AN ANALYTICAL FRAMEWORK FOR LEARNING SYSTEMS*

ROBERT C. HOLTE

FEBRUARY 1988         AI88-72

* This work consists of the author's PhD dissertation from Brunel University, England, Department of Electrical Engineering and Electronics. The author is a Visiting Scholar at the University of Texas at Austin, Department of Computer Sciences.
AN ANALYTICAL FRAMEWORK FOR LEARNING SYSTEMS

by

ROBERT CRAIG HOLTE

A thesis submitted for the degree Doctor of Philosophy

Department of Electrical Engineering and Electronics, Brunel University, England

February, 1988

ABSTRACT

The problem addressed in this thesis is that of defining a set of concepts and techniques
that facilitate the comparison and analysis of learning systems. Systems are modelled in
terms of certain abstract processes and bodies of information. Different types of system
correspond to different ways of representing the model. Systems of different types are
compared using behaviour-preserving transformations. Formal definitions are given for
"representation" and the "generative structure" of a system. These and related concepts,
such as "bias" and "implicit knowledge", facilitate the analysis of a system's efficiency and
its use of task-specific knowledge
Acknowledgements

I gratefully acknowledge the financial support that has made this work possible. Scholarships were awarded by the Imperial Order of the Daughters of the Empire (Canada), by the Natural Sciences and Engineering Research Council (Canada), and by the Committee of Vice-Chancellors and Principals (U.K.). Research fellowships were provided by Douglas Young (University of Manitoba), Tom Addis (ALVEY grant, Brunel University), and Gerry Musgrave and Michael Fourman (ALVEY grant, Brunel University).

My research has benefited immeasurably from the encouragement and criticism provided over the years by many people. Foremost among these is Alan MacDonald, my supervisor and staunchest supporter. I am deeply indebted to Alan, and also to Robert Zimmer, for the careful thought and countless hours they have invested in criticizing drafts of the thesis. Larry Rendell has been the source of much encouragement and inspiration. Tony Begg, Igor Aleksander, Antonia Jones, Yves Kdratoff, Pat Langley, and David Kirsh have all greatly assisted my research endeavours. Finally, I would like to thank my family and friends for their unfailing support.
# TABLE OF CONTENTS

## CHAPTER 1: OVERVIEW
- Background ......................................................... 1
- The Objective of the Thesis ................................... 3
- Content and Organization of the Thesis ...................... 4
  -- Related Work ................................................. 6

## CHAPTER 2: TASKS, MODELS, AND SYSTEMS
- Nonincremental Concept Learning ................................ 11
  -- An Abstract Model of Nonincremental Concept Learning 13
- Hypothesis Selection ............................................. 15
- Sequential Learning Tasks ...................................... 17

## CHAPTER 3: REPRESENTATION
- Preliminary Definitions .......................................... 23
  -- Domain ......................................................... 23
  -- Domain Relation ............................................. 24
  -- The Structure of a System ................................ 25
- Full Representation .............................................. 28
  -- Alternative Representations ................................ 30
  -- Corepresentation ............................................. 33
  -- Representing Systems With Systems ....................... 37
- Distributed Representation .................................... 40
- The Declarative Representation of Functions ................ 42
- Adequacy and Efficiency ........................................ 43
  -- Efficiency ...................................................... 44

## CHAPTER 4: THE EXPLOITATION OF TASK-SPECIFIC KNOWLEDGE
- Example: Overcoming Difficulties Applying ID3 to Tasks Involving Sequences .... 46
  -- Difficulty #1: Limitations of Decision Trees ............... 48
  -- Difficulty #2: Limitations of Attribute-Value Vectors .... 50
- Providing Task-Specific Knowledge Explicitly ................. 51
- Providing Task-Specific Knowledge Implicitly ................ 52
  1. Representing Semantic Entities ............................. 54
  2. Representing Semantic Candidates .......................... 55
  3. Exploiting Properties of a Specific Learning System .... 56
CHAPTER 5: REPRESENTING THE MODEL OF THE LEARNING PROCESS
1. Enumerative Systems ................................................................. 61
2. Candidate Set Modification Systems ......................................... 63
3. Hypothesis Modification Systems ............................................. 65
4. Performance History Modification Systems .............................. 67
5. Mediating Systems .................................................................. 68
6. Mediating Systems (continued) .................................................. 70

CHAPTER 6: EXAMINING THE RELATIONSHIPS BETWEEN SYSTEMS
1. Relationships Based on Transitional Systems ............................ 73
   1. Transitional Merging/Performance History Update Systems ...... 74
   2. Transitional Candidate Set/Hypothesis Modification Systems ... 75
   3. Transitional Candidate Set Modification/Mediating Systems ...... 76
   4. Transitional Mediating/Merging/Constructive Systems ............ 77
2. Relations Based on Behaviour-Preserving Transformations .......... 78
   1. Transforming an Analogy System: A Simple Example ............. 79
      -- Type 1 Decision Trees .................................................... 80
      -- Type 2 Decision Trees .................................................. 81
      -- Type 3 Decision Trees .................................................. 82
   2. Transforming An Analogy System: A Full-Scale Example ......... 83
      -- Specification of the System .......................................... 84
      -- The Analogy System .................................................... 85
      -- The First Transformation: The Table ............................. 86
      -- Second Transformation: The Inverted Table .................... 87
      -- An Alternative Definition of $T_1(Q,V)$ ........................ 88
      -- Third Transformation: The Hypertable ......................... 89
      -- Fourth Transformation: The (Compressed) Inverted Hypertable 89
      -- A NDFSM Alone is Inadequate for the Specified Learning Task90
   3. The Relation Between Analogy and Hypothesis Modification Systems .... 91
3. Important Relationships Not Covered in this Chapter ................ 92
INDEX OF FIGURES

Figure 2.1 ................................................................. 11
Figure 2.2 ........................................................................ 18
Figure 2.3 ........................................................................ 19
Figure 2.4 ........................................................................ 21
Figure 3.1 ......................................................................... 35
Figure 3.2 ......................................................................... 37
Figure 3.3 ......................................................................... 38
Figure 3.4 ......................................................................... 39
Figure 4.1 ......................................................................... 58
Figure 4.2 ......................................................................... 60
Figure 5.1 ......................................................................... 61
Figure 6.1 ......................................................................... 73
Figure 6.2 ......................................................................... 77
Figure 7.1 ......................................................................... 109
Figure 7.2 ......................................................................... 135
Figure 7.3 ......................................................................... 141
Figure 8.1 ......................................................................... 158
CHAPTER 1

OVERVIEW

1.1 Background

Recent years have witnessed the development of a number of task-oriented inductive learning systems that have demonstrated an impressive performance in their specific domain of application. Major weaknesses, however, persist in much of the research in this area. Most systems lack generality and extensibility. The theoretical principles upon which they are built are rarely well explained. Lack of common terminology and an adequate formal theory makes it difficult to compare different learning methods.¹

Machine learning is the study of learning tasks and systems that perform those tasks. Broadly speaking, there are three different types of investigation in machine learning, empirical, analytical, and descriptive. Empirical investigations, in which one observes the behaviour of a specific system on a specific task, are by far the most common. Over the years, many different tasks have been defined, a plethora of systems has been constructed, and the behaviours of the systems on the tasks have been measured and occasionally compared. The results produced by empirical investigations are highly specific.

Analytical investigations may be formal or informal. Most formal investigations define a specific learning task and analyze whether or not an algorithm exists that can perform that task. Other formal investigations analyze a specific learning algorithm to determine its computational complexity or preconditions for its correctness. Until very recently formal investigations have proceeded without reference to the tasks, systems, or issues studied empirically. By contrast, informal analytical investigations attempt to give general explanations of observed phenomena and extract general principles and techniques from the body of empirical results. These investigations have been largely unsuccessful. The proposed principles, such as "model-driven systems have greater noise immunity than

¹ p. 86, [Michalski 1983]
data-driven systems", or "the difficulty of a learning task increases with the number of possible hypotheses", are either imprecisely formulated or demonstrably false in some circumstances. Despite these failings the informal analytical investigations serve two useful purposes. They identify issues of practical importance, such as efficiency and domain-independence, and they express the insights produced by the empirical investigations, such as the importance of "representation" and "bias" to the success of a system.

