagated to the tests before they can be pruned.

There are, however, steps that can be taken to minimize generate and test within the concept states. Consider the case above when the system is trying to satisfy the constraints of the second concept state of *trap*—the take move mrsform. The take move rule is like the normal move rule in that it has two sub rules: a rule that generates the legal take moves and a rule that makes the moves on the board. The rule that generates the take moves is given below:

```
(if (and (side $player1 $side1)
         (type $player1 $type1)
         (occupied $state $from $player1)
         (opside $side1 $side2)
         (connected $from $over $direct)
         (legal-direction $side1 $type1 $direct)
         (side $player2 $side2)
         (occupied $state $over $player2)
         (connected $over $to $direct)
         (occupied $state $to empty))
  (find-take $state $type1 $from $over $to $player1 $side1)).
```

Normally MRS, or any other logic programming language, solves each subgoal in a conjunction as above in fixed left to right order. This usually works fine, because the rule is always going to be invoked with known variables bound and the programmer can order the clauses appropriately so that generate and test within the conjunction is minimized. In the case of the generation interpreter, the variables bound at the time of invocation are not fixed and could be any combination.

What is needed is a dynamic ordering over the clauses in the conjunction, so at each new subgoal invocation, the one with the minimum number of solutions is used as a generator in preference to any other. This was explored by Smith &

Genesereth (1984). To implement this, it was necessary to change the way MRS handles conjunctive goals. The `residues` backward chainer in MRS employs an agenda of possible solutions. Each of these agenda tasks includes a list of bindings found so far and the list of goals remaining to be proved. The changes made where to treat this list of subgoals, for the take and normal move rules, as sub agendas to be deliberated over.

Consider the case when, in the previous example for *trap*, the white take move in state2 (with a red king in s6) was to be satisfied (see Figure 2.11). The take move mrsform will have the over square bound from the previous move (s7). The clauses are given below with the bindings plugged in and an estimation of the number of solutions.

SUBGOAL	NUMBER OF SOLUTIONS
(side $player1 white)	2
(type $player1 $type)	4
(occupied state2 $from $player1)	64
(opside white $side2)	1
(connected $from s6 $direct)	4
(legal-direction white $type1 $direct)	6
(side $player2 $side2)	4
(occupied state2 s6 $player2)	5
(connected s6 $to $direct)	4
(occupied state5 $to empty)	32

The number of solutions information is computed by first performing `trueps` and caching the size of the set associated with the query. The number of solutions for the side and type predicates is computed with there being two red playing pieces (king and man) and two white playing pieces in the data base. For example, information cached for the legal-direction predicate includes:

(number-of-solutions (legal-direction $side $type $direct) 16)

(number-of-solutions (legal-direction white $type $direct) 6)

(number-of-solutions (legal-direction white king $direct) 4)

(number-of-solutions (legal-direction white man $direct) 2)

The deliberation algorithm is "greedy" and will choose the (opside white $side2) goal to pass to **residue** because it has the fewest number of estimated number of solutions. **Residue** will return possible bindings for the $side2 variable. In this case there is only one: red. With $side2 bound, the number of solutions is recomputed, and both side goals have only two solutions. Following the processing of both side goals and the type goal, four alternative tasks will be on the MRS agenda, corresponding to the four different combinations of king and man. Below we consider the task with a red king and white man.

SUBGOAL	NUMBER OF SOLUTIONS
(occupied state2 $from wm1)	32
(connected $from s6 $direct)	4
(legal-direction white man $direct)	2
(occupied state2 s6 rk1)	1
(connected s6 $to $direct)	4
(occupied state2 $to empty)	32

The (occupied state5 s6 rk1) fact is first trivially satisfied. Next the legal-direction predicate is chosen, which returns two different bindings for $direct. Below we give the agenda task with direction bound to nw (north west).

SUBGOAL	NUMBER OF SOLUTIONS
(occupied state2 $from wm1)	32
(connected $from s6 nw)	1
(connected s6 $to nw)	1
(occupied state2 $to empty)	32

The connected facts are run giving binding for $from s10 and $to s1. The (occupied state2 s1 empty) fact is trivially satisfied leaving the single occupied fact. Residue consults the assumable rules and make the assumption (occupied state1 $from wm1) in C3 satisfying this occupied fact in state2 through frame axioms.

2.3.3 Summary of Wyl's Interpreters

This completes our discussion of Wyl learning functional concepts and applying these concepts to performance tasks.

First we described how Wyl learned new concepts via a two stage process. First the envisionment interpreter is applied to translate the supplied structural instances into functional form. Next the functional instances are generalized inductively to form the functional concepts.

Second we described how these functional concepts get applied to performance tasks. First we described applying the functional concepts as tests of structual instances. Next we described how to use the functional concepts as generators of instances.

We follow with a description of the process that forms more compact structural concept descriptions.

2.3.4 Compaction

The compaction process takes the set of positive structural instances produced by the generation interpreter and creates a new structural concept language in which the instances can be described succinctly. The procedure has been run on the *trap* instances, and produced the structural concept language described in Section 2.2.1.2. and fully given in Appendix B.

It might appear that compaction is unnecessary. After all, given the complete set of positive instances, recognition can be simply achived by a complete lookup. There are, however, many reasons for building a concept language, some of which are listed below:

- The recognition predicates written in the concept language may be more efficient than simple table lookup. This result was found by Quinlan (1986).
- The new terms created may be useful in describing future concepts.
- The knowledge learned by the system is easier to understand (Arbab & Michie 1985).

The problem of new term creation is of great interest to the machine learning community. This thesis attempts to demonstrate a method of learning that does not require highly engineered structural language. Rather the system, beginning with a very simple observational language, can incrementally build a suitable structural language that captures the concepts learned. The results from the *trap* case illustrate the kind of language that would have had to have been engineered and presupplied to allow a traditional structural similarity based learning system, such as $AQ11$ or $ID3$, to succeed. Currently Wyl has only used the concept language in recognition predicates. The issue of further uses of the new terms, such as for learning other concepts, as generalization hierarchies, or in the enumerator, are discussed in the conclusions.

Two new term creation techniques are employed in Wyl. First we describe the method that creates new relational terms by composition of the primitives in the instance langauge. Then we describe in detail the method that creates new descriptive terms by naming common disjunctions. We describe the rationale behind the disjunction method and describe some relevant work. This is followed by a description of the disjunction naming algorithm, called Tax, employed in Wyl.

Relational term creation

Relational terms describe relations between objects in a description. Examples of relational terms in Winston's ARCH are touching, on-top-of, etc. In Wyl, as part of the domain specification, primitive relational terms are supplied, that describe the local relationships between squares. Each square has a relational term that describes its relation to its immediate neighbours. For example s2 is connected to the square s6 in the south west (sw) direction.

Wyl discovers new relational terms that describe the relation between squares in concept instances. For example, in the instance of *trap* given in Figure 2.3, the white man on square s10 is south-2-squares from the red king on square s2. This relationship south-2-squares is defined in terms of the relational primitives. Two squares s_1 and s_2 are connected south-2-squares when the square connected in the sw direction from square s_1 is also connected in the se direction to s_2. This term is defined in the structural language below:

```
(if (and (connected $square-1 $square-3 sw)
         (connected $square-3 $square-2 se))
    (connected $square-1 $square-2 south-2-squares))
```

There is another path from $square-1 to $square-2 that is included as part of the definition:

```
(if (and (connected $square-1 $square-3 se)
```

```
        (connected $square-3 $square-2 sw))
    (connected $square-1 $square-2 south-2-squares))
```

Relational terms are formed in an analogous way to the main Wyl functional learning process. A general functional term is defined that describes the relationship between any two squares on the board by as search. This schema describes the relationship as a breadth first search, moving forward through the board by applying any of the primitive connected relationships. The term south-2-squares is a specialization of this general schema.

To find a new relation between two squares, the general schema is applied to perform a breadth first search through the board from one of the squares in an instance to the other square(s). Each path found is formed into a definition for the new term. In the example above, two paths are found: one through s6, the other through s7. The primitive connected relations making up the path are first formed into a conjunction, for example:

```
(if (and (connected s2 s7 se)
         (connected s7 s10 sw))
    (connected s2 s10 south-2-squares)).
```

Then the rule is generalized by turning constants into variables, to produce the rule given above.

Descriptive term creation

The method that creates new descriptive terms in Wyl works by identifying common disjunctions in the set of instances and naming them. The idea is best illustrated by a simple example taken from Thagard and Holyoak (1985).

$$\forall x MacDonalds(x) \wedge Hamburger(x) \supset Greasy(x)$$

$$\forall x BurgerKing(x) \wedge Hamburger(x) \supset Greasy(x)$$

From this Thagard proposes forming the *inductive* generalization (via the dropping condition rule)

$$\forall x Hamburger(x) \supset Greasy(x).$$

The alternative, employed by Wyl, is to perform a deductive transformation by factoring out the disjunction and naming it

$$\forall x MacDonalds(x) \vee BurgerKing(x) \supset Fastfood(x)$$
$$\forall x Fastfood(x) \wedge Hamburger(x) \supset Greasy(x)$$

This has advantages over the inductive approach. First, becuase of the nature of induction, the generalization will usually be incorrect, especially if not driven by strong bias. Second, the deductive approach allows incremental change as it is truth preserving. Third, the new term can help in the explanation of reasoning.

There is nothing new in this idea. Wolff, in 1982, developed a language acquisition system (called SNPR) that identified common disjunctions in sentences. After seeing ``the dog chased...'' and ``the cat chased,'' SNPR forms the term noun = {dog cat}. In 1983, Quinlan talked of using this method to form new descriptive terms automatically as part of the research with ID3. Many *lost-in-3-ply* chess positions share common arrangements of all but one of the pieces on the board. The remaining piece can be on any of a set of squares. What Quinlan proposed was to name this set of squares to form a new term and then use it to describe the previous set of board positions by reducing them all to one description. This method reduces the number of instances by the size of the set in the disjunction. A similar method is employed by Fu and Buchanan (1985) in the work on forming hierarchies in knowledge bases.

What these systems share is the identification of common disjunctions. Consider a set of instances that are all known to be examples of a concept, CON. Suppose each instance is represented as a conjunction of attribute values. Initially, because

the conjunctions are instances, the attribute values will relate to directly observable values, such as *red*, $s1$, or *king*. In the general case when we have n attributes, an instance can be described as

$$att1(Val_p\ A) \vee att2(Val_q\ A) \ldots att_n(Val_r\ A) \supset CON(A),$$

where $att_m(val_k\ A)$ means the instance A, has attribute att_m, with value val_k (for example att_m could be *color* with val_k being *red*).

A common disjunction of attribute values can be identified by collecting together a set of positive instances that differ in attribute values for only one attribute. This is equivalent to factoring out all the other similarly-valued attribute facts leaving the disjunction of different values for the single attribute. This condition of sets of instances will be referred to as *focus*. In the example below we have a set of instances which have *focus* for the jth attribute:

$$att1(val_p\ A) \vee \ldots \vee att_j(val_\epsilon\ A) \vee \ldots \vee att_n(val_r\ A) \supset CON(A)$$
$$att1(val_p\ D) \vee \ldots \vee att_j(val_\theta\ D) \vee \ldots \vee att_n(val_r\ D) \supset CON(D)$$
$$att1(val_p\ F) \vee \ldots \vee att_j(val_\rho\ F) \vee \ldots \vee att_n(val_r\ F) \supset CON(F)$$
$$att1(val_p\ H) \vee \ldots \vee att_j(val_\sigma\ H) \vee \ldots \vee att_n(val_r\ H) \supset CON(H)$$
$$att1(val_p\ O) \vee \ldots \vee att_j(val_\phi\ O) \vee \ldots \vee att_n(val_r\ O) \supset CON(O)$$

To compact this set of instances requires a two stage process. First the set of values for the j attribute is named and the new term defined:

$$\begin{aligned} &\forall x\ att_j(val_\epsilon\ x) \\ &\vee \quad att_j(val_\theta\ x) \\ &\vee \quad att_j(val_\rho\ x) \\ &\vee \quad att_j(val_\sigma\ x) \\ &\vee \quad att_j(val_\phi\ x) \\ &\qquad \supset att_j(val_{NEW}\ x) \end{aligned}$$

Then the original set of instances is replaced by a single expression that describes the same set:

$$\forall y\ att1(val_p\ y) \vee \ldots \vee att_j(val_{NEW}\ y) \vee \ldots \vee att_n(val_r\ y) \supset CON(y).$$

We have described what a *focus* set is and how it can be used to create a new descriptive term. Now we describe the methods used by Tax and another learning system to locate *focus* sets in a given set of instances. Generally, systems search the space of all subsets looking for those that satisfy the *focus* condition. Once these are found, the new term is created and the defining instances replaced by the new single concept description. The process continues until there are no more subsets with the *focus* condition.

Clearly there is need for search control, because the space being searched, all possible subsets, is too large. We review two approaches to search control, one used in GLAUBER (Langley et al. 1985) and the one used in Tax.

GLAUBER

GLAUBER works in the domain of simple chemical reactions. From instances of particular reactions, it forms general laws. The laws are described in new terms formed by the system by naming disjunctions. Examples of the terms are acid, alkali and salt. Instances are presented in vectors of attribute names and values. For example the following reactions form one of GLAUBER's training sets:

(reacts inputs $\{HCl\ NaOH\}$ outputs $\{NaCl\}$)
(reacts inputs $\{HCl\ KOH\}$ outputs $\{KCl\}$)
(reacts inputs $\{HNO_3\ NaOH\}$ outputs $\{NaNO_3\}$)
(reacts inputs $\{HNO_3\ KOH\}$ outputs $\{KNO_3\}$)

The first term is the predicate name, the second, one of the attribute names, next come the two values that form the input for the reaction, finally the other attribute name and a value that describes the output of the reaction.

GLAUBER does not search for sets that differ in only one attribute (i.e., satisfy the *focus* condition) as this would lead to no new terms being created. Rather, GLAUBER searches for sets that are different in *two* attribute values. To see why allowing two differences rather than one works, we need to follow GLAUBER futher. GLAUBER searchs this space exhaustively by generating all the instantiations of the query (?predicate ?attribute ?value), where ?predicate binds predicate names (such as **reacts**), ?attribute binds attribute names (such as **inputs**) and ?value binds values (such as $NaOH$). The instantiation with the most bindings is used to form the new terms. In the example above, the instantiation,

(**reacts inputs** HCl)

is one that has the most bindings. The other attribute values of the set that satisfy the instantiation are used to create the new terms:

alkali = $\{NaOH\ KOH\}$
salt = $\{NaCl\ KCl\}$

The original instances are rewritten with the new terms and put back in to the instance set. The process continues.

There are two reasons why GLAUBER searches for sets that differ in two values rather than one. First, most obviously, the method of searching for simple focus sets does not work. Secondly, of more relevance, is that there is a relationship between the two inputs and the output that is not apparent in the representation chosen. The output is functionally determined (in the database sense) by the two inputs. Suppose that we have two reactions that are different in one input, because of this functional dependency, the outputs will also be different. Hence, with this representation, it is imposible to get any focus sets. In Tax we apply a technique that solves this problem and allows us to search only for focus sets. Rather than represent the inputs and outputs of a functional dependency in the vector, we replace one of

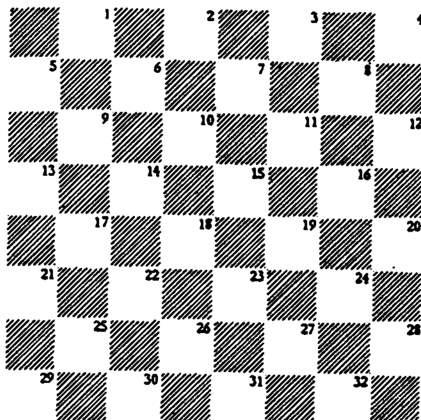


Figure 2.12: Checker board numbering scheme

the terms with the name of the dependency that applies. For example, above we could repace the output with the relationship **reacts-to-form**. In the description of Tax we give another example of this process.

TAX

In this section we describe how Tax compresses a set of positive instances of a concept by searching for and replacing focus sets. We illustrate Tax's operation with an extended example from the checkers concept *trap*. We show Tax compressing a set of 22 trap instances into 3 high level descriptions and creating 4 new descriptive terms.

In Figure 2.12 we give the numbering scheme used in checkers for reference.

In this example we consider only the set of *trap* instances that describe a white piece trapped in squares s13, s21 and s29. We give the instances in vector form

below [9]:

(trap s21 man ne-2-square king) (trap s13 man north-2-square king)
(trap s21 man north-2-square king) (trap s29 man east-2-square king)
(trap s21 man ne-2-square man) (trap s13 man ne-2-square king)
(trap s29 king north-2-square king) (trap s29 man north-2-square man)
(trap s21 king east-2-square king) (trap s13 man ne-2-square man)
(trap s29 king east-2-square king) (trap s29 man ne-2-square man)
(trap s13 man east-2-square king) (trap s29 king north-2-square man)
(trap s13 king east-2-square king) (trap s29 king ne-2-square king)
(trap s21 man north-2-square man) (trap s29 king ne-2-square man)
(trap s29 man north-2-square king) (trap s21 man east-2-square king)
(trap s13 man north-2-square man) (trap s29 man ne-2-square king).

The instances are in vector form, the five tuple has the following attributes: (concept-name trapped-square trapped-piece-type relative-direction trapping-piece). The relative-direction attribute is the relational term created for this instance that describes the relationship between the trapped-square and the square where the trapping piece is. This solves the problem discussed in the GLAUBER section above.

The search method used by Tax is not like that used in GLAUBER because the space of all focus sets soon becomes too large and the algorithm becomes exponential. Rather, Tax[10] locates focus sets in a very directed manner through repeated partitions of the input set. Tax partitions the instance set by attribute values in a fixed order. In this example we first partition the instance set using the values of the trapped-square attribute, then with the trapped-piece-type attribute, and finally

[9] The sets are randomized before Tax is run, for a curious reason. After generation, the instances have a regular order due to the way the primitive generators of the process work. If this ordered set is input to Tax, only some of the descriptive terms are found. This is due to the order imposing artificial patterns on the data that influence Tax.

[10] Tax is a polynomial algorithm.

with the relative-direction attribute.

When Tax partitions on the on the trapped-square attribute, three sets are produced. Tax explores the smallest set first for reasons that will become clear later on in the example. The set for the value s21 is the smallest with only 6 members so it is processed first. Tax now partitions this set of 6 elements on the trapped-piece-type attribute, and again follows the smallest set. This set is a singleton,

(trap s21 king east-2-square king).

In Figure 2.13 we illustrate this partitioning process. Tax backtracks from the singleton and explores the next smallest set (s21 man east-2-square king), which is also a singleton. Backtracking again Tax explores the next set and finds the first *focus* set:

(trap s21 man north-2-square man)
(trap s21 man north-2-square king).

The *focus* set is boxed in Figure 2.13. The user is prompted for a name of the set {man king} and returns anytype. Tax now defines the new term in MRS, by refering to the form of the type primitive:

```
(if (or (type $playing-piece man)
        (type $playing-piece king)
    (type $playing-piece anytype)).
```

Tax returns and explores the other set where the relative-direction attribute has the value ne-2-square. Here Tax encounters the same set {man king}. This time, because a name is known for this set, the value anytype is returned.

Tax now returns to the relative-direction attribute. The state of the search is illustrated in Figure 2.14.

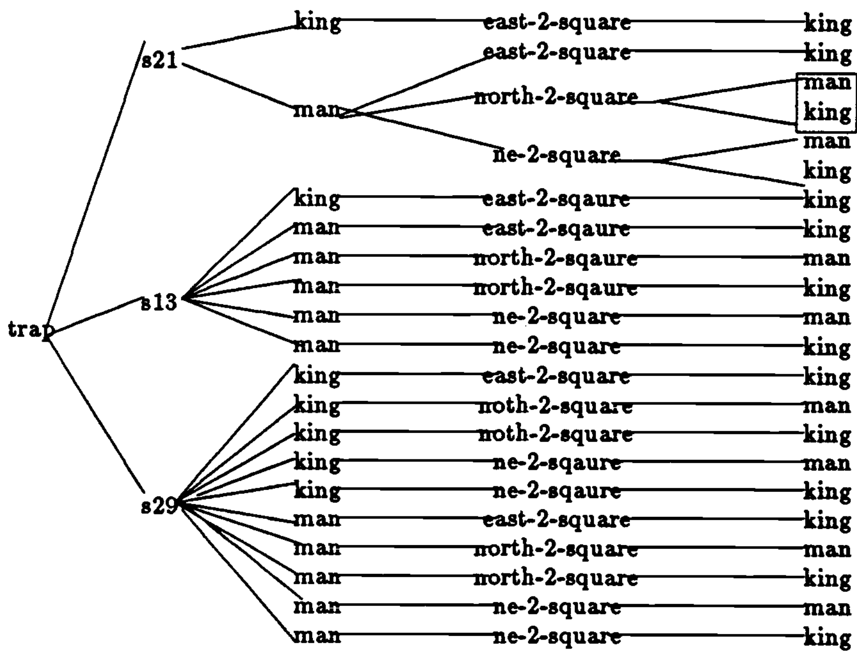


Figure 2.13: Tax identifies first *focus* set

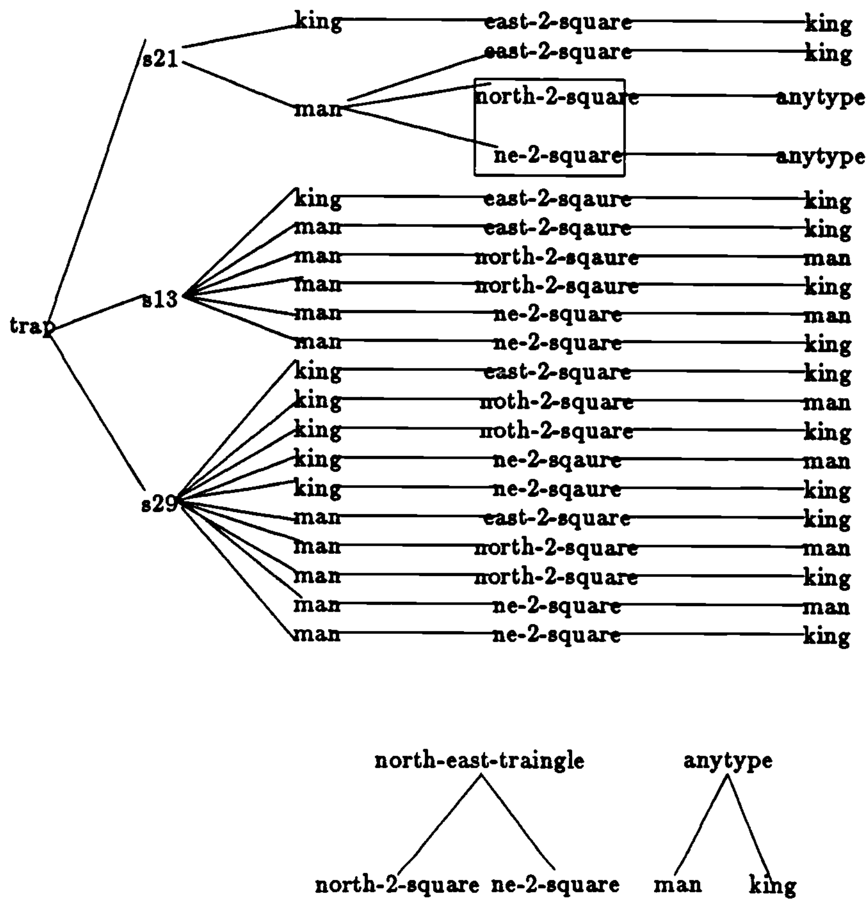


Figure 2.14: Tax finds second a new term, over the relative-direction attribute

We come to a key point in the operation of Tax that illustrates how focus sets are found for attributes other than the last. In the Figure 2.14 a *focus* set is boxed given below:

(trap s21 man north-2-square anytype)

(trap s21 man ne-2-square anytype).