A descriptive investigation produces a general framework, either formal or informal, with which to describe learning systems. The first purpose of a general framework is to "unify the field" by providing a standard terminology at a suitably abstract level. The need to unify the field arises because systems are often described in implementation-specific terms rather than abstract ones. Important relationships between systems can be entirely obscured by differences in the details of their implementation. In a general framework learning systems are described in abstract terms that suppress much, if not all, of the implementation detail. The first purpose is thus purely descriptive: to describe all systems in a uniform way. A framework that serves this purpose is called a descriptive framework for learning systems². If a descriptive framework happens to highlight features of a system that are important for some analysis tasks, it serves an analytical purpose as well as a descriptive one, and is called an analytical framework for learning systems. For example, the framework in [Mitchell 1982] is clearly intended to be an analytical framework:

The purpose of this paper is to compare various approaches to generalization in terms of a single framework. Toward this end, generalization is cast as a search problem, and alternative methods for generalization are characterized in terms of the search strategies that they employ. This characterization uncovers similarities among approaches and leads to a comparison of relative capabilities and computational complexities of alternative approaches. This characterization allows a precise comparison of systems that utilize different representations for learned generalizations.³

The analytical purpose, in this case, is to facilitate the comparison of relative capabilities and computational complexities.

² definitions are indicated by printing the defined term in bold font. An index of definitions is given at the end of the thesis.
³ (pp. 203-204) in [Mitchell 1982]. In this context, generalization is a learning task.
The frameworks proposed to date have served neither purpose to a significant degree. Their shortcomings for the analytical purpose are due primarily to lack of development. On the other hand, their shortcomings for the descriptive purpose are the result of a fundamental obstacle: the great diversity of learning systems. For each framework there are systems to which the framework applies naturally. For example, a framework that describes systems in terms of particular abstract processes applies naturally to systems having distinct subsystems corresponding to each process. The diversity of learning systems is such that there are always many systems to which a given framework does not naturally apply. No descriptive investigation has addressed the problem of applying a framework to systems to which it does not naturally apply.

1.2 The Objective of the Thesis

The problem addressed in this thesis is that of defining an analytical framework for learning systems. The framework should be applicable to a diverse set of existing systems and it should facilitate analytical activities of importance to the machine learning community, such as,

(1) analysis of the task-specific knowledge used by a system;

(2) analysis of the efficiency of specific systems and whole families of systems;

(3) transferring techniques and analyses from one system or family of systems to another;

(4) comparative analysis: analyzing or explaining the differences in behaviour of two (families of) systems in terms of differences in the (families of) systems.

An analytical framework for learning systems is a set of terms with which to describe learning systems, and techniques for manipulating those descriptions. The terms denote abstract properties and components of learning systems. This is to be contrasted with a theory of learning systems which is a set of general principles and results of analytical activities.
The development of a complete analytical framework, like the development of a complete theory, is a distant goal. If the existing frameworks and informal analyses are regarded as the first step towards this goal, the objective of the present work is to take a second step. More precisely, the objective is to develop the existing ideas into an analytical framework that, although incomplete in other ways, applies to a very diverse set of systems.

1.3 Content and Organization of the Thesis

The thesis brings together into a single framework useful ideas scattered throughout the machine learning literature. In almost every case the formulation of these ideas provided in the thesis is much more precise than the original. New ideas, i.e. terms and techniques, are developed and existing ones revised so that the presented collection of terms and techniques facilitates the analytical activities listed above.

The analytical framework applies to all systems that perform learning tasks of a certain type. This set of systems is very diverse. It includes all systems for the most widely studied learning task, called concept learning, in which a system is given a sample of classified objects and required to produce a rule that can be used to classify any object.

Chapter 2 defines the type of task to which the framework is applicable and introduces an abstract model of systems that perform these tasks. The components of the model are abstract processes and bodies of information. The primary purpose of the model is to identify the processes and bodies of information in a learning system that are important in carrying out the analytical activities of interest. Describing systems in terms of this model thus assists the analytical activities. As with all previous frameworks, there arises the problem of applying this abstract model to systems in which components of the model cannot easily be identified. This problem is essentially one of understanding the ways in which the components may be represented in a system. To resolve the problem (Chapter 5) it is necessary to investigate the phenomenon of representation.
Chapter 3 formally defines the notion of representation (and interpretation) as a relation between two domains. From this starting point there follow definitions of several intuitive notions, such as adequacy and generality, and restatements of empirical observations, such as the influence of representation on efficiency. Using these concepts, Chapter 4 addresses the first analytical activity, the use of task-specific knowledge by a system. Task-specific knowledge can influence both the adequacy and the efficiency of a system. Particular attention is paid to what is commonly called "implicit" knowledge.

Chapter 5 discusses the different ways that the components of the abstract model may be represented in systems. In doing so, it establishes that the abstract model can be used to describe a diverse set of learning systems. Each different way of representing the components defines an architecture for learning systems, and systems can be classified according to their architectures. The resulting classification scheme closely resembles some existing schemes.

Chapter 6 addresses the analytical activities that involve relating one system or family of systems to another. As a general technique it proposes transforming one system into another in a way that preserves those aspects of the system of analytical interest. It demonstrates that the different classes of system defined in Chapter 5 are related by transformations that preserve behaviour (but not necessarily efficiency). Consequently, it is shown that analyses of behaviour and principles governing behaviour cannot differ systematically between these classes.

Chapter 7 addresses the analysis of the efficiency of a system. It gives a formal definition of the "generative structure" of a specific type of system and demonstrates that this property is important in the analysis of the efficiency of systems of this type. Finally, generative structure is shown to be a generalization of the notions of structure presently used to analyze other types of system.
Related Work

The first section of the present chapter is a general summary of work related to this thesis. Descriptive investigations are the most closely related type of work. Formal frameworks are given in [Haralick 1978, Haralick and Kartus 1978] and [Holland 1975]. Informal frameworks are given in [Barto and Sutton 1981, Buchanan et al. 1977, Michalski 1983] and [Mitchell 1982]. All of these frameworks are extremely underdeveloped. Informal analyses of systems include [Bundy et al. 1985, Lenat and Brown 1984, Dietterich and Michalski 1981] and [Langley 1983]. Conjectures of general principles occur throughout the literature. The most important analytical concept to emerge from these investigations is that of the "structure" of a set of hypotheses. The informal framework in [Rendell 1986, Rendell 1987] includes the most extensive treatment of this concept. Chapter 7 compares different notions of the structure of a set of hypotheses.

In contrast to frameworks for learning systems, there are two frameworks for the formal analysis of learning tasks that are very well-developed: the computational framework and the statistical framework. In both frameworks a task is specified by

-- a set of target functions.

    computational framework: targets are, e.g., characteristic functions of formal languages

    statistical framework: targets are, e.g., members of a family of probability density functions characterized by a single real-valued parameter, θ

-- for each target, the set of admissible presentations of the target. When a system is applied to the task it receives as input one of the admissible presentations.

-- a set of hypotheses, i.e. functions (or descriptions of functions). When a system is applied to the task it must produce one of the hypotheses (or one sequence of hypotheses) as output. The set of hypotheses is often a superset of the set of targets.
-- a criterion of success, i.e. a criterion by which to evaluate the success of a system
applied to the task.

computational framework: the two most common criteria of success are
(i) **identification in the limit**: for every admissible presentation of
every target, all but a finite number of the hypotheses in the
sequence produced by the system are equivalent to the target
[Angluin and Smith 1983, Gold 1967].
(ii) **approximation in polynomial time**: within polynomial time, the
system must produce, with high probability, a hypothesis that
approximates the target with high accuracy
[Haussler 1987, Valiant 1984].

statistical framework: e.g., "naive consistency" -- for every admissible presentation
of every target, \( \theta_n \to \theta \) as \( n \to \infty \), where \( \theta_n \) is the hypothesis produced
by the system after receiving the first \( n \) members of the presentation
(pp.280-282, [Rao 1965]).

Analyses carried out within both frameworks address the same general question: "construct
an algorithm that satisfies the criterion of success of a given task, or show that no such
algorithm exists". Beyond these similarities at a very abstract level, the computational and
statistical frameworks differ sharply in their conceptual and mathematical details. No
framework for tasks that unifies the computational and statistical frameworks has yet been
developed.

Naturally, neither of the frameworks for task analysis is intended to facilitate the analysis of
systems. For example, the efficiency of the algorithms constructed in a task analysis is of
no concern (unless it is explicitly mentioned in a criterion of success). Thus the algorithms
constructed in the course of analyzing an "identification in the limit" task are typically easy
to analyze but extremely inefficient. As a second example, the relationships between
systems that succeed at a task is not an issue addressed within a framework for task
analysis. Unlike the existing task-analysis frameworks, the framework developed in the
thesis is intended to facilitate the analysis of these (and other) "system" issues.

Some analyses, for example the analysis of the use of task-specific knowledge by a system,
require the relationship between the task and the system to be examined. To facilitate these
analyses, a framework for systems should be compatible with a framework for tasks, in the
sense that
-- the tasks considered within the framework for systems should be analyzable within
   the framework for tasks, and
-- the systems constructed within the framework for tasks should be analyzable within
   the framework for systems.