Once Tax has explored all subsets below an attribute, as in this case with the attribute relative-direction, the new sets returned are repartitioned. The set returned from below north-2-square was (anytype) while the set returned from ne-2-square was (anytype). The sets returned are combined with their respective attribute values to form partial vectors:

(north-2-square anytype)

(ne-2-square anytype).

These partial vectors are partitioned on all attribute values other than the first. In this case the partitioning gives only one set {north-2-square ne-2-square}. The user is queried and the set is named north-east-triangle. Tax defines the new term in MRS:

```
(if (or (connected $s1 $s2 north-2-square)
        (connected $s1 $s2 ne-2-square)
    (connected $s1 $s2 north-east-triangle)).
```

The results of this partitioning will find *focus* sets for the current attribute for two reasons. First, because the values of all the attributes prior to the current one (in this case concept-name trapped-square trapped-piece-type) will have the same

values due the partitioning on the way down the tree. Second the remaining attributes (in this case trapping-piece) will have the same values because of the recent partitioning. The next level will further clarify this process.

Tax now returns to the trapped-square and, because there are no more sets to explore, the partial vectors are constructed. The king value had one set returned, while the man value has two sets returned. The partial vectors are given below:

(king east-2-square king)
(man east-2-square king)
(man north-east-triangle anytype).

These sets are repartitioned on the (relative-direction trapping-piece) attributes. The set {man king} is rediscovered, and the following sets are returned to the trapped-square level:

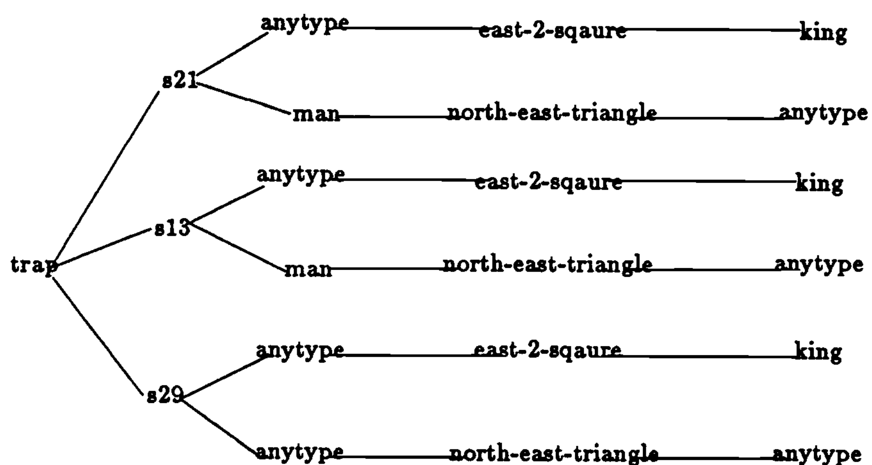
(anytype east-2-square king)
(man north-east-triangle anytype).

Let us summarize the operation of Tax so far. Tax has processed 6 of the trap instances and compressed them to 2 descriptions given below:

(trap s21 anytype east-2-square king)
(trap s21 man north-east-triangle anytype).

The second description covers 4 of the original instances and states that a white man on s21 is *trapped* if there is a red man or king on the squares directly north-2-square or ne-2-square (squares s13 s14).

In Figure 2.15 we show the result of Tax processing the remaining subtrees under the trapped-square attribute (s13 s29). Tax is now at the position to repartition the



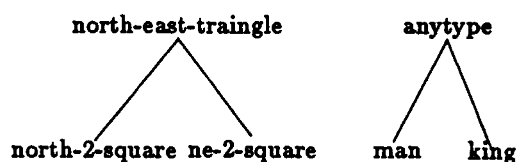


Figure 2.15: Tax, ready to repartition to find trapped-square terms

sets to locate *focus* sets for the trapped-square attribute. The partial vectors are given below:

```
(s21 anytype east-2-square king)
(s21 man north-east-triangle anytype)
(s13 anytype east-2-square king)
(s13 man north-east-triangle anytype)
(s29 anytype east-2-square king)
(s29 anytype north-east-triangle anytype).
```

It is here that we see hierarchies of new terms being constructed. Note there are 3 distinct vector values for the last three attribues:

```
(anytype east-2-square king)
(man north-east-triangle anytype)
(anytype north-east-triangle anytype).
```

Each of these will lead to a set of trapped-square values being defined. Tax sorts these sets smallest first before the user is asked to name them. Hence, the first set the user is asked to name is {s21 s13}, which is named west-center-side. This set describes squares where only a man can be trapped by a king or a man in directions east-2-square or ne-2-squares (man north-east-triangle anytype). The trapped-square s29 has a different partial vector that describes the same kind of traps, but includes a king as a trapped-piece-type (anytype north-east-triangle anytype). Tax first defines the set west-center-side named above:

```
(if (or (sq $square s13)
        (sq $square s21)
   (sq $square west-center-side)).
```

Now Tax finds the set {s21 s29 s13} (from partial vectors (anytype east-2-square king)) to be named by the user. Before Tax offers this set or any other for naming, a search is made to see if the set has any known subsets. In this case there is a subset known and Tax replaces the set {s21 s13} by its definition. Hence, when the user is asked, the set offered is {west-center-side s29}. Tax builds the definition of this term as a disjunction of the term west-center-side and the value s29:

```
(if (or (sq $square west-center-side)
        (sq $square s29)
    (sq $square west-south-side)).
```

The reason for searching the sets smallest first on the way down recursion can now be understood. By searching for *focus* sets smallest first, new terms for attributes are defined smallest first, hence, new term hierachies are built bottom up. If the sets where identified in the opposite order or in an arbitrary order, the problems of hierachy reformation experienced by Utgoff (1983) would need to be dealt with. By ordering, building heirarchies is *incremental*—a new term definition never requires the reformation of a previously defined term.

It is apparent that the attribute order is important to the results of the algorithm. Different terms are created depending upon the supplied order. Some initial experiments have been run to develop some feel for mechanical means of determining the ideal attribute order. The order does not effect the truth preserving nature of the algorithm, only some orders give a more succinct description or create more intuitive, already known terms. The order given here and in the example of Tax running on the full *trap* set produced the well known terms. In all 13 new terms were discovered and the final structural *trap* description was captured in 12 disjunctions.

2.4 Chapter Summary

To summarize, in this chapter we have presented a detailed description of the program Wyl. The description included a trace of Wyl learning the checkers concept *trap*.

The next chapter gives examples from checkers of Wyl compiling the concept *trap* and learning the concept *trap-in-2-ply*. In chess we show a Wyl learning two concepts: *knight-fork* and *skewer*.

Chapter 3

Examples of Wyl Learning

In this section we illustrate Wyl learning four different concepts, two in checkers and two in chess. The first lesson is the checkers concept *trap* described in Chapter 2. Next we teach the concept *trap-in-2-ply*, which relies on Wyl 's previous lesson. Next Wyl learns two concepts in the domain of chess. Both concepts describe stratagems to win the queen by forcing the king to move out of check. The first concept *knight-fork* captures the queen by exploiting the power of the knight move and simultaneously threatening the king and queen. The second concept *skewer*, employs a different technique. The king is threatened by a piece, thereby forcing it to move out of check, the threatening piece now captures the queen which was sheltered behind the king.

In the description of Wyl learning the first concept *trap* we emphases the compilation stage as the learning stage was discussed in detail in Chapter 2. We give a dialogue of Wyl compiling *trap* and timed recognition tasks before and after compilation. In each of the remaining concept learning descriptions we emphases the training session. Each description begins with Wyl learning the concept from two well chosen examples, next the new concept definition is tested against some "near miss" negative instances and more positive examples. Each description of the training sessions include the training instances illustrated as board positions and a first

order logical formalization of the concept learned. In addition, for the checkers concept *trap-in-2-ply*, we give an annotated dialogue with Wyl and the machine representation of the concept.

3.1 Lesson 1: Checkers concept *trap*

In Chapter 2 we illustrated Wyl by describing in detail how the concept *trap* was learned. Here we give an annotated dialogue with Wyl during the compilation stage once the functional concept is learned. The dialogue includes timing information as a guide to the computational resources required for the current implementation. The program runs on a Vax-11/750 during periods of low activity.

```
Script started on Thu Apr 10 15:01:33 1986
.......
-> (Wyl)
.......
Domain: checkers, Concept name:trap

Thu Apr 10 15:11:15 1986 Start learning loop 3.
state name:q
          Who plays first?:red
          Max search depth?:2
     Example number:1
          Square:s2
          Playing piece:rm1
          Square:s10
          Playing piece:wm1
          Square:q
```

```
    Domain State:state-24

Domain: checkers, Concept name:trap
  Thu Apr 10 15:11:39 1986 Testing instance with current theory.
  Thu Apr 10 15:15:00 1986 Example number 1 classified as positive.

Is this an example of the concept?yes
```

Wyl tests the functional concept definition with a position very like the second training instance given in Chapter 2 (Figure 2.3), the only difference is that the white king is replaced by a white man. Recognition takes 3 minutes, 21 seconds.

```
Should Compilation commence?yes
Thu Apr 10 15:15:06 1986 Begin Compilation.
```

Wyl checks if compilation should begin whenever the functional concept definition correctly classifies an instance.

```
  Thu Apr 10 15:15:23 1986 Begin Enumeration.
    Thu Apr 10 16:19:24 1986 Total instances found :0
    Thu Apr 10 16:41:18 1986 Total instances found :1
    Thu Apr 10 16:44:46 1986 Total instances found :2
    Thu Apr 10 18:20:42 1986 Total instances found :6
                    ................
    Fri Apr 11 04:47:16 1986 Total instances found :65
    Fri Apr 11 05:08:06 1986 Total instances found :67
    Fri Apr 11 05:31:15 1986 Total instances found :69
    Fri Apr 11 05:56:25 1986 Total instances found :72
```

The enumeration stage is very time consuming, taking a total of 14 hours, 41 minutes. Note only one white traps are generated.

```
Should code generation commence?yes
  Fri Apr 11 06:05:43 1986 Begin code generation.
  Fri Apr 11 06:09:41 1986 Building the vector forms of the instances.
```

The states generated are translated into a vector form suitable for the main compaction routine—Tax. It is at this stage that the new relational terms are created such as north-2-square.

```
  Fri Apr 11 06:41:46 1986 Begin randomize of vectors.
```

The vectors are randomized before Tax is called for reasons given in section 2.3.4

```
  Fri Apr 11 06:51:21 1986 Enter Tax.
```

Tax queries the user for a name for each focus set identified.

```
Classification set:{king man}? anytype
Classification set:{ne-2-square north-2-square}? north-east-triangle
Classification set:{south-2-square se-2-square}? south-east-triangle
Classification set:{south-2-square sw-2-square}? south-west-triangle
Classification set:{s30 s31}? south-center-side
Classification set:{s2 s3}? north-center-side
Classification set:{s12 s20}? east-center-side
Classification set:{s21 s13}? west-center-side
Classification set:{west-center-side south-single-corner}? west-single-
side
Classification set:{east-center-side north-single-corner}? east-single-
side
Classification set:{s23 s7 s11 s22 s19 s14 s10 s15 s18 s6}? center
```

The terms constructed are illustrated in Figure 3.1 below

```
Fri Apr 11 06:54:12 1986 writing out code generated.
Fri Apr 11 06:54:14 1986 Translate new terms into MRS code.
Fri Apr 11 06:56:20 1986 Finished!!
```