The framework developed in the thesis is compatible, in this sense, with the computational
framework for learning tasks. It is left as an open question whether or not the framework
developed in the thesis facilitates the analysis of systems applied to the tasks that arise
within the statistical framework.
CHAPTER 2

TASKS, MODELS, AND SYSTEMS

As defined in Chapter 1, a task is a set of targets, a set of hypotheses or outputs, a criterion of success, and, for each target, a set of admissible presentations or inputs. In a classification task there is one target, a relation (CORRECT) specifying the correct output(s) for each input, inputs are called examples, outputs, called classifications, are sets of classes, and the criterion of success is

\[ \text{FOR ALL } i \in \text{INPUTS}: (\text{CORRECT } <i, (F I)>). \]

where F is the system whose success at the task is being evaluated. For instance, if the examples are positive integers and the classes are prime, odd, and even, the correct classification(s) of 3 could be (prime, odd) in one task and (prime), (odd), and (prime, odd) in another.

A system is applied to a task if an input associated with the task is used as the input to the system. A system, S, that satisfies the criterion of success of task T is said to perform T, denoted (PERFORMS S T). A system that performs task T is called a T-system, e.g., a system that performs a classification task is called a classification system.

Typically, the inputs and outputs associated with a system are not the same as those associated with a task. The adjectives syntactic and semantic will be used to distinguish system-related and task-related entities. Thus to apply a system to a task it is necessary to represent semantic inputs as syntactic inputs and semantic outputs as syntactic outputs. For example, a system whose input is a list of integers and whose output is one of the characters "P", "O", or "E" may be applied in several ways to the task of classifying a positive integer in which the correct classification of a prime is (prime), of an odd composite is (odd), and of an even composite is (even). Three of the lists that could be used to
represent the semantic input \( N \) are the list \([N]\), the list of primes \([P_1, \ldots, P_n]\) such that \( n \leq P_i \) and \( N = \prod P_i \), and the list of binary digits \([B_1, \ldots, B_j]\) such that \( B_i = 1 \) and \( N = \sum B_i 2^{i-1} \). There are six (3!) ways of representing the semantic classifications (prime), (odd), and (even) by "P", "O", and "E".

Tasks and the "represents" relation between tasks and systems are important parts of an analytical framework for systems. Some analytical activities are defined in terms of a system and a task, for example, the analysis of a system's use of task-specific knowledge. Comparative analyses often are made in the context of a specific task. Indeed, a major failing of the "Langley/Ohlsson law"\(^1\) relating the performance of two types of system is that its analysis of one type of system is made (unwittingly) in the context of a different task than its analysis of the other.

Some analytical activities can be carried out on tasks and systems independently. In particular, the computational complexity of a task can be analyzed without reference to a specific system\(^2\), and the computational complexity of a system can be analyzed without reference to a specific task\(^3\). However, the independent analysis of a task and a system need not give an accurate estimate of the computational complexity of the system applied to the task (with some specific representation). On one hand, a system can be much less efficient than the task analysis indicates. This happens when a "general-purpose" system fails to exploit properties specific to the task. For example, a system for finding the smallest member in an arbitrary list is inefficient compared to the complexity of the task of finding the smallest member of a list sorted in ascending order. On the other hand, a system can be much more efficient than its complexity analysis indicates. This happens, for example, if all the semantic inputs are represented by syntactic inputs that are "best cases" for the system.

\(^1\) [Ohlsson 1983]. Other shortcomings of this law are given in [Bundy 1984].
\(^2\) e.g. [Angluin and Smith 1983], [Keams et al. 1987], and [Garey and Johnson 1979].
\(^3\) e.g. [Minsky and Papert 1969] and [Angluin 1982]
All the analytical activities are facilitated by including in the analytical framework the semantic context of a learning system, that is, the learning task, the semantic domain associated with the task (see Chapter 3), and the "represents" relation between the task (and domain) and the system (see Figure 2.1). Including the semantic context in the analytical framework also facilitates the descriptive use of the framework. The abstract model of learning systems developed in the following sections is very similar to certain types of task. This has two desirable consequences. First, the abstract model "bridges the gap" between tasks and systems. Second, the relation between the model and a specific system is the same type ("represents") as the relation between a task and a system.

![Figure 2.1](image)

Figure 2.1. Initial version of the semantic context of a learning system. The final version is in Figure 2.4.

2.1 Nonincremental Concept Learning

In a nonincremental concept learning task, targets are classification systems; \( \text{INPUTS}_T \) denotes the set of inputs associated with target \( T \). Given any input associated with any target \( T \), a nonincremental learning system must produce a classification system that produces the same classification as \( T \) for all the examples in the given input. Note that the classification produced by \( T \) is part of the input. More precisely,
INPUT: a set of pairs \( \langle E_i, (T, E_j) \rangle \in \text{INPUTS}_T \)

where

- \( T \) is TARGETS is a classification system,
- \( E_i \) is an element of the set of possible examples.
- This set does not depend on \( T \).

In a given input, \( \langle E_i, (T, E_j) \rangle \), each pair is a \text{classified example} and each \( E_i \) is a \text{seen example}. If \( K \) is a class, the concept corresponding to \( K \) (for target \( T \)) is \( \langle E_j | K \in (T, E_j) \rangle \).

OUTPUT: a classification system.

The set of possible outputs is called CANDIDATE-SET; its elements are called \text{candidates}. The candidate chosen as output is called the \text{hypothesis}.

CRITERION OF SUCCESS:

FOR ALL \( T \in \text{TARGETS} \) and FOR ALL \( l \in \text{INPUTS}_T \):

FOR ALL \( \langle E, X \rangle \in l \): \( \text{HYPOTHESIS} = X \)

where

\( \text{HYPOTHESIS} = \langle F < l, \text{CANDIDATE-SET} > \rangle \)

and \( F \) is the system being applied to the task.

The predicate \( \langle SC < l, C > \rangle \equiv \langle \text{forall} \langle E, X \rangle \in l : \langle C, E \rangle = X \rangle \) is the \text{selection criterion} for a nonincremental concept learning task. A candidate, \( C \), is \text{compatible} with input \( l \) if \( \langle SC < l, C > \rangle \) is true.

This definition includes tasks that are conventionally distinguished from "concept learning".

Some of the distinctions are based on the way hypotheses are represented: according to convention, the learning of decision trees and sets of production rules is concept learning, but the learning of finite state machines and grammars is not. Other distinctions are based on the nature of the hypothesis, e.g. whether the number of possible classifications is finite and bounded ("concept learning") or not ("program synthesis"). Finally, some distinctions are based on the use made of the hypothesis, e.g. whether the classification task to which it is applied is the primary task ("concept learning") or a subtask of the primary task ("strategy learning").

The analytical significance of these distinctions varies considerably. To maximize the range of application of the analytical framework none of the conventional distinctions will be adopted unless necessary. For example, Chapters 5 and 6 discuss at length distinctions

---

4 Dicterich et al. 1982 and Holte 1985 are surveys of the conventional concept learning task. Further discussion of the limits of the conventional task is given in Holte and MacDonald 1987 (pp.127-128).
based on the way hypotheses are represented.

An Abstract Model of Nonincremental Concept Learning

"Learning as search" summarizes, in a phrase, the following model of nonincremental concept learning systems5.

given: CANDIDATE-SET and input I={<E_i,X_i>}

repeat: HYPOTHESIS←(ENUMERATE CANDIDATE-SET)
until: FOR ALL <E_i,X_i>∈ I: (HYPOTHESIS E_i)=X_i

output: HYPOTHESIS

where (ENUMERATE CANDIDATE-SET) produces the elements of CANDIDATE-SET one at a time in such a way that every element eventually occurs6. The following block diagram depicts this model. The dotted line separating ENUMERATOR and SELECTION CRITERION indicates that the two processes are tightly coupled.

\[
\text{CANDIDATE SET} \downarrow
\]

\[
\text{ENUMERATOR}
\]

I

\[
\text{SELECTION CRITERION} \downarrow
\]

HYPOTHESIS

A model such as this can be viewed in two ways, concretely or abstractly. In the concrete view, the model is a system architecture; each process in the model is a distinct subsystem in the architecture. For example, [Smith et al. 1977] defines a model of incremental learning.

---


6 more precisely, if E_i is the i_th element produced by (ENUMERATE CANDIDATE-SET), then for every C∈ CANDIDATE-SET there exists an integer N such that C=E_N. All enumeration sequences in the thesis are assumed to have this property.
and encourages viewing it concretely (p. 339):

The components of the model are conceptual entities which specify the functions that must be performed to effect learning. Although the functional decomposition suggested by the model is not necessarily reflected in the physical decomposition of many existing systems, we do advocate such a correspondence in future learning system designs.

It is evident from this quotation that the model in [Smith et al. 1977], viewed concretely, is inapplicable to many systems. It is not the model that is at fault. Existing learning systems are sufficiently diverse that no concretely-viewed model can be applicable to them all.

For an analytical framework to be broadly applicable, the model(s) in it must be viewed abstractly. Viewed in this way, a model is more a decomposition of the task than the system. The enumeration of CANDIDATE-SET and the evaluation of the selection criterion for each candidate are subtasks of the nonincremental learning task, not subsystems of nonincremental learning systems. To relate the model to an actual system, one must identify in the system the means by which the subtasks in the model are achieved. In most systems there will not be distinct subsystems responsible for each subtask. For example, in the three architectures for nonincremental learning systems described in [Mitchell 1982] activities related to the enumeration of CANDIDATE-SET are interspersed among those related to the evaluation of the selection criterion. As a second example, according to the present model the set of candidates is fixed. Viewed concretely, the model is inapplicable to a system that modifies the set of candidates during the learning process. Viewed abstractly, the model is applicable to such a system: the fixed set of candidates in the model corresponds to the union of the all the sets that can be constructed by the system.

In addition to providing a uniform descriptive framework, the "learning as search" model identifies processes of importance to certain analytical activities. The postconditions of a nonincremental learning system are given by the selection criterion. The quantifier "FOR ALL \( x \in E \)" in the selection criterion is a potential source of inefficiency: the ID3 system was originally designed as a solution to the problem of learning efficiently from a very large set of examples. The enumerator in the model draws attention to the principal

---

7 "He CANDIDATE-SET" is also a postcondition
8 references for learning systems cited by name are given at the end of the thesis

- 14 -
efficiency issue for nonincremental learning system: "how can the number of candidates enumerated be kept within acceptable bounds?". It also highlights the issue "how is it possible to enumerate CANDIDATE-SET?". In some domains the enumeration of the desired set of candidates is a major difficulty. For example, the discovery of an algorithm for enumerating all cyclic molecular structures, without duplication, constituted a major breakthrough in the DENDRAL project [Lindsay et al. 1980].

A model "bridges the gap" between tasks and systems in the sense that the entities -- e.g. CANDIDATE-SET, I, and HYPOTHESIS -- and processes in it may be viewed as either syntactic or semantic. Note, however, that the syntactic view of a model is not equivalent to the semantic view: the relation between these views is examined in the next three chapters. For example, the syntactic candidate set can include entities that do not represent any semantic candidate whereas the semantic candidate set cannot (by definition). So too can there be possible syntactic examples that do not represent possible semantic examples. Another difference is that syntactic candidates need not be classification systems. Very often they are entities, such as decision trees, for which there exists an interpreter that maps them onto classification systems. If a syntactic HYPOTHESIS is not a classification system then the selection criterion, which involves computing (HYPOTHESIS E), can be evaluated only if the learning system has access to an interpreter of syntactic candidates. The model in [Mitchell 1982] is explicitly syntactic and includes as an input to the learning system an interpreter of syntactic candidates.

2.2 Hypothesis Selection

The definition of a hypothesis selection task is a generalization of the definition of a nonincremental concept learning task. In a hypothesis selection task an output may be any type of system, it need not be a classification system. There are no restrictions on the task to which an output is applied, called the performance task. The terms CANDIDATE-SET, candidate, and hypothesis are used as before.

- 15 -
There are also no restrictions on the nature of the input of a hypothesis selection task. An input may be any form of information or "advice" pertinent to the selection of a hypothesis. It may be information about the performance task, about a particular target or the set of targets, or about candidates. For example, in a concept learning task the seen examples provide information about the performance task: specifically, the seen examples are possible inputs for the performance task. In a hypothesis selection task input could include examples that are not possible inputs for the performance task [Rollinger 1987]. As a second example, in a concept learning task, the classification of a seen example indicates the output of a target given a specific input. In a hypothesis selection task, input could also include a trace, or explanation, of a target behaviour\(^9\).

Having no restrictions on input or output, a **hypothesis selection task** is any learning task whose criterion of success is of the form

\[
\text{FOR ALL } T \in \text{TARGETS and FOR ALL } I \in \text{INPUTS}_T: \quad (\text{SC} < I, \text{CANDIDATE-SET}, \text{HYPOTHESIS}>)
\]

where

\[\text{HYPOTHESIS} = (F < I, \text{CANDIDATE-SET}>)\]

\(F\) is the system being applied to the task,

\(\text{SC}\), the selection criterion, is any relation between inputs, the set of candidates, and individual candidates.

Many clustering and (nonincremental) theory formation tasks have a criterion of success of this form and therefore are hypothesis selection tasks. Typically, the selection criterion is true only of those candidates that minimize a given function. This function usually combines a measure of the "complexity" of a candidate and a measure of the "poorness-of-fit" of the candidate and the input\(^{10}\), as in

\[
(\text{SC} < I, \text{CANDIDATE-SET}, H >) \iff \quad \text{FOR ALL } C \in \text{CANDIDATE-SET}: \quad (\text{COMPLEXITY } H) + (\text{POORNESS } < H, I >) \leq (\text{COMPLEXITY } C) + (\text{POORNESS } < C, I >)
\]

An input in a clustering or theory formation task is a set of "observations". The exact nature of the output varies from task to task. For example, in [Michalski and Stepp 1983] an output

\(^9\) [Mitchell et al. 1986], [Porter and Basseiss 1986], [Bundy et al. 1985], [Bauer 1979]

\(^{10}\) p. 345, [Michalski and Stepp 1983]; [Geoff and Wallace 1984]
is a function from observations to names of clusters, and in [Georgeff and Wallace 1984] it is a function from observations to probabilities of occurrence.

The model for hypothesis selection is based on that for nonincremental learning:

\[
\begin{align*}
given: & \text{CANDIDATE-SET and input I} \\
repeat: & \text{HYPOTHESIS} \leftarrow (\text{ENUMERATE CANDIDATE-SET}) \\
until: & (\text{SC} \ < i, \text{CANDIDATE-SET}, \text{HYPOTHESIS}) \\
output: & \text{HYPOTHESIS} \\
\end{align*}
\]

where SC is the selection criterion for the task.

The block diagram depicting this model is the same as above.

2.3 Sequential Learning Tasks

One additional feature common to all [the six chess programs examined] has not been brought out -- namely, that they all choose moves in a chess position rather than play a game of chess. The chess game is put together as a series of essentially independent move choices.\(^{11}\)

The input and output of a sequential task are interleaved sequences, that is, the \(k^{th}\) item in the input sequence, \(P_k\), occurs after the \((k-1)^{th}\) item in the output sequence, \(H_{k-1}\). The possible values of \(P_k\) may or may not depend on the values of \(P_i\) and \(H_i\) (i<k). For example, in a chess-playing task, \(H_k\) could be the \(k^{th}\) move by the chess program and \(P_k\) the state of the chess board after the chess program and its opponent have each made \(k\) moves. As a

\(^{11}\) p. 703, [Newell and Simon 1972]
second example, if $H_{k-1}$ is a system, $P_k$ could include a trace of the execution of $H_{k-1}$ on a given performance-task input. The block diagram in Figure 2.2 depicts the interleaving of input and output sequences.

![Block Diagram](image)

*Figure 2.2*

A **sequential hypothesis selection task** is a sequential task with a criterion of success of the form

$$\text{FOR ALL } T \in \text{TARGETS and FOR ALL } [P_1, P_2, \ldots] \in \text{INPUTS}_T \text{ and FOR ALL } k \geq 1:\n$$

$$(\text{SC } <\text{HISTORY}_k, \text{CANDIDATE-SET}, \text{HYPOTHESIS}_k>)$$

where

- SC is the selection criterion, as before,
- HYPOTHESIS$_k = (F <\text{HISTORY}_k, \text{CANDIDATE-SET}>)$,
- $F$ is the system being applied to the task.

Each $P_k$ is an item of **performance information**.

$\text{HISTORY}_k = (\text{UPDATE } <\text{HISTORY}_{k-1}, P_k, \text{HYPOTHESIS}_{k-1}>)$,

where UPDATE may be any function,
and HISTORY$_0$ and HYPOTHESIS$_0$ may be any values.

HISTORY$_k$ is called a **performance history**.

An example of a sequential hypothesis selection task is **incremental concept learning**, defined as follows. For a given target $T$, each $P_k$ is $<E_k(T, E_k)>$, a classified example.

HISTORY$_0 = \{\}$, and UPDATE simply adds $P_k$ to HISTORY$_{k-1}$. Thus, HISTORY$_k$ is a finite set of classified examples. The selection criterion is
\( (SC < HISTORY_k, CANDIDATE-SET, HYPOTHESIS_k >) \iff \\
\text{FOR ALL } < E, X > \in HISTORY_k; (HYPOTHESIS_k E) = X \)

As before, a candidate, \( C \), is compatible with \( HISTORY_k \) if
\( (SC < HISTORY_k, CANDIDATE-SET, C >) \) is true.