*Wyl has generated recognition predicates written in MRS code that define trap. We now test these structural descriptions with two of the original trap positions. First the position (*state-24*) used above in learning loop 3, red to play with a red man on* 2 *and white man on* 10. *Although the recognition predicates where generated using white trapped positions, the concept descriptions can be used to recognize both white and red postitions as Wyl uses symmertry.*

```
Domain checkers, Concept: trap

Fri Apr 11 06:56:21 1986 Start learning loop 4.
state name:state-24
          Who plays first?:red
          Max search depth?:2
     Domain State:state-24

Domain checkers, Concept: trap
  Fri Apr 11 06:58:32 1986 Testing instance with current concept.
  Fri Apr 11 06:58:38 1986 Example number 4 classified as positive.

Is this an example of the concept?yes
```

The recognition task is much faster with structural descriptions, 6 seconds compared with 201 seconds for functional descriptions.

```
continue with next example ?:yes
```

```
Domain checkers, Concept: trap

Fri Apr 11 07:02:03 1986 Start learning loop 5.
state name:q
          Who plays first?:white
          Max search depth?:2
     Example number:5
          Square:s23
          Playing piece:wm1
          Square:s15
          Playing piece:rm1
          Square:q
     Domain State:state-458
```

This position illustrated in Figure 2.2.

```
Domain checkers, Concept: trap
  Fri Apr 11 07:03:45 1986 Testing instance with current theory.
  Fri Apr 11 07:03:54 1986 Example number 5 classified as positive.
```

In Figure 3.1 we illustrate the new terms defining sets of squares, constructed by the compilation stage above. Each term describes an area of the board that is important in describing the concept *trap*. For example, the term south-center-side names a set that is useful in describing *trap* because s30 and s31 are the only two squares where both a white king and man can be trapped by a red king. The squares in the center are useful in describing all *trap* positions where a white man can be trapped by either a red man or red king. The "single corner" squares (s4, s29) are especially important as they are the only squares on the board where you can be trapped three different ways. The only squares not to be included in any

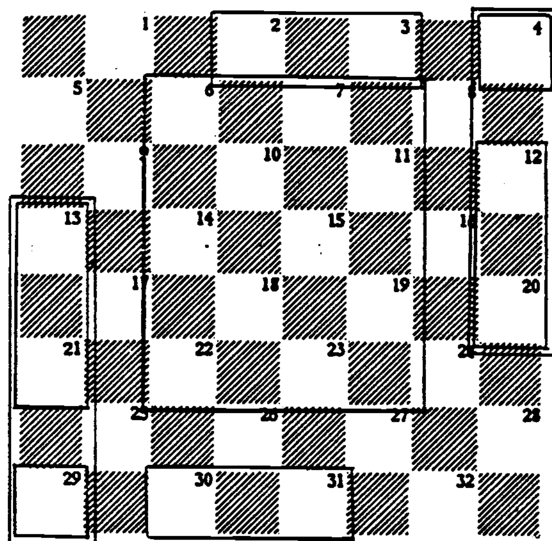


Figure 3.1: Structural concept terms for checkers *trap*

terms are equally important: The so called "double corner" squares (s1, s5 and s28, s32) describe squares in which you cannot be trapped.

This concludes the review of Wyl compiling the concept *trap*.

3.2 Lesson 2: Checkers concept *trap-in-2-ply*

The first training instance is given in Figure 3.2, with red to move. The red king on square 16 has four options, two of these lead to an immediate loss (16-11, 16-19), the other two lead to a trap postion (16-12 then 15-11 and *trap*, or 16-20 then 15-19 and *trap*). Wyl analyses this position and determines that the outcome is an unavoidable *trap* for red in 2 ply. The moves which lead to the loss are pruned from the functional instance because Wyl considers *trap* better than *loss*. Although *trap* is known to be a kind of *loss*, goals which delay an immediate loss are prefixed.

The second training instance is given in Figure 3.3, with white to move. In this position the pieces move towards each other, either way the white man moves, the red king can *trap*.

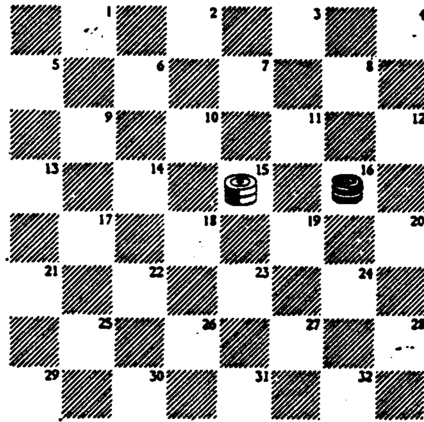


Figure 3.2: Training Instance 1: Checkers concept *trap-in-2-ply*

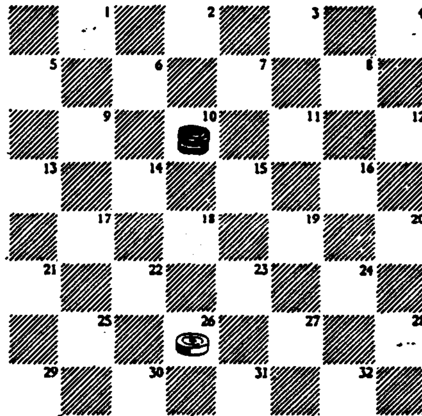


Figure 3.3: Training Instance 2: Checkers concept *trap-in-2-ply*

The final concept learned from these two training instances is given below in logic. A state (*state1*) with side (*side1*) to move is a *trap-in-2-ply* iff:

- *All* legal operators available to *side1* are *normalmoves* and lead to a position which
- *there exists* a move leading to a position in which
- is a *trap* for *side1*.

$$
\begin{aligned}
&\forall\ state1\ side1\ 2PlyToTRAP(state1\ side1) \Leftrightarrow \\
&\quad \forall\ move1\ state2\ side2\ from1\ to1\ type1 \\
&\qquad oppositeplayer(side1\ side2) \\
&\qquad \wedge legalmove(state1\ side1\ move1) \supset \\
&\qquad\quad normalmove(state1\ move1\ state2\ from1\ to1\ side1\ type1) \\
&\qquad \wedge\ \exists\ move2\ state3\ from2\ to2\ type2 \\
&\qquad\quad normalmove(move2\ state2\ state3\ from2\ to2\ side2\ type2) \\
&\qquad\quad \wedge TRAP(state3\ side1)
\end{aligned}
$$

The terms in this description are described fully in Chapter 2. The *legalmove* predicate generates *all* legal moves while the *normalmove* predicate only generates none take moves consistent with any bindings.

The concept definition is tested with two more positions. First we present a negative instance illustrated in figure 3.4.

In this case the red king on square 6 is not in a *trap-in-2-ply* position because it can move to square 1 where it cannot be trapped. This method of avoiding a *trap* by retreating into the "double corner" (squares 1, 5 and 28, 32) is one of the first lessons learned by any checker player.

Wyl correctly identifies the position as a negative instance of *trap-in-2-ply*. The proof fails at the sub goal,

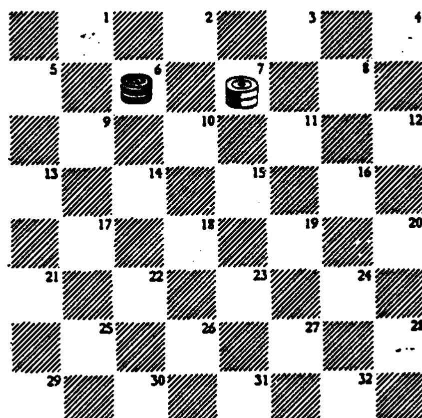


Figure 3.4: Test Instance 1: Checkers concept *trap-in-2-ply*

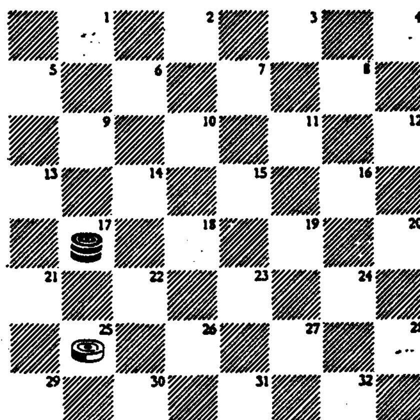


Figure 3.5: Test Instance 2: Checkers concept *trap-in-2-ply*

$$\exists\, move2\ state3\ \ from2\ to2\ type2$$
$$normalmove(move2\ \mathrm{state12}\ state3\ from2\ to2\ \mathrm{white}\ type2)$$
$$\wedge TRAP(state3\ \mathrm{red}),$$

where state12 is the position following red's move to square 1— there are no moves for white that lead to a *trap*.

The second test instance is a positive example of the concept and serves to demonstrate (1) that the concept has been correctly learned and (2) that the concept instances have diverse structural representation. The instance is illustrated in figure 3.5, with white to move.

Here the white man must move into square 21 to avoid capture, only to to

trapped by the red king moving into square 22.

3.2.1 A dialogue with Wyl

In this section we give the dialogue with Wyl learning the concept *trap-in-2-ply* using the training session detailed above.

```
Script started on Sat Apr 12 14:52:15 1986
1:mrs


MRS Version 7.1 in Franz Lisp, Opus 38 created 1-24-84
-> (load 'load.load)
t
-> (Wyl)
continue with new domain ?:y
DOMAIN:checkers
checkers/fc-cache.mrs
checkers/board.mrs
checkers/meta.mrs
checkers/legal.mrs
checkers/players.mrs
checkers/playmoves.mrs
checkers/state-0.mrs
checkers/trap-structural.mrs
checkers/relation.mrs
```

Wyl loads the functional and structural definition of the checkers domain

```
Concept name?:trap-in-2-ply
```

```
Domain: checkers, Concept: trap-in-2-ply


Sat Apr 12 14:56:24 1986 Start learning loop 1.
state name:q
          Who plays first?:red
          Max search depth?:2
     Example number:1
          Square:s16
          Playing piece:rk1
          Square:s15
          Playing piece:wk1
          Square:q
     Domain State:state-2
```

The first training instance (figure 3.2) is tested against the current (non existent) concept definition.

```
Domain: checkers, Concept: trap-in-2-ply
  Sat Apr 12 14:57:33 1986 Testing instance with current concept.
  Sat Apr 12 14:57:33 1986 Example number 1 classified as negative.


Is this an example of the concept?yes


Domain: checkers, Concept: trap-in-2-ply
  Sat Apr 12 14:57:42 1986 Current concept being generalized.
    Sat Apr 12 14:57:42 1986 Performing domain state search.
```

The envisionment stage, builds the complete min/max search tree.

```
    Sat Apr 12 15:11:09 1986 Walking over the tree trace
```

The sufficient and necessary conditions are extracted from the complete explanation

```
    Sat Apr 12 15:12:16 1986 Generalizing search trace.
```

The first stage of generalization, the functional instance is compressed to a single generalized sequence of operators.

```
    Sat Apr 12 15:12:59 1986 Matching generalized trace with current concept
```

The final stage of generalization, in this case trivial as there is no previous concept definition

```
    Sat Apr 12 15:14:32 1986 Walking the concept.
```

Translates the result of the generalizer (in the frame/slot/value representation discussed in section 2.3.1.2) to the binding list representation used by the test and generator interpreters (section 2.2.2.1).

```
continue with next example ?:y


Domain: checkers, Concept: trap-in-2-ply


Sat Apr 12 15:20:04 1986 Start learning loop 2.
state name:q
          Who plays first?:white
          Max search depth?:2
     Example number:2
          Square:s10
          Playing piece:rk1
          Square:s26
          Playing piece:wm1
          Square:q
```

```
    Domain State:state-41

Domain: checkers, Concept: trap-in-2-ply
  Sat Apr 12 15:21:46 1986 Testing instance with current concept.
  Sat Apr 12 15:22:51 1986 Example number 2 classified as negative.
```

This training instance is illustrated in Figure 3.3

```
Is this an example of the concept?yes

Domain: checkers, Concept: trap-in-2-ply
    Sat Apr 12 15:23:37 1986 Walking over the tree trace
    Sat Apr 12 15:23:39 1986 Generalizing search trace.
    Sat Apr 12 15:23:40 1986 Matching generalized trace with current concept
    Sat Apr 12 15:23:51 1986 Walking the concept.

continue with next example ?:y

Domain: checkers, Concept: trap-in-2-ply

Sat Apr 12 15:29:41 1986 Start learning loop 3.
state name:q
          Who plays first?:red
          Max search depth?:2
     Example number:3
          Square:s6
          Playing piece:rk1
          Square:s7
          Playing piece:wk1
```

```
        Square:q
    Domain State:state-47

Domain: checkers, Concept: trap-in-2-ply
  Sat Apr 12 15:31:02 1986 Testing instance with current concept.
  Sat Apr 12 15:32:19 1986 Example number 3 classified as negative.
```