Systems that perform sequential hypothesis selection tasks will be modelled as the composition of two processes, a performance history update process and a (nonsequential) hypothesis selection process (see Figure 2.3). The hypothesis selection process will be modelled in terms of an enumerator and a selection criterion evaluator (not shown in the figure), as before. The hypothesis application process generates successive items of performance information: it is the "input generator" of Figure 2.2. Typically, this process involves applying the hypothesis to performance-task inputs.

\[\text{CANDIDATE SET} \]

\[\text{PERFORMANCE HISTROY}\]

\[\text{Hypothesis Selection} \]

\[\text{Performance History Update} \]

\[\text{HYPOTHESIS} \]

\[\text{Hypothesis Application} \]

\[\text{PERFORMANCE INFORMATION} \]

\[\text{System Boundary} \]

\[\text{Figure 2.3. An abstract model of sequential learning systems.}\]
There are many sequential learning tasks that are not sequential hypothesis selection tasks. For example, sequential learning tasks that have "identification in the limit" (defined in Chapter 1) as a criterion of success are not sequential hypothesis selection tasks. This is because the criterion "all but a finite number of the hypotheses in the output sequence are equivalent to the target" cannot be used directly as a selection criterion; i.e., it cannot be used directly to select a hypothesis, given a performance history and a set of candidates. The qualification "directly" is important, because many tasks defined in terms of identification in the limit can be reduced to sequential hypothesis selection tasks. By the same reasoning, sequential learning tasks that have "approximation in polynomial time" as a criterion of success are not themselves sequential hypothesis selection tasks, although they can often be reduced to sequential hypothesis selection tasks.

The significance of sequential hypothesis selection tasks is this. Let LT1 be any sequential learning task for which there exists a system, S1, that performs LT1. Then there must exist a sequential hypothesis selection task, LT2, such that, for any system S2, (PERFORMS S2 LT2)⇒(PERFORMS S2 LT1). That is, any system that can perform LT2 can also perform LT1. The reason for this is straightforward:

-- If the system S1 performs LT1 then any system that produces the same output as S1, given the same input, will also perform LT1;

-- the task "produce the same output as S1, for the same input" is a sequential hypothesis selection task.

For example, in many cases where LT1 is an "identification in the limit" task or an "approximation in polynomial time" task, there exists an incremental concept learning task, LT2, such that any system that can perform LT2 can also perform LT1. This illustrates that the study of learning tasks and systems can be divided into

1. the study of the relation between arbitrary sequential learning tasks and sequential hypothesis selection tasks; and

2. the study of the relation between sequential hypothesis selection tasks and learning systems.

This division is incorporated into the semantic context of a learning system in Figure 2.4.
The framework in this thesis is intended to facilitate analytical activities involving any sequential learning system and any sequential hypothesis selection task. For this purpose, the abstract model in Figure 2.2 will be applied to all sequential learning systems. The framework is not intended to facilitate analytical activities involving general sequential learning tasks. For example, it is intended to facilitate analyzing the efficiency of hypothesis selection but not the efficiency, e.g. as measured by rate of convergence, of a sequence of hypothesis selections. Finally, for a given sequential learning task, LT, there may be many sequential hypothesis selection tasks, \( HST_1, HST_2, \ldots \) such that \((\text{PERFORMS S } HST_i) \Rightarrow (\text{PERFORMS S } LT)\). Any individual \( HST_i \) can be analyzed within the framework of the thesis. But because the \( HST_i \) can differ considerably, for example in computational complexity, an analysis of any particular \( HST_i \) cannot be regarded as an analysis of LT.
CHAPTER 3

REPRESENTATION

This chapter formalizes the following notions. A domain is a collection of entities and operations on entities. In a chess endgame domain, for example, each board configuration is an entity, and typical operations include moving a piece, determining if a player is in "check", and measuring the distance (e.g. number of moves) between one piece and another. A system is a specific organization of the operations and entities of a domain. The organization of the system is called its structure. The essence of representation is that it preserves behaviour, i.e., the input-output behaviour of a system in one domain is mirrored by the input-output behaviour of the system that represents it in another domain. Representation may or may not preserve structure.

The term "representation" is most often used to described the correspondence between the entities and functions associated with a task and those in a specific learning system. For example, the entities in a medical diagnosis task, such as specific diagnoses, laboratory tests, and test outcomes are represented in the ID3 system by classes, attributes, and values. The activity of selecting a representation of the entities and functions associated with a task is sometimes distinguished from the activity of formulating the task, but this distinction is not made in the discussion that follows. The two activities are treated as two stages of a single activity and "representation" is used to describe the correspondence between entities and functions in the original domain, called the semantic domain, and those in a specific learning system. To distinguish it from the semantic domain, the domain containing the learning system is called the syntactic domain.

The abstract model of the learning process defined in Chapter 2 is an organized collection of entities and functions (processes). The entities in the abstract model have direct counterparts in the semantic domain - semantic candidates, semantic performance
information, and so on. The correspondence between the model and a system in the syntactic domain is often much less straightforward. Chapter 5 examines this correspondence by treating it as a representation of the model in the syntactic domain.

3.1 Preliminary Definitions

Application of a function \( F \) to an argument \( X \) is denoted \( (F \ X) \). If \( F \) is a partial function and \( X \) a set define \( \text{DOM}_X \ F \) to be \( \{x \in X | (F \ x) \text{ is defined}\} \) and \( \text{IMAGE}_X \ F \) to be \( \{ (F \ x) | x \in \text{DOM}_X \ F \} \). In the context of a specific \( X \) \( \text{DOM}_X \) is abbreviated as \( \text{DOM} \) and \( \text{IMAGE}_X \) as \( \text{IMAGE} \).

Domain

A domain is a 4-tuple \(<E,F,S,C>\) of (possibly infinite) sets, defined as follows. The elements of \( E \) are the entities of the domain. A collection of entities, such as a set \( \{e_1,e_2,e_3\} \) or tuple \(<e_1,e_2,e_3>\), is distinct from the entities in it; the collection may or may not be an entity in the domain. The elements of \( F \) are the primitive functions of the domain. Each is a partial function from \( E \) to \( E \). Functions normally regarded as having \( N \) arguments are treated as functions having one argument that is an \( N \)-tuple.

The elements of \( S \) are the structured functions of the domain. Each element of \( S \) has a definition, where a definition is a composition of a finite number of entities, primitive functions, and other definitions. \( C \) is the set of composition functions that may be used for this purpose. Some typical composition functions are

\[
(\text{WHILE } <F_c,F_b>) \quad \text{is a function, } G, \text{ such that} \\
\quad (F_c X) \Rightarrow (G X) = (G (F_b X)) \\
\quad \text{and } \sim (F_c X) \Rightarrow (G X) = X
\]

\[
(\text{SEQUENCE } <G_1,G_2,\ldots,G_n>) \quad \text{is a function, } G, \text{ such that} \\
\quad (G X) = <(G_1 X), (G_2 (G_1 X)), \ldots, (G_n (G_{n-1} (\ldots (G_1 X) \ldots)))>.
\]

\[
(\text{BIND_ARG1 } <H,Y>) \quad \text{is a function } G \text{ such that} \\
\quad (G <Z_1,\ldots,Z_n>) = (H <Y,Z_1,\ldots,Z_n>)
\]
The functions of a domain are the elements of \( F, S, \) and \( C \). The systems are the elements of \( F \) and \( S \).

**Domain Relation**

A domain relation, \( R \), specifies a correspondence between the entities and functions in \( \text{Domain1}=\langle E_1, F_1, S_1, C_1 \rangle \) and those in \( \text{Domain2}=\langle E_2, F_2, S_2, C_2 \rangle \). \( R \) is usually a collection of relations or functions. If \( R=\langle R_1, \ldots, R_n \rangle \) is a collection of functions the notation \( (R_i X)=Y \) will be used; if any of the \( R_i \) are relations the notation \( (R_i <X, Y>) \) will be used for all \( R_i \). For example, \( R \) is a simple isomorphism if it is a triple \( <R_i, Rf, Rs> \), with the following properties.

1. \( Re, Rf, \) and \( Rs \) are functions.
2. \( (\text{DOM } Re)=E_1, (\text{DOM } Rf)=F_1, \) and \( (\text{DOM } Rs)=S_1 \).
3. \( (\text{IMAGE } Re)=E_2, (\text{IMAGE } Rf)=F_2, \) and \( (\text{IMAGE } Rs)=S_2 \).
4. \( (Re \ X)=(Re \ Y) \Rightarrow (X=Y), (Rf \ X)=(Rf \ Y) \Rightarrow (X=Y), \) and \( (Rs \ X)=(Rs \ Y) \Rightarrow (X=Y) \)
5a. for all \( f \in F_1: \)
   - \( X \in (\text{DOM } f) \Leftrightarrow (Re \ X) \in (\text{DOM } (Rf \ f)) \)
   - for all \( X \in (\text{DOM } f): (Re \ (f \ X))=((Rf \ f) \ (Re \ X)) \)
5b. for all \( f \in S_1: \)
   - \( X \in (\text{DOM } f) \Leftrightarrow (Re \ X) \in (\text{DOM } (Rs \ f)) \)
   - for all \( X \in (\text{DOM } f): (Re \ (f \ X))=((Rs \ f) \ (Re \ X)) \)

If \( R=\langle R_1, \ldots, R_n \rangle \) only some of the \( R_i \) will be applicable to a given \( X \in E_1 \cup F_1 \cup S_1 \cup C_1 \). The notation "\( (R \ X) \)" is a shorthand for all the \( (R_i \ X) \) such that \( R_i \) is applicable to \( X \). Using this convention the definition of a simple isomorphism is

1. \( R=\langle Re, Rf, Rs \rangle \) is a function.
2. \( (\text{DOM } Re)=E_1, (\text{DOM } Rf)=F_1, \) and \( (\text{DOM } Rs)=S_1 \).
3. \( (\text{IMAGE } Re)=E_2, (\text{IMAGE } Rf)=F_2, \) and \( (\text{IMAGE } Rs)=S_2 \).
4. \( (R \ X)=(R \ Y) \Rightarrow (X=Y). \)
5. for all \( f \in F_1 \cup S_1: \)
   - \( X \in (\text{DOM } f) \Leftrightarrow (R \ X) \in (\text{DOM } (R \ f)) \)
   - for all \( X \in (\text{DOM } f): (R \ (f \ X))=((R \ f) \ (R \ X)) \)

Informally, a simple isomorphism between two domains specifies a 1-1 correspondence between entities, and a 1-1 correspondence between systems such that corresponding systems produce corresponding output given corresponding arguments. An isomorphism between domains should specify a 1-1 correspondence between composition functions such
that corresponding composition functions produce corresponding systems given corresponding input. Formally, if \(<\text{Re},\text{Rf},\text{Rs}>\) is a simple isomorphism between Domain1 and Domain2 then \(R=<\text{Re},\text{Rf},\text{Rs},\text{Rc}>\) is a domain isomorphism between Domain1 and Domain2 if the following conditions hold.

(1) \(\text{Rc}\) is a function.
(2) \((\text{DOM } \text{Rc})=\text{C1}\)
(3) \((\text{IMAGE } \text{Rc})=\text{C2}\)
(4) \((\text{Rc } X)=(\text{Rc } Y) \Rightarrow (X=Y)\).
(5) for all \(f \in \text{C1}:
\quad - <X_1,...,X_n> \in (\text{DOM } f) \iff <(R X_1),..., (R X_n)> \in (\text{DOM } (\text{Rc } f))\)
\quad - for all \(X=<X_1,...,X_n> \in (\text{DOM } f):
\quad \quad (R (f X)) = ((R f) (R X))\)

Clause (5) is not identical to its counterpart in the definition of simple isomorphism because of the following difference between systems and composition functions: the argument to a system is an entity in the domain whereas the argument to a composition function is a tuple which, although made up of entities and systems in the domain, is not an entity in the domain. If \(C \in \text{C1}, X=<X_1,...,X_n> \in (\text{DOM } C),\) and \(Y=<Y_1,...,Y_n>\) then the notation \((R <X,Y>)\) may be used a shorthand for \((R <X_1,Y_1>) \& \& ... \& (R <X_n,Y_n>)\) whenever it is unambiguous, i.e., whenever \(X\) is not an entity. With this notation a domain isomorphism is a 4-tuple \(R=<\text{Re},\text{Rf},\text{Rc},\text{Rs}>\) such that

(1) \(R\) is a function.
(2) \((\text{DOM } \text{Re})=\text{E1}, (\text{DOM } \text{Rf})=\text{F1}, (\text{DOM } \text{Rs})=\text{S1},\) and \((\text{DOM } \text{Rc})=\text{C1}\).
(3) \((\text{IMAGE } \text{Re})=\text{E2}, (\text{IMAGE } \text{Rf})=\text{F2}, (\text{IMAGE } \text{Rs})=\text{S2},\) and \((\text{IMAGE } \text{Rc})=\text{C2}\).
(4) \((\text{R } X)=(\text{R } Y) \Rightarrow (X=Y)\).
(5) for all \(f \in \text{F1} \cup \text{S1} \cup \text{C1}:
\quad - x \in (\text{DOM } f) \iff (\text{R } X) \in (\text{DOM } (\text{R } f))\)
\quad - for all \(x \in (\text{DOM } f): (\text{R } (f X)) = ((\text{R } f) (\text{R } X))\)

The Structure of a System

The structure of a system may be defined in several ways. The main objective of the present chapter is the development of a general formal framework in which representation and related issues can be defined. For this purpose it is desirable to use a definition of structure such that
The following definition of structure is used. With each $K \in E \cup F \cup C$ there is associated a unique structure symbol $[K]$. $(\text{STRUCTURE } K)$ is defined to be $[K]$. Each $s \in S$ has a definition of the form $(c <X_1, \ldots, X_n>)$, where $c \in C$ and either $X_i \in E \cup F$ or $X_i$ is a definition. $(\text{STRUCTURE } s)$ is defined to be $[[c] (\text{STRUCTURE } X_1) \ldots (\text{STRUCTURE } X_n)]$. The set of all structures in domain $D$ is $(\text{STRUCTURES } D)$.

The preceding chapter implicitly defined a domain in which there are two types of primitive function -- performance history update functions and hypothesis selection functions -- and three types of entity -- sets of candidates, performance histories and items of performance information. According to the abstract model, if PHupdate is performance history update function, Hselection a hypothesis selection function, and Cset a candidate set, then the system with the structure $[[\text{SEQUENCE}] \ [\text{PHupdate}] \ [[\text{BIND_ARG1}] \ [\text{Hselection}] \ [\text{Cset}]]]$ is a learning system.

The notion of structure just defined may be contrasted with those in which the structure of a system depends on the meaning of the composition functions used in its definition. For example, according to these latter notions the structure of $s - (\text{WHILE } <F_c, F_b>)$ is the infinite tree

```
(F_c X)
 /   \
X    (Fc (F_b X))
 /   \
(F_b X) (Fc (F_b (F_b X)))
 /   \
(F_b (F_b X)) etc.
```

Composition functions are thus viewed in two ways: as second-order functions that compose first-order functions, and as functions that compose structures. This type of structure is useful in analyzing the behaviour of systems. A structure of this type is defined in
Chapter 7 for the purpose of analyzing the efficiency of learning systems. For present purposes the simpler notion of structure defined above is more suitable.

Given a domain relation $R$ between the entities and functions in Domain1 and those in Domain2 the relation, $R_{\text{STRUCT}}$, between the structures in Domain1 and Domain2 is defined as follows.

$$(R_{\text{STRUCT}} <[P_1],[\text{STRUCTURE } P_2]>) \iff (P_1 \in E_1 \cup F_1) \& (P_2 \in E_2 \cup F_2 \cup S_2) \& (R <P_1,P_2>)$$

$$(R_{\text{STRUCT}} <[c_1],[c_2]>) \iff (c_1 \in C_1) \& (c_2 \in C_2) \& (R <c_1,c_2>)$$

& (for all $X \in \text{DOM } c_1$ and for all $Y=<Y_1,...,Y_n>$):

$$R <X,Y> \Rightarrow Y \in \text{DOM } c_2$$

if $A=[c_1] \cup ... \cup [c_n] \in \text{STRUCTURES Domain1}$ and $B=[c_2] \cup ... \cup [c_n] \in \text{STRUCTURES Domain2}$ then

$$(R_{\text{STRUCT}} <A,B>) \iff (R_{\text{STRUCT}} <[c_1],[c_2]>) \& (R_{\text{STRUCT}} <U_1,V_i>) \quad (\text{for all } i)$$

A domain relation $R$ is structure preserving if every $Y$ corresponding to $X$ has a structure corresponding to the structure of $X$. If $R$ is structure preserving and some of the primitive functions in Domain1 correspond to structured functions in Domain2, $R$ is structure elaborating. Formally,

if $X \in E_1 \cup F_1 \cup C_1 \cup S_1$, $R$ preserves the structure of $X$ if, for all $Y$,

$$(R <X,Y>) \Rightarrow (R_{\text{STRUCT}} <(\text{STRUCTURE } X),(\text{STRUCTURE } Y>))$$

$R$ is structure preserving if, for all $X \in E_1 \cup F_1 \cup C_1 \cup S_1$, $R$ preserves the structure of $X$

$R$ is structure elaborating if $R$ is structure preserving

and there exist $f \in F_1$ and $S \in S_2$ such that $(R <f,S>)$

Domain isomorphisms are structure preserving, but simple isomorphisms are not because for all $X \in S_1$ $(R_{\text{STRUCT}} <(\text{STRUCTURE } X),(\text{STRUCTURE } (R \ X)>)$ is false.