This test instance is illustrated in Figure 3.4

```
Is this an example of the concept?no

continue with next example ?:y

Domain: checkers, Concept: trap-in-2-ply

Sat Apr 12 15:34:24 1986 Start learning loop 4.
        Who plays first?:white
        Max search depth?:2
    Example number:4
        Square:s17
        Playing piece:rk1
        Square:s25
        Playing piece:wm1
        Square:q
    Domain State:state-56

Domain: checkers, Concept: trap-in-2-ply
  Sat Apr 12 15:36:03 1986 Testing instance with current concept.
  Sat Apr 12 15:37:15 1986 Example number 4 classified as positive.
```

This test instance is illustrated in Figure 3.5

```
Is this an example of the concept?yes


continue with next example ?:no
continue with new concept ?:no
continue with new domain ?:no
->(exit)
```

The machine representation of the concept *trap-in-2-ply* is given below and illustrated as a tree in Figure 3.6:

```
(checkers/trap-in-two-ply-0 normal-move
                       tstate-0 final-constraints
                             (($quantification . for-all)
                             ($next-theory-node . nextfc-0)
                             ($type . $typefc-2)
                             ($from . $fromfc-2)
                             ($to . $tofc-2)
                             (t . t)) )
(checkers/trap-in-two-ply-0 normal-move
                       nextfc-0 final-constraints
                             (($quantification . there-exists)
                             ($next-theory-node . nextfc-1)
                             ($type . $typefc-1)
                             ($from . $fromfc-1)
                             ($to . $tofc-1)
                             (t . t)) )
(checkers/trap-in-two-ply-0 terminate-state
```

```
nextfc-1 final-constraints
        (($result . trap)
        (t . t)) )
```

This completes lesson 2, we follow with two more lessons in chess.

3.3 Lesson 3: chess concept *knight-fork*

The first training instance is illustrated in Figure[1] 3.7, by with white to play. The red knight on square c2 (refer to Appendix A, Figure 6.1 for board notation) threatens both the queen on a1 and the king on e1. White is forced to move the king out of check, allowing the knight to capture the queen.

In the second instance illustrated in figure 3.8 it is red's turn to loss the queen. The white knight on square d6 threatens both the queen and the king. As there is no alternative other than to move the king to avoid check, the queen is lost.

Following these two training instances Wyl learns the following description:

$\forall$ $st1\ side1\ KNIGHTFORK(st1\ side1) \Leftrightarrow$

$\quad\forall$ $m1\ st2\ side2\ knightsq\ kingsq\ dir1\ type1$

$\quad\quad oppositeplayer(side1\ side2)$

$\quad\wedge[\exists\ st3\ m2\ dir2\ Move(sm2\ st1\ st3\ knightsq\ kingsq$ take knight $dir2\ side2$ king$)]$

$\quad\wedge LegalMove(st1\ side1\ m1)$

$\quad\quad\supset\ Move(m1\ st1\ st2\ kingsq\ tonew2$ nontake king $dir1\ side1$ nil$)$

$\quad\wedge\ \exists\ m3\ st4\ dir3\ queensq$

$\quad\quad Move(m3\ st2\ st4\ knightsq\ queensq$ take knight $dir3\ side2$ queen$)$

$\quad\wedge LOSEQUEEN(st4\ side1)$

---

[1]The user may instruct Wyl to ignore certain playing pieces that are irrelevant to the concept to reduce the size of the search space during envisionment.

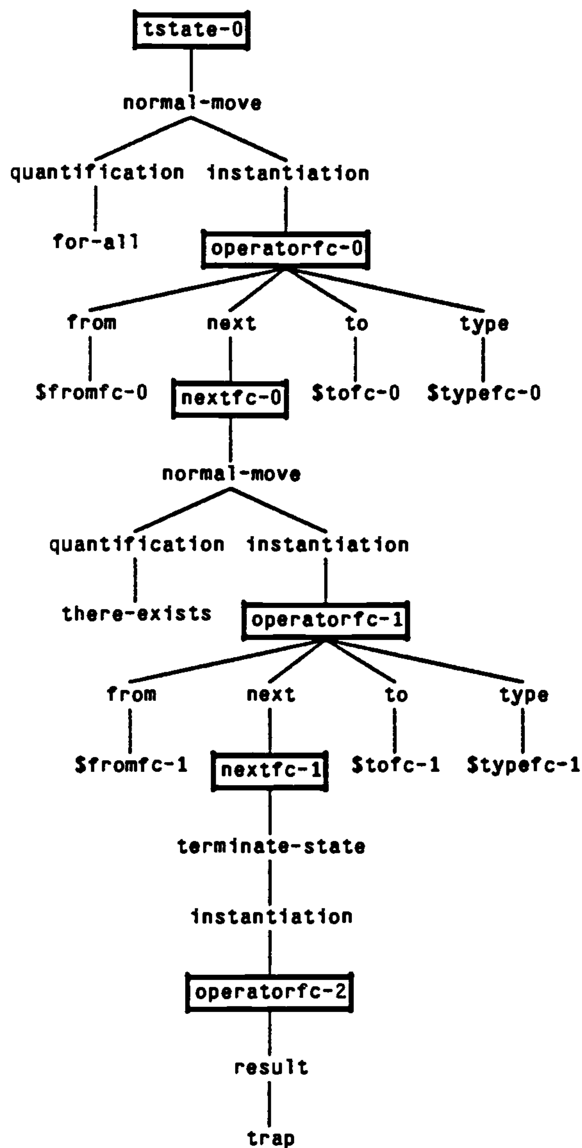


Figure 3.6: Functional Concept *trap-in-2-ply*

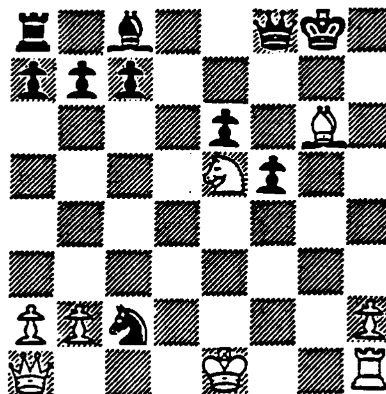


Figure 3.7: Training Instance 1: Chess concept *knight-fork*

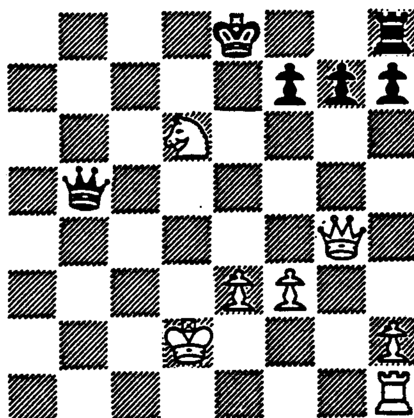


Figure 3.8: Training Instance 2: Chess concept *knight-fork*

The terms in this concept are fully described in Appendix A. They are similar to those used in checkers previously. The *LegalMove* predicate generates all legal moves available to side *side*1 in state *state*1, while the *Move* predicate generates only those moves consistent with any bindings. The implication between the *LegalMove* predicate and the *Move* predicate is used to encode the notion of *forced move* discussed in Chapter 2 and illustrated by the first test instance.

The *Move* predicate describes the legal moves of a chess game. It has the following form:

Move(*name st newst fromsq tosq kind typemoved direct side typetaken*)

Where *name* refers to the refied name of the move, *st* refers to the current state, *newst* refers to the next state formed by applying the move, *fromsq* is the square the move is from, *tosq* is the square the move is to, *kind* is either take or nontake, *typemoved* refers to the type of piece (eg, knight. king etc.) moved, *direct* is the direction the move is in (eg, ne. north etc.), *side* is red or white, and finally *typetaken* refers to the piece taken (if any). For example the final knight move of the instance illustrated in figure 3.7 taking the queen could be described;

Move(move34 state23 state25 c2 a1 take knight wsw red queen).

The first ∃ clause of the concept definition above makes explicit the fact that

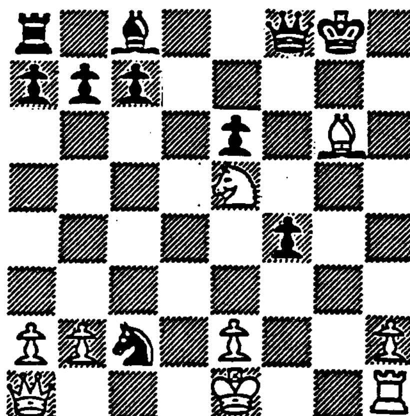


Figure 3.9: Test Instance 1: Chess concept *knight-fork*

the king is currently in check. It describes a move for the opposite player (*side*2) which can take the king. This test for check is included in the definition of the chess recursive schema.

The concept definition can be summarized as:

A state (*st*1) with side (*side*1) is in a *knightfork* iff:

- *side*1 is in check from a knight (on square *knightsq*).
- All current legal moves involve moving the king (on *kingsq*), leading to state *st*2.
- *For all* resulting states (*st*2), *there exists* a move of the knight from *knightsq*, to *queensq*, that captures the queen.

The concept is tested with three more instances. First we present a negative instance illustrated in figure 3.9.

This instance is very similar *structurally* to the first training instance. Where it differs is the black pawn on square f5 has been moved one square forward to square f4. This small change in the structural representation leads to a great change in the functional representation. In this position, because the pawn is no longer blocking the move, the bishop in square g6 can take the knight in c2. Hence, the king is no longer *forced* to move out of check and the concept description does not apply.

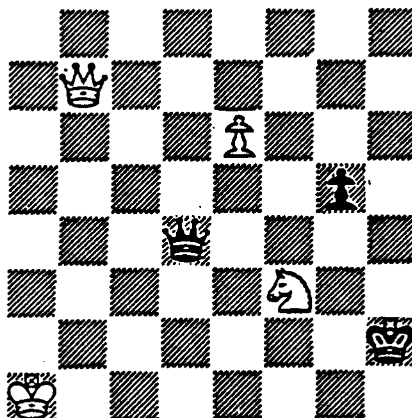


Figure 3.10: Test Instance 2: Chess concept *knight-fork*

The test for check was satisfied with *kingsq* bound to e1, but failed at the subgoal,

$\forall$ *m*1 *st*2 *tonew*1

LegalMove(state12 white *m*1)

$\supset$ *Move*(state12 *state*2 *m*1 e1 *tonew*1 nontake king white nil),

where state12 is the original training instance. This failure is due to the implication failing, the moves of the king from e1 (the *Move* predicate) do not describe *all* legal moves generated from the *LegalMove* predicate.

The second test instance, illustrated in Figure 3.10, is a simple positive example with white to play, to check that the concept was learned correctly. The knight in square f3 forks both the king and the queen.

The third test instance (in Figure 3.11) is a negative example that illustrates a different way in which the training instance can fail a concept definition. In this case it is red to play. The king is not in check, hence, the constraint

$\exists$ *st*3 *m*2 *dir*2

Move(*m*2 st23 *st*3 *knightsq kingsq* take knight *dir*2 white king)

fails, and the instance is classified negative.

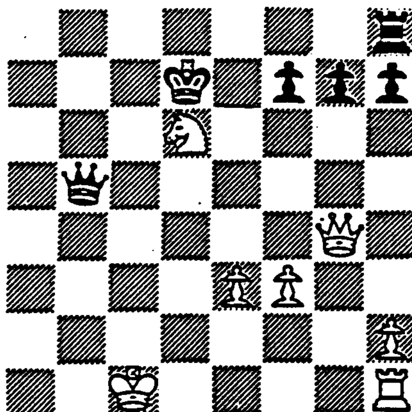


Figure 3.11: Test Instance 3: Chess concept *knight-fork*

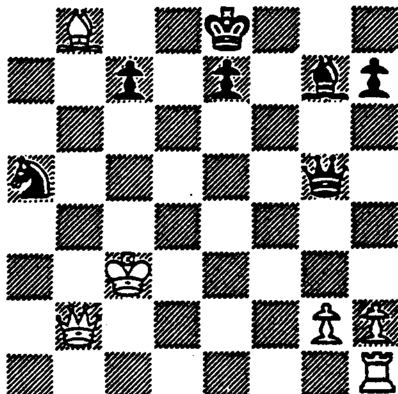


Figure 3.12: Training Instance 1: Chess concept *skewer*

3.4 Lesson 4: chess concept *Skewer*

Skewer is a stratagem in chess to capture the queen using pieces other than the knight. The first training instance is illustrated in Figure 3.12. The red bishop in square g7 is checking the white king. White is *forced* to move out of check, exposing the queen to capture by the same bishop.

The second instance is illustrated in figure 3.13. In this position it is the rook in square c1 threatening the king and capturing the queen.

Following these two training instances, Wyl learns the following description:

$\forall$ *st*1 *side*1 $SKEWER$(*st*1 *side*1) $\Leftrightarrow$

$\forall$ *m*1 *st*2 *side*2 *threatsq threatdir dir*1 *kingsq type*1

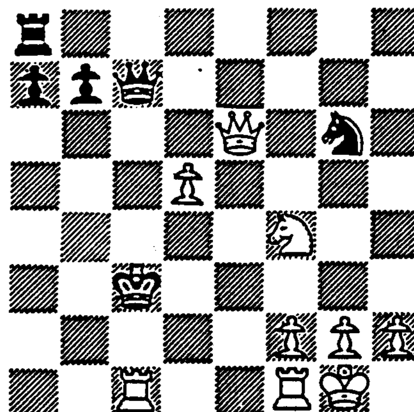


Figure 3.13: Training Instance 2: Chess concept *skewer*

$$
\begin{array}{l}
\quad opposite player(side1\ side2) \\
\wedge[\exists\ st3\ \ m2\ type2 \\
\qquad Move(m2\ st1\ st3\ threatsq\ kingsq\ \text{take}\ type2\ threatdir\ side2\ \text{king})] \\
\wedge LegalMove(st1\ side1\ m1) \\
\quad \supset Move(m1\ st1\ st2\ kingsq\ to2\ \text{nontake king}\ dir1\ side1\ \text{nil}) \\
\qquad \wedge\ \exists\ m3\ st4\ queensq \\
\qquad\quad Move(m3\ st2\ st4\ threatsq\ queensq\ \text{take}\ anytype1\ threatdir\ side2\ \text{queen}) \\
\qquad \wedge LOSEQUEEN(st4\ side1)
\end{array}
$$

Skewer is a very similar to *knightfork*, and can be summarized as:
A State (*st*1) with side (*side*1) to play is in a *skewer* iff:

- *side*1 is in check from a piece (on square *threatsq*) in direction *threatdir*.
- All current legal moves involve moving the king (on *kingsq*) out of check, (leading to state *st*2).
- *For all* resulting states (*st*2), *there exists* a move of the piece from *threatsq* square, to *queensq* in direction *threatdir*, that captures the queen.

The concept is tested with three more instances. First we present a negative instance Figure 3.14, that is very similar *structurally* to the initial training instance.

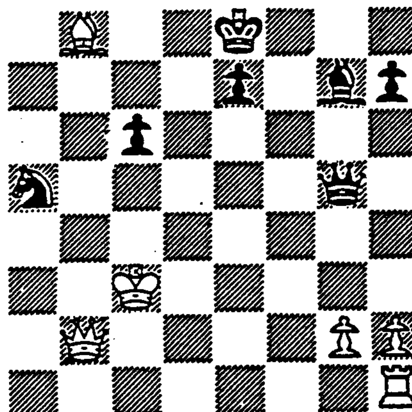


Figure 3.14: Test Instance 1: Chess concept *skewer*

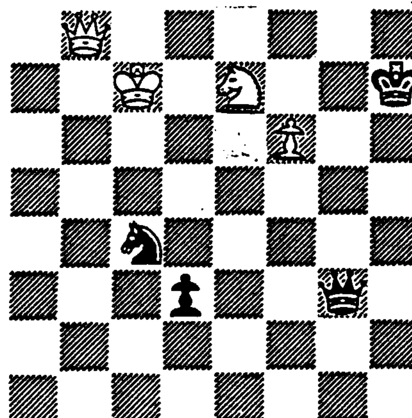


Figure 3.15: Test Instance 2: Chess concept *skewer*

The pawn on square b7 has moved forward one square allowing the bishop in square b8 to block the threat of check.

The second test instance (Figure 3.15) demonstrates that Wyl has correctly learned the concept. Here a queen is used to threaten the opponents queen.

In third test instance (Figure 3.16) the position is not a skewer because the knight can take the threatening rook.

This concludes our illustration of Wyl learning.

Figure 3.16: Test Instance 3: Chess concept *skewer*

Chapter 4

Analysis and Conclusions

In Chapter 1 we identified constraints that inductive learning and recognition impose on representation and noted that these constraints often conflict. We introduced a learning system that satisfies both constraints by employing two representations. In Chapter 2 we described in detail the operation of the implemented system, Wyl. In Chapter 3 we gave examples of Wyl learning concepts in chess and checkers.

In this Chapter, we give a summary of the thesis and detail the lessons we have learned. The first section gives a brief critique of Wyl. The second section gives some pointers for future research questions. In the final section we summarize the main contributions of this research.

4.1 Summary

In the summary of Wyl we first develop a criterion to measure the effectiveness of learning systems in general. We apply these criteria to evaluate Wyl and conclude that, although limited, Wyl is an effective learner. Finally we give reasons that explain Wyl's effectiveness.

4.1.1 The Effectiveness of Wyl

There are two view points from which the effectiveness of a learning system can be judged: from that of a teacher of a learning program or that of a learning program designer. The teacher can judge *performance*—how good the system is at learning. One such measure is the number of training instances needed to learn a concept. A system that needs only one training instance is a much better learner than one that needs ten. Another measure is correctness, does the system discover the correct definitions of the concept? A further measure is the breadth, how many different concepts can the system learn and in how many domains?

The system designer, on the other hand, takes an *engineering* approach and looks inside the system to evaluate the representations and methods underlying the learning behavior. Some issues are: How general are the methods and representations employed? How much domain and concept engineering is required? How much time and space does the program need?

Performance level evaluation

At the performance level, Wyl is an effective learner: it learns from few training instances, it discovers the correct concept definitions, and it learns concepts in two different domains. For the concepts we have investigated two well chosen examples suffice to correctly learn a concept. By the measure of bias described by Utgoff (1986), Wyl has near perfect bias. Wyl learned concepts in two similar domains. In each domain Wyl has learned two concepts, and demonstrated incremental learning (*trap-in-2-ply* from *trap*). Compared to human learners, Wyl's performance is dismal. But compared to other learning systems, Wyl incorporates some relatively new characteristics; learning multiple concepts in different domains and incremental learning.

Engineering Level evaluation

The generality of a technique can be determined by considering whether it is suitable for different domains. The overall technique employed in Wyl of using different representations (such as structural or functional) and performing different tasks (such as induction or recognition) in the most appropriate one, appears to be very general. Indead, the success of many explanation based learning systems can be attributed to employing this technique. For example, Lex2 (Mitchell et. al. 1982) employs two representations to describe the concept *productive-operator.* The functional representaiton is employed to succinctly capture the concept, while a structural representation is employed to allow efficient recognition.

The generality of performing induction in the most appropriate representation has strong support considering the many successful inductive learning systems. Although when the most appropriate representation is different from that of the concept instances, the generality of the method is dependent upon the generality of the translation methods available.

More specificly, the methods and representations employed by Wyl were certainly general enough for chess and checkers. The same program was used for each. To the program, a domain is simply specified in the functional and structural language described in Chapter 2. Both the functional and structural languges, although simple, are powerful enough to capture many game playing domains. It is unclear, however, whether the learning method would be suitable for learning concepts in these new domains. Consider the game go-mocu. If we described a legal move as simply a play in any vacant square, it is difficult to see how Wyl could learn any concepts other than *loss-in-1-ply*, *loss-in-2-ply*, etc.

Perhaps the least general method in Wyl is the one employed to translate the functional definition into structural form. Often the representation of a functional concept in a simple structural representation is too disjunctive (consider *loss-in-1-*

ply in go-moku described as all arangements of four squares in a row). The size of this description makes complete enumeration impractical.

The amount of domain and concept engineering required for Wyl was minimal. Both the functional and structural languages were simple to design and required no special considerations for the learning task. The operators and goals that made up the functional language are represented as simple predicates in logic and describe the minimum required for a problem solver. The structural language entailed no special design, it describes instances as they appear, in terms of playing pieces occupying primitive squares.

Wyl was neither efficient in time or space. Wyl runs on a Vax11/750 and is written in LISP and MRS. A typical recognition task in chess, applying the test interpreter on a functional definition, takes approximantly 150 cpu minutes. Recognition using structural definitions was the only task performed efficiently by Wyl, with times in the order of 1 or 2 cpu seconds to classify a *trap* example. However, to compile the *trap* concept into its structural form took approximately 700 cpu minutes and generated over 2000 MRS facts. Another example is the complete training session for the concept *skewer* (given in Chapter 3) which took a little over 1000 cpu minutes and generated over 3000 facts.

4.1.2 Why Wyl works

By employing two different representations, Wyl satisfies the conflicting constraints on representation imposed by the two tasks, of learning and recognition. In this section we expand on how these constraints are satisfied by Wyl. In particular we emphasize how the functional representation employed made inductive learning methods effective.

In the introductory chapter of this thesis, we developed a set of constraints on the representation of concepts and instances for inductive learning to be successful.

Below we reiterate these constraints and illustrate how Wyl satisfies them.

The first constraint states that the concept description should be able to represent the desired concepts succinctly, that is, with the fewest disjunctions. Wyl is able to capture all the concepts of interest as conjunctions, because Wyl uses the most "natural" representation. End game concepts all describe ways to achieve goals through sequences of operators. Hence, a language of operators and goals is most natural.

The second constraint states that the training instances and the concept description should have the same form. This is important as inductive learning methods employ syntactic matching and similarity comparisons to drive generalization and specialization. This constraint is satisfied by Wyl employing an envisionment stage that translates the initial structural instances into functional instances before generalization.

The third constraint states that the representation should capture the semantic similarities between the examples of the concept syntactically. This constraint is satisfied as a product of employing the most "natural" representation. Examples of an end game concept (such as *trap*) are syntactically diverse when described in structural terms. But when described in terms of operators and goals they are syntactically similar. It becomes evident that this constraint is satisfied for the *trap* concept, when the diverse structural descriptions illustrated in Figures 2.2, 2.3, 2.7 and 2.10 are compared with the functional descriptions illustrated in Figures 2.5, 2.6, 2.8, and 2.9.

The final constraint is related closely with the third—namely that negative and positive instances should be syntactically different. We demonstrated that this constraint is satisfied in Chapter 3 during the teaching sessions of Wyl with chess concepts. In both *skewer*, and *knight-fork* we gave both positive instances and "near miss" negative instaces. To form a negative instance from a positive instance, a

single piece is moved one square. Hence, the syntactic structural descriptions of these different instances are very similar. In contrast, because the single move completely changes the outcome of the position, the functional descriptions differ wildly. The envisionment process maps the diverse structural descriptions into separate clusters in the functional space that correspond to separate functional concepts.

4.2 Future research

This section surveys some directions for future research at two distinct levels: First we detail short-term questions on the current implementation and suggest extensions that would considerably increase Wyl's effectiveness. Second, we apply what we have learned from Wyl to the question of constructing more effective learning machines.