If $X=\{c_1 <X_1,...,X_n>, (R <X,Y>)$, and $\neg (R_{\text{STRUCT}} <(\text{STRUCTURE } X),(\text{STRUCTURE } Y>))$ then $X_1...X_n$ are merged by $R$ in $Y$. For example, if $(R <X,Y>)$, where $X$ is the structured function $(f_2(f_1,Z))$ in Domain1 and $Y$ is a primitive function in Domain2, then $f_1$ and $f_2$ are merged in $Y$ by $R$. 

- 27 -
3.2 Full Representation

Intuitively, Domain1 is represented (through domain relation R) in Domain2 only if R satisfies certain requirements. Consider the following diagram in which f is a function in Domain1, \( X \in (\text{DOM } f) \), G is a function corresponding to f (according to R), Z corresponds to X, and Y corresponds to (f X).

\[ X \xrightarrow{R} Z \]
\[ f \quad \downarrow \quad G \]
\[ (f X) \xrightarrow{R} Y \]

An essential property of a "representation" is that Y and (G Z) must be equivalent. Definitions of "representation" may differ in the details of the definition of "equivalence" but all will include a requirement of this type, called a fidelity requirement. The strongest possible fidelity requirement is that \( Y = (G Z) \). Domain isomorphisms satisfy this requirement by definition:

(5) for all \( f \in F \cup S \cup C \):
- \( X \in (\text{DOM } f) \Leftrightarrow (R X) \in (\text{DOM } (R f)) \)
- for all \( X \in (\text{DOM } f) \):
  \[ (R (f X)) = ((R f) (R X)) \]

There are requirements other than the fidelity requirement. For example, it seems reasonable to require all types of functions and entities in Domain1 to be represented in Domain2. Simple isomorphisms fail to satisfy this requirement.

In this section the definition of "full representation" is developed by progressively weakening the definition of domain isomorphism. If domain relation R satisfies the final definition then Domain1 is fully-represented (through R) in Domain2; the pair \(<R, \text{Domain2}>\) is a full representation of Domain1. If Y in Domain2 corresponds (according to R) to X in Domain1 then
X is an interpretation of Y,
Y is a full representation of X,
Y is interpreted as X,
X is fully represented as Y,
and Y fully represents X.

The initial definition is that of domain isomorphism.

(REQ.1) Re, Rf, Rs, and Rc are functions.
(REQ.2) (DOM Re)=E1, (DOM Rf)=F1, (DOM Rs)=S1, and (DOM Rc)=C1.
(REQ.3) (IMAGE Re)=E2, (IMAGE Rf)=F2, (IMAGE Rs)=S2, and (IMAGE Rc)=C2.
(REQ.4) (R X)=(R Y)⇒(X=Y)
(REQ.5) for all f∈(F1∪S1∪C1):
   (a) X∈(DOM f) ⇔ (R X)∈(DOM (R f))
   (b) for all X∈(DOM f): (R (f X))=((R f) (R X))

The first step towards the definition of full representation is to weaken REQ.3 to

(REQ.3.1) (IMAGE Re)⊆E2, (IMAGE Rf)⊆F2, (IMAGE Rs)⊆S2, and (IMAGE Rc)⊆C2.

This permits Domain2 to contain uninterpreted entities (functions), i.e., entities (functions) that do not represent entities (functions) in Domain1. For example it permits a domain whose entities are integers to be represented in a domain whose entities are reals.

Similarly, REQ.2 is weakened to

(REQ.2.1) (DOM Re)⊆E1, (DOM Rf)⊆F1, (DOM Rs)⊆S1, and (DOM Rc)⊆C1.

This permits Domain1 to contain unrepresented entities (functions), i.e., entities (functions) that are not represented in Domain2. The motivation for this relaxation is that a representation of a domain need only reflect the salient aspects of the domain. For example, in the domain of chess it is not necessary to represent all possible board configurations if one is interested in studying endgames involving four specific pieces. [Utgoff 1986] describes the use of a candidate set in Domain2 in which not all the hypotheses in Domain1 are represented as "a very practical method for focusing the concept learner on a set of preferred hypotheses" (p. 111). For most purposes it is also not necessary to represent all the functions in a domain. Choosing which "attributes", i.e. functions, to represent is an important part of the formulation of a learning task.
REQ.5 must be amended to apply only to represented entities and functions.

\[(REQ.5.1) \text{ for all } f \in (\text{DOM Rf}) \cup (\text{DOM Rs}) \cup (\text{DOM Rc})
\]
\[(a) \quad X \in (\text{DOM} \downarrow f) \iff (R X) \in (\text{DOM} (R f))
\]
\[(b) \quad \text{for all } X \in (\text{DOM} \downarrow f) : (R (f X)) = ((R f) (R X))
\]

where \((\text{DOM} \downarrow f)\) is defined to be
\[
f \in F_{1 \cup S_1} : (\text{DOM} f) \cap (\text{DOM} \Re)
f \in C_1 : \{X_1, ..., X_n \in (\text{DOM} f) | \text{for all } i \} \quad X_i \in (\text{DOM} \Re) \cup (\text{DOM} Rf) \cup (\text{DOM} Rs)\]

For consistency of numbering the unchanged requirements are given new names,

\(REQ.1.1 = REQ.1\) and \(REQ.4.1 = REQ.4\).

Alternative Representations

At this point there is a 1-1 correspondence between the represented entities (functions) in Domain1 and the interpreted entities (functions) in Domain2. An important property of a 1-1 correspondence is that each entity (function) in Domain2 has a unique interpretation. In practical terms this allows one to translate a function and specific argument values in Domain1 into their counterparts in Domain2, apply the function to the arguments in Domain2, and un ambiguously translate the computed value back into Domain1. The 1-1 correspondence requirement can be weakened to a unique interpretation requirement, as follows.

First, \(REQ.1.1\) is weakened so that there may be several entities (functions) in Domain2 corresponding to an entity (function) in Domain1.

\[(REQ.1.2) \quad \Re, \Rf, \Rs, \text{ and } \Rc \text{ are relations.}\]

If nothing else, this change requires rewriting of all the requirements in the notation for relations rather than that for functions. The following conventions minimize the disfigurement involved in this rewriting.
define
\((R^* X) = \{Y | (R < X, Y>)\}\)
\((\text{DOM}^* \text{Re}) = \{e_1 \in \text{E1} | \text{there exists } e_2 \in \text{E2} \text{ such that } (\text{Re} < e_1, e_2>)\}\)
\((\text{DOM}^* \text{Rf}), (\text{DOM}^* \text{Rs}), \text{ and } (\text{DOM}^* \text{Rc}) \text{ are defined similarly.}\)
\((\text{IMAGE}^* \text{Re}) = \{e_2 \in \text{E2} | \text{there exists } e_1 \in \text{E1} \text{ such that } (\text{Re} < e_1, e_2>)\}\)
\((\text{IMAGE}^* \text{Rf}), (\text{IMAGE}^* \text{Rs}), \text{ and } (\text{IMAGE}^* \text{Rc}) \text{ are defined similarly.}\)

redefine \((\text{DOM} \downarrow f)\) as
\(f \in \text{F1} \cup \text{S1}: (\text{DOM} f) \cap (\text{DOM}^* \text{Re})\)
\(f \in \text{C1}: (X = X_1 \ldots X_n, > e (\text{DOM}^* f)) \text{ for all } I \) \(X_e \in (\text{DOM}^* \text{Re}) \cup (\text{DOM}^* \text{Rf}) \cup (\text{DOM}^* \text{Rs})\)

REQ.2.1 and REQ.3.1 are now
\((\text{REQ.2.2}) (\text{DOM}^* \text{Re}) \subseteq \text{E1}, (\text{DOM}^* \text{Rf}) \subseteq \text{F1}, (\text{DOM}^* \text{Rs}) \subseteq \text{S1},\) \text{ and } (\text{DOM}^* \text{Rc}) \subseteq \text{C1}.
\((\text{REQ.3.2}) (\text{IMAGE}^* \text{Re}) \subseteq \text{E2}, (\text{IMAGE}^* \text{Rf}) \subseteq \text{F2}, (\text{IMAGE}^* \text{Rs}) \subseteq \text{S2}\) \text{ and } (\text{IMAGE}^* \text{Rc}) \subseteq \text{C2}.

The alternative representations of \(X\) are the entities (functions) \(Z\), in Domain2 such that \((R < X, Z>)\). The requirement that every interpreted entity (function) in Domain2 has a unique interpretation is
\((\text{REQ.4.2}) \text{ for all } Z: (R < X, Z>) \text{ and } (R < Y, Z>) \Rightarrow (X=Y)\)

A consequence of this requirement is that the reflexive, symmetric relation
\((\text{SAME}_\text{INTERPRETATION} Z_1, Z_2)\)
\(\iff \) \(\text{for some } X: (R < X, Z_1>) \text{ and } (R < X, Z_2>)\)
\(\text{OR } \text{for all } X: (\neg (R < X, Z_1>) \text{ and } \text{for all } X: (\neg (R < X, Z_2>))\)

is also transitive and thus an equivalence relation.