In each of these discussions we cover research on learning functional descriptions and research on the knowledge transformation methods employed. Lastly, we briefly address research on the underlying architectures for AI systems employing such methods.

4.2.1 Extensions to Wyl

On knowledge transformation methods

The method Wyl employes to translate its functional concepts into structural form cannot be applied to concepts such as *lost-in-3-ply* or *skewer* because the current implementation of the generator is too slow to generate the immense number of positions. By using special purpose generators as Quinlan (1982) did, it will be possible to evaluate better the compaction algorithms introduced in the thesis.

There are open questions about the compaction method. First, how general

is it? Can it be applied to other domains? Second, how do we identify the best attribute order in which to search for new terms? And finally how can the general method of generation and compaction be changed to operate incrementally?

Of more general interest is to develop a method in Wyl that can construct mixed functional and structural descriptions similar to those designed by Quinlan.

On learning functional descriptions

The functional language employed by Wyl is particularly simple. Concepts are restricted to linear sequences of general move descriptions, terminating in a recognized goal. There are many ways in which this language of concept descriptions could be extended that would greatly increase the number of concepts that could be learned. One extension is to include some disjunctions, such as allowing checker concept definitions to specify operators as *take* or *normal* moves. Another instance is in chess, where, instead of generalizing the values of each "slot" of the operators (*tosquare*, *fromsquare* etc.) to variables, the sets of values can be retained in the concept definition. For example, in the teaching of the concept *skewer* in Chapter 3, Wyl could have defined the piece types to be only {rook ∨ queen} rather than $typefc1.

Many concepts in chess and checkers include repeated sequences of moves that could be described in Wyl's functional language if it were extended to include recursion. The chess concept "square of the pawn law" is such a concept currently being studied by Tadepalli (1985). Wyl could learn such concepts by incorporating recursion in the concept language and employing some of the successful learning methods that can identify repeated sequences in training examples (Dietterich, 1984; Andreae, 1984).

The simplicity of the evaluation function, initially a priority ordering on the known goals (*win* is better than *loss*), limits the learnable concepts. The important

feature in any concept is not that it terminates in a known goal, rather that it involves some form of gain (or loss). Consider learning the concept *advantageous-trade*. The concept definition must include some description that compares the values of the terminating and starting states. One way to achieve this would be to include the reasoning that proved the outcome was better (worse) than the starting state in the functional instances and hence, after generalization, in the concept description.

4.2.2 General Issues

On knowledge transformation methods

There are many important issues in the knowledge transformation methods currently available. The method of envisionment is well understood and imposes few constraints on the domain or the representation. In contrast, the compilation methods currently available are extremely limited and impose strong constraints on the representation and domain. The compilation stage is becoming a "bottle-neck" in learning systems.

Currently, the only method available—constraint back-propagation—translates a restricted set of forms of functional concepts (see Porter & Kibler, 1985). Moreover, the method only provides a partial solution to the problem of designing a new structural language. Utgoff (1986) shows that this method can create new structural terms to extend a given structural language. But it does not appear that this method can generate the entire structural language.

Although currently, there is no general method developed to translate functional descriptions to structural descriptions, the theoretical aspects of such a translation are becoming better understood through the work of Bennett & Dietterich (1986). We cast inefficient functional definitions as procedures which have many delayed tests, that is search needs to be performed before the tests can apply. Compilation is

viewed as moving the applicable tests to positions earlier in the search, reexpressing them in a suitable structural language.

On learning functional concepts

When we turn to the issues inherent in applying the techniques employed in Wyl to general learning problems, there are many open research questions. These questions were cast in terms of explanation based methods and similarity based methods and addressed in detail in Mitchell et al 1986.

In this thesis we cast these questions in terms of representation. We extrapolate the lessons from Wyl and speculate on what they may tell us about an effective learning machine.

A fair summary of Wyl is the following:

- Wyl has limited effectiveness as a learner.
- Wyl employees two simple representations of knowledge and primitive methods for translating knowledge.

This suggests a more general summary of a learning machine (LM)[1]:

- LM has great effectiveness as a learner.
- LM employs many complex representations of knowledge and powerful methods for translating knowledge.

The lessons we have learned from Wyl suggest that for LM to be effective this hypothetical machine must:

- Apply inductive methods only in the representation most natural for the concept of interest.

[1]Via the "climbing generalization tree" rule.

- Translate any new knowledge into other representations where it can be applied to some performance task.

- Maintain consistency between the different representations of the same knowledge.

The first two conditions have been discussed above, the final issue has not been discussed and is addressed here. In Wyl, the compilation stage is only suitable for "one-shot" operation. If, after Wyl had translated a concept definition into structural space, it were shown to be incorrect, the compiler would have to be run again from scratch. Clearly a more incremental approach is needed were only those affected parts of all the representations are updated following a change in the concept definition.

This leads to two questions: In which representation should the concept description be modified? And how shall the updated description translate the changes to the other representations effectively and incrementally?

The most suitable representation in which the description should be modified is, if our thesis is correct, the one in which the concept can be captured most naturally. Wyl does not answer the second question, since it has no way of knowing which parts of the structural description are affected by a change in the functional description. A solution would be to maintain *dependencies* between the different descriptions, that could be used to drive the modifications. For example in Wyl, the single functional description of *trap* could be represented as a disjunction of more specialized *traps* (such as king/king, king/man and man/man *traps*) each describing a smaller set of structural instances. If the king/king *trap* were to be corrected only that set of structural descriptions would need to be recomputed.

4.2.3 Directions in the underlying architecture

Wyl is implemented as a set of interpreters working over expressions in a language defined on top of MRS (Russell 1985). The advantages of using logic programming in learning systems are numerous. First, the program is easy to analyze and modify. Second, the logic descriptions support different kinds of reasoning. For example, the functional concept definitions were interpreted as tests to classify instances by use of the MRS interpreter `trueps` and generators by use of the MRS interpreter `residues`. Third and last, logic tends to canonicalize the descriptions making them explicit and easily manipulated syntactically by the machine.

The great disadvantage of using logic is its great inefficiency due to the time spent in unification and building new index structures. What is needed is better compliers that can translate the inference to some faster form of computation invisibly, like today's LISP compilers.

4.3 Contributions of the Thesis

The first contribution of the thesis is that we have clarified the conditions under which induction will be successful.

The second contribution is to incorporate both explanation-based and similarity-based methods into a working system, gaining the advantages of both.

Finally, we have contributed to the development of methods for transforming functional knowledge into a structural form efficient for recognition. The compaction method developed has been shown to create many useful new terms.

Chapter 5

References

Andreae, P. M., "Justified Generalization: Acquiring Procedures From Examples," Ph.D. dissertation, MIT Artificial Intelligence Laboratory, 1985, also Technical Report 834.

Arbab, B. and Michie, D., "Generating Rules From Examples," in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Bennet, J. S. and Dietterich, T. G, "The test hypothesis and the weak methods," *Rep. No. TR 86-30-4,* Oregon State University, Corvallis, OR. 1986

Buchanan, B. G. and Mitchell, T. M., "Model-Directed Learning of Production Rules," in *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Eds.), Academic Press, New York, 1978.

Dietterich, T. G. and Michalski, R. S., "Learning to Predict Sequences," in *Machine Learning: An Artificial Intelligence Approach, Vol II* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1986.

Finger, J. J., and Genesereth, M. R., "Residue—A deductive approach to design," *Rep. No. HPP-83-46,* Stanford Heuristic Programming Project, Computer Science Department. Stanford University. 1983.

Fu, L. and Buchanan, B. G., "Learning Intermediate concepts in constructing a hierarchical knowledge base," in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Gold, E. "Language identification in the limit." in *Information and Control,* Vol 16, pp447-474, 1967.

Horning, J. J. "A Study of grammatical inference." *Rep. No. CS-139,* Computer Science Department, Stanford University. 1969.

Kedar-Cabelli, S.T., "Purpose-Directed Analogy," in *Proceedings of the Cognitive Science Society,* Irvine, Calif., 1985.

Keller, R. M., "Learning by Re-expressing Concepts for Efficient Recognition," in *Proceedings of AAAI-83*, Washington, D.C., 1983.

Langley, P. W., Zytkow, J., Simon, H. A., and Bradshaw, G. L., "The search for regularity: Four Aspects of Scientific Discovery," in *Machine Learning: An Artificial Intelligence Approach, Vol II* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1986.

McCarthy, J., "Programs with common sense." in *Proceedings of the Symposium on the Mechanization of Thought Processes,* National Physical Laboratory I:77-84. 1958. (Reprinted in M. L. Minsky (Ed.). *Semantic Information Processing.* Cambridge, Mass.: MIT Press, 403-409. 1968)

Michalski, R. S., "A Theory and Methodology of Inductive Learning," *Artificial Intelligence,* Vol. 20, No. 2 pp. 111-161, February 1983.

Minsky, M. "Society of mind," Technical Report, Massachusetts Institute of Technology, (1985).

Minton, S. "Constraint-Based Generalization: Learning Game-Playing Plans from Single Examples," in *Proceedings of AAAI-84*, 1984.

Mostow, D. J. "Machine Transformation of Advice into a Heuristic Search Procedure," in *Machine Learning: An Artificial Intelligence Approach, Vol I* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1982.

Mitchell, T.M., Utgoff, P. E. and Banerji, R., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics", in *Machine Learning: An Artificial Intelligence Approach, Vol I* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1982.

Mitchell, T.M., Mahadevan, S., and Steinberg, L. I., "LEAP: A Learning Apprentice for VLSI Design," in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Mitchell, T.M., Keller, R. M., and Kedar-Cabelli, S.T. "Explanation-Based Generalization: A Unifying view," in *Machine Learning 1, 1*, 1986.

Porter, B. W., and Kibler, D. F., "A Comparison of Analytic and Experimental Goal Regression for Machine Learning," in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Quinlan, J. R., "Learning Efficient Classification Procedures and their Application to Chess End Games" in *Machine Learning: An Artificial Intelligence Approach, Vol I* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tioga Press, Palo Alto, 1982.

Quinlan, J. R., "The Effects of Noise on Concept Learning," in *Machine Learning: An Artificial Intelligence Approach, Vol II* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tiogga Press, Palo Alto, 1986.

Reiter, R., "A logic for Default Reasoning," *Artificial Intelligence,* Vol. 13, No. 1,2, 1980.

Russell, S., "The Compleat Guide to MRS," Knowledge Systems Laboratory, Report No. KSL-85-12, Stanford University, CA, 1985.

Shavlik, J. W., "Learning About Momentum Conservation", in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Simith, d. E., and Genesereth, M. R., "Ordering Conjunctive Queries," *Report No. HPP-82-9* Stanford Heuristic Programming Project, Computer Science Department. Stanford University. 1983.

Tadepalli, p. v., "Learning in Intractable Domains," *Proceedings of the Third International Machine Learning Workshop*, Sky top, Pennsylvania, 1985.

Thagard, P., and Holyoak, K., "Discovering the wave theory of sound: inductive inference in the context of problem solving," in *Proceedings of IJCAI-85*, Los Angeles, CA 1985.

Utgoff, P. E., "Shift of Bias for Inductive Concept Learning," in *Machine Learning: An Artificial Intelligence Approach, Vol II* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), Tiogga Press, Palo Alto, 1986.

Vere, S. A., "Induction of Concepts in Predicate Calculus," in *Proceedings of IJCAI-1975,* Tbilisi, USSR, pp. 281-287, 1975.

Waldinger, R. J., "Achieving several goals simultaneously," in *Machine Intelligence 8,* New York, 1977.

Winston, P., Binford, T., Katz, B. and Lowry, M. "Learning Physical Descriptions from Functional Definitions, Examples and Precedents," *Proceedings of AAAI-83*, Washington, D.C., 1983.

Winston, P. H., "Learning Structural Descriptions from Examples," in *The Psychology of Computer Vision*, Winston, P. H. (Ed.), McGraw Hill, New York, ch. 5, 1975.

Wolff, J. G., "Language acquisition, data compression, and generalization," in *Language and Communication,* Vol 2, pp57-89. 1982.

Appendices

Appendix A

Chess Domain Specification

In this appendix we give a summary of the domain specification of chess. Domian specifications consist of the following:

- A search schema.
- Descriptions of the actions and goals of the domain.
- Structural language in which to describe board positions.

In Figure A.1 we illustrate the numbering schema used in chess.

A.1 Search schema

The search schema of chess is given below:

```
(o-or (single-terminate terminate-state)
      (o-and (single-terminate test-check?)
             (multiple-recursive normal-moves))).
```

The first primitive checks for the termination. The mrsform of the terminate-state frame is given below:

a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

Figure A.1: Chess board notation

```
(terminate-state mrsform (terminate-state $name $result $state $side)).
```

The mrsform is the same for checkers and chess, the rules that recognize goals all have the same form of consequent.

The second form in the search schema illustrates the o-and connective. This is used to make explicit the test for check disscussed in Chapter 3. The first primitive of the o-and has the following code mrsform defined:

```
(test-check? mrsform (move $name $state $newstate $from $to $kind
$type $direct $side $typetaken)).
```

where the move predicate defines the legal moves in chess described later. In addition to the mrsform slot, this test-check? has another slot giving "initial constraints"

```
(test-check? initial-constraints ( ($side . $lastside)
                                   ($kind . take)
                                   ($typetaken . king)))
```

These initial constraints define this move as a move by the opponent that takes the king. The interpreters lookup this initial-constraints information and plug in the bindings before any others. The variable $lastside is maintained by the interpreters to be bound to the non-current player. As the primitive is single-terminate the interpreters check whether the condition defined by the mrsform exists in the current state.

The code mrsform for the final primitive is given below:

```
(normalmoves mrsform (move $name $state $newstate $from $to $kind
$type $direct $side $typetaken)
```

The form unifies with the consequent of the main MRS rule defining legal chess moves described next.

A.2 Actions and Goals

In chess the concept *lose-queen* was given to Wyl as part of the domain specification and used in both the concepts learned. The definition of *lose-queen* in MRS is given below:

```
(if (and (side $player $side)
         (type $player queen)
         (unprovable (occupied $state $square $player))
         (newname $player $side $name))
   (terminate-state $name lose-queen $state $side))
```

The main action rule is made up of two subrules like the actions rule for checkers, a rule that generates legal moves and a rule that makes the moves. First the top level rule:

```
(if (and (find-move $state $kind $from $direct $to $player $side $type $typetaken)
         (make-move $state $from $to $player $newstate)
         ($side opside $opside)
         (newname $from $to $name)
         (unprovable (find-move $newstate take $anyfrom $anydir $anyto $anyp $opside $anyt
   (move $name $state $newstate $from $to $kind $type $direct $side $typetaken))
```

The final form in the clause determines if the resulting state of the move ($newstate) is a check position for side $side.

The rule that makes the move is very similar to the rule used in checkers:

```
(if (and (new-state $newstate)
```

```
      (add-info ((occupied $newstate $to $player)
                 (occupied $newstate $from empty)
                 (nextstate $state $newstate))))
  (make-move $state $from $to $player $newstate))
```

The rule that finds the moves is given below:

```
(if (and (side $player $side)
         (type $player $type)
         (occupied $state $from $player)
         (legal-direction $side $type $direct $count)
         (move-dir $state $kind $direct $count $from $from $to $side $typetaken))
  (find-move $state $kind $from $direct $to $player $side $type $typetaken))
```

The side, type and legal-direction are similar to those used in checkers. The legal-direction facts differ because, in addition to the legal direction of movement, the legal number squares movable in the direction is also specified. The move-dir form computes all the legal moves in a given direction. There are two kinds of moves, those that end in taking a piece and those that end on an empty square. The rules defining this move-dir form are given below:

First the recursive form of move-dir:

```
(if (and (not-zero $count)
         (connected $tempfrom $next $direct)
         (occupied $state $next empty)
         (sub-1 $count $newcount)
         (move-dir $state $kind $direct$newcount $from $next $to $side $typetaken))
  (move-dir $state $kind $direct $count $from $tempfrom $to $side $typetaken))
```

The connected facts are similar to checkers. The next rule is a termination condition for move-dir that defines non-take moves:

```
(if (and (not-zero $count)
         (connected $tempfrom $to $direct)
         (occupied $state $to empty))
   (move-dir $state nontake $direct $count $from $tempfrom $to $side nil))
```

The final rule of move-dir defines take moves:

```
(if (and (not-zero $count)
         (connected $tempfrom $to $direct)
         ($side opside $opside)
         (side $player $opside)
         (type $player $typetaken)
         (occupied $state $to $player))
   (move-dir $state take $direct $count $from $tempfrom $to $side $typetaken))
```

A.3 Structural Language

The board positions are described in a very simple langauge very like that used for checkers. The chess playing squares on the board are described by 420 facts enumerating all the connected relationships between each square and its neighbors:

```
(connected a1 a2 n)
(connected a1 b2 ne)
(connected a1 b1 e) ....
```

Playing peices are named and described by type and side facts like checkers:

```
(type rb1 bishop)
(side rb1 red)
(type wk1 king)
```

(side wk1 white)

The facts used in the legal moves definitions that define the legal directions of types of peices is illustrated below:

(legal-direction white king ne 1)
(legal-direction white king n 1)

(legal-direction red bishop se 8)
(legal-direction red bishop ne 8)

(legal-direction red rook n 8)
(legal-direction red rook s 8)

Appendix B

Checkers concept trap structural concept langauge

In this section we give the complete structural concept language for *trap* created by the compaction stage described in Section 2.3.4. First we give the relational terms created by the path finding algorithm then the descriptive terms created by Tax.

B.1 Relational terms

The relational terms are all defined from the instance language primitive connected.

```
(if (and (connected $lsquare1 $lsquare3 sw)
         (connected $lsquare3 $lsquare2 se))
    (connected $lsquare1 $lsquare2 south-2-square))

(if (and (connected $lsquare1 $lsquare3 se)
         (connected $lsquare3 $lsquare2 sw))
    (connected $lsquare1 $lsquare2 south-2-square))

(if (and (connected $lsquare1 $lsquare3 se)
```

```
        (connected $lsquare3 $lsquare2 se))
    (connected $lsquare1 $lsquare2 se-2-square))

(if (and (connected $lsquare1 $lsquare3 ne)
         (connected $lsquare3 $lsquare2 se))
    (connected $lsquare1 $lsquare2 east-2-square))

(if (and (connected $lsquare1 $lsquare3 se)
         (connected $lsquare3 $lsquare2 ne))
    (connected $lsquare1 $lsquare2 east-2-square))

(if (and (connected $lsquare1 $lsquare3 ne)
         (connected $lsquare3 $lsquare2 ne))
    (connected $lsquare1 $lsquare2 ne-2-square))

(if (and (connected $lsquare1 $lsquare3 ne)
         (connected $lsquare3 $lsquare2 nw))
    (connected $lsquare1 $lsquare2 north-2-square))

(if (and (connected $lsquare1 $lsquare3 nw)
         (connected $lsquare3 $lsquare2 ne))
    (connected $lsquare1 $lsquare2 north-2-square))

(if (and (connected $lsquare1 $lsquare3 nw)
         (connected $lsquare3 $lsquare2 nw))
    (connected $lsquare1 $lsquare2 nw-2-square))
```

```
(if (and (connected $lsquare1 $lsquare3 nw)
         (connected $lsquare3 $lsquare2 sw))
    (connected $lsquare1 $lsquare2 west-2-square))

(if (and (connected $lsquare1 $lsquare3 sw)
         (connected $lsquare3 $lsquare2 nw))
    (connected $lsquare1 $lsquare2 west-2-square))

(if (and (connected $lsquare1 $lsquare3 sw)
         (connected $lsquare3 $lsquare2 sw))
    (connected $lsquare1 $lsquare2 sw-2-square))
```

B.2 Descriptive terms

The descriptive terms are all created by Tax. There are new terms created for each of the instance language descriptive terms.

Square terms

```
(if (or (sq $square s23)
        (sq $square s7)
        (sq $square s11)
        (sq $square s22)
        (sq $square s19)
        (sq $square s14)
        (sq $square s10)
        (sq $square s15)
```

```
          (sq $square s18)
          (sq $square s6))
      (sq $square center))

(if (or (sq $square west-center-side)
          (sq $square s29))
      (sq $square west-single-side))

(if (or (sq $square east-center-side)
          (sq $square s4))
      (sq $square east-single-side))

(if (or (sq $square s21)
          (sq $square s13))
      (sq $square west-single-side))

(if (or (sq $square s12)
          (sq $square s20))
      (sq $square east-center-side))

(if (or (sq $square s2)
          (sq $square s3))
      (sq $square north-center-side))

(if (or (sq $square s30)
          (sq $square s31))
      (sq $square south-center-side))
```

Connected terms

```
(if (or (connected $square-1 $square-2 south-2-square)
        (connected $square-1 $square-2 sw-2-square))
    (connected $square-1 $square-2 south-west-triangle))

(if (or (connected $square-1 $square-2 south-2-square)
        (connected $square-1 $square-2 se-2-square))
    (connected $square-1 $square-2 south-east-triangle))

(if (or (connected $square-1 $square-2 ne-2-square)
        (connected $square-1 $square-2 north-2-square))
    (connected $square-1 $square-2 north-east-triangle))
```

Type terms

```
(if (or (type $player king)
        (type $player man))
    (type $player anytype))
```

Appendix C

Trap structural concept description

In this section we give the complete structural concept description for the concept *trap* generated by the compaction stage described in Section 2.3.4. The descriptions capture *traps* for both sides of the board by use of the mirror-square relation. If the side is white the squares tested are mapped to their symmetrical equivalent. Some examples of the mirror-square relation are given below:

```
(mirror-square s1 s32 white)
(mirror-square s2 s31 white)


(mirror-square s1 s1 red)
(mirror-square s2 s2 red).
```

The 12 members of the disjunction are given below:

```
(if (and ($side1 opside $side2)
         (side $var-piece-1 $side1)
         (type $var-piece-1 anytype)
         (occupied $state $var-square-1 $var-piece-1)
         (mirror-square $var-square-1 $orientsq1 $side1)
         (sq $var-square-1 s4)
```

```
        (side $var-piece-2 $side2)
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 south-west-triangle))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 man)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 center)
        (side $var-piece-2 $side2)
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 south-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 king)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 s29)
        (side $var-piece-2 $side2)
```

```
        (type $var-piece-2 king)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 noth-east-triangle))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 man)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 west-single-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
        (not-equal $var-square-1 $var-square-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 south-east-triangle))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 king)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 s29)
        (side $var-piece-2 $side2)
```

```
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 east-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 anytype)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 east-single-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 king)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 west-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 man)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 s5)
        (side $var-piece-2 $side2)
        (type $var-piece-2 king)
```

```
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 east-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 anytype)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 west-center-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 king)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 east-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 king)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 south-center-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 king)
        (occupied $state $var-square-2 $var-piece-2)
```

```
        (not-equal $var-square-1 $var-square-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 north-2-square))
    (trap-0 $state relevant-side $side1))

(if (and ($side1 opside $side2)
        (side $var-piece-1 $side1)
        (type $var-piece-1 man)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 east-center-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 south-west-traingle))
    (trap-0 $state relevant-side $side1))

(if (and (side $var-piece-1 $side1)
        ($side1 opside $side2)
        (type $var-piece-1 anytype)
        (occupied $state $var-square-1 $var-piece-1)
        (mirror-square $var-square-1 $orientsq1 $side1)
        (sq $orientsq1 north-center-side)
        (side $var-piece-2 $side2)
        (type $var-piece-2 anytype)
        (occupied $state $var-square-2 $var-piece-2)
```

```
        (mirror-square $var-square-2 $orientsq2 $side1)
        (connected $orientsq1 $orientsq2 south-2-square))
    (trap-0 $state relevant-side $side1))

(if (and (trap-0 $state relevant-side $side1)
        (newname trap term $name))
    (terminate-state $name trap $state $side1))
```