In the following diagram \(f\) is a represented function in Domain1 and \(X \in (\text{DOM} \downarrow f)\). \(Z_1, Z_2\ldots\)
are the alternative representations of \(X\), and \(Y_1, Y_2\ldots\) are the alternative representations of \((f X)\). \(G_i\) is one of \(G_1, G_2\ldots\), the alternative representations of \(f\).
The fidelity requirement is that if \( Z \) and \( G \) represent \( X \) and \( f \), then \((G, Z)\) represents \((f, X)\).

In other words, \(((G, Z) | (R < t, G >) & (R < X, Z >))\) must be a subset of \([Y_1 ... Y_m]\), the alternative representations of \((f, X)\). Formally,

\[
(REQ.5.2) \text{ for all } f \in (DOM^* Rf) \cup (DOM^* Rg) \cup (DOM^* Rc):
\]

for all \( G \in (R^* f):\)

(a) \( X \in (DOM \downarrow f) \iff (R^* X) \subseteq (DOM G) \)

(b) for all \( X \in (DOM \downarrow f)\): \( (R^* (f X)) \supseteq ((G Z) | Z \in (R^* X)) \)

As an example of alternative representations consider

**Domain1**
- entities: finite sets of integers, True, False
- functions: union, set-equality, and identity

**Domain2**
- entities: finite lists of integers, True, False
- functions: list-equality, same-members, remove-duplicates, sort-ascending, concatenate, and
  - \( \text{merge } [] \quad Y = Y \)
  - \( \text{merge } X \quad [] = X \)
  - \( \text{merge } X \quad Y = \text{if } (\text{head } X) < (\text{head } Y) \)
    - then \( \text{(cons } (\text{head } X) \quad \text{(merge } (\text{tail } X) \quad Y)) \)
    - else \( \text{(cons } (\text{head } Y) \quad \text{(merge } X \quad (\text{tail } Y)) \)

Two of the possible representations of Domain1 in Domain2 are \( \text{REP}_{\text{SORT}} \) and \( \text{REP}_{\text{UNSORT}} \)

defined as follows. In \( \text{REP}_{\text{SORT}} \) sets are represented by lists sorted in ascending order with
no duplicate entries. Thus there is a unique list representing every set. Many lists have no
interpretation. In \( \text{REP}_{\text{UNSORT}} \) a set is represented by any list whose members are exactly
the elements of the set. The list need not be ordered and it may contain duplicate
members. Thus, each set in Domain1 has a countably infinite set of alternative
representations and every list has an interpretation.

In both representations the merge function represents set union. Concatenation is an
alternative representation of set union in \( \text{REP}_{\text{UNSORT}} \) but is uninterpreted in \( \text{REP}_{\text{SORT}} \)
because it sometimes results in lists that have no interpretation. In both representations
same-members represents set-equality. List-equality is an alternative representation of set-equality
in \( \text{REP}_{\text{SORT}} \) but is uninterpreted in \( \text{REP}_{\text{UNSORT}} \) because

- 32 -
-- if $Z_1$ and $Z_2$ are alternative representations for an entity in Domain1 then (LIST-EQUALITY $<Z_1, Z*>$) and (LIST-EQUALITY $<Z_1, Z*>$), i.e. True and False, must be alternative representations of an entity in Domain2, and
-- there is no entity in Domain1 for which True and False (in Domain2) are alternative representations.

In both representations sort-ascending and remove-duplicates are alternative representations of the identity function on sets. The reasons are subtly different. In $\text{REP}_{\text{SORT}}$ (IMAGE* Re) is the set of lists that are sorted in ascending order and contain no duplicates. When restricted to lists in this set, sort-ascending and remove-duplicates are true identity functions, i.e. they return their argument unchanged. Their behaviour on other lists is irrelevant. In $\text{REP}_{\text{UNSORT}}$ the behaviour of sort-ascending and remove-duplicates on all lists is relevant because (IMAGE* Re) contains all lists. The two functions represent the identity function on sets because when applied to any list $L$ they produce a list that, although not always identical to $L$, always represents the same set as $L$.

**Corepresentation**

The definition is next weakened to permit several entities (functions) in Domain1 to be represented by the same entity (function) in Domain2. Define

$$(\text{COREP } X_1, X_2)$$

$$\iff (\text{for some } Z: (R <X_1,Z>) \& (R <X_2,Z>))$$

$$\text{OR (for all } Z: \neg (R <X_1,Z>) \& \text{ for all } Z: \neg (R <X_2,Z>))$$

If $(\text{COREP } X_1, X_2)$ then $X_1$ and $X_2$ are **corepresented**. Introducing corepresentation entails weakening $\text{REQ.4.2}$, the unique interpretation requirement: if $(R <X_1,Z>)$ and $(R <X_2,Z>)$ then $Z$ has a set of interpretations $\{X_1, X_2\}$. In order for alternative representations, $Z_1$ and $Z_2$, to be genuine alternatives, i.e. freely substitutable, the set of interpretations of $Z_1$ must be the same as that of $Z_2$. Formally,

$$(\text{REQ.4.3}) \quad (\text{COREP } X_1, X_2) \Rightarrow \text{for all } Z: (R <X_1,Z>) \iff (R <X_2,Z>)$$

Consequences of this requirement are
COREP is an equivalence relation. Its equivalence classes are called corepresentation classes. The set of interpretations of an entity (function) in Domain2 is a corepresentation class.

SAME_INTERPRETATION remains an equivalence relation.

(f X₁)=Y₁ & (f X₂)=Y₂ & (COREP X₁ X₂) ⇒ (COREP Y₁ Y₂).

The fidelity requirement at this point is

(REQ.5.2) for all f∈(DOM\star Rf)∪(DOM\star Rs)∪(DOM\star Rc):
for all G∈(R* f):
(a) X∈(DOM↓ f) ⇒ (R* X)⊆(DOM G)
(b) for all X∈(DOM↓ f): (R* (f X))⊆{(G Z)|Z∈(R* X)}

Part (a) requires a strict correspondence between (DOM↓ f) and (DOM G): if f is defined for X then G must be defined for the representations of X, and if G is defined for the representations of X then f must be defined for X. This restricts corepresentation. X₁ and X₂ cannot be corepresented if there is a function defined on X₁ but not on X₂, and functions f₁ and f₂ cannot be corepresented if (DOM↓ f₁)≠(DOM↓ f₂). This restriction runs counter to common practice. In the definition of full representation (a) is weakened to

if f is defined for X then G must be defined for the representations of X.

Formally,

for all f∈(DOM\star Rf)∪(DOM\star Rs)∪(DOM\star Rc):
for all G∈(R* f):
(a) X∈(DOM↓ f) ⇒ (R* X)⊆(DOM G)
(b) for all X∈(DOM↓ f): (R* (f X))⊆{(G Z)|Z∈(R* X)}

or equivalently,

(REQ.5.3) for all f∈(DOM\star Rf)∪(DOM\star Rs)∪(DOM\star Rc):
for all G∈(R* f) and for all X∈(DOM↓ f):
(R* X)⊆(DOM G) & (R* (f X))⊆{(G Z)|Z∈(R* X)}

In practice, corepresentation is used in two different ways. In the first, two entities (functions) are corepresented because the distinction between them is irrelevant for the task at hand. For example, two states in a finite state machine may be corepresented if doing so does not affect the behaviour of the finite state machine, where behaviour may be either the language accepted or the output sequence produced (pp. 68-70, [Hopcroft and Ullman
As a second example, in learning systems in which entities are represented by attribute-values, the attributes and values that are used often corepresent entities that require identical processing in the learning and performance tasks.

Studies of representation and change of representation in problem-solving emphasize the corepresentation of states. For example, the representation of the state-space of the Towers-of-Hanoi puzzle is discussed in [Korf 1980]. Figure 3.1 is the state-space for two disks and operations that move a disk in one direction only (from peg A to peg B, B to C, and C to A). "1" denotes the smaller disk, "2" the larger. $M_1$ denotes moving disk 1 from its current position to the next peg in the sequence; $M_2$ denotes moving disk 2. "1,2:" denotes the state in which disks 1 and 2 are on the first peg, A, and no disks are on the other pegs; "::2:1:" denotes the state in which no disks are on peg A, disk 2 is on peg B, and disk 1 is on peg C.

*Figure 3.1.* The 2-function state-space of the Towers-of-Hanoi puzzle.
This state-space can be represented in the 1-function state-space

\[
\begin{align*}
\text{M} & \quad \text{M} \\
\text{:2::} & \quad \text{:2::} \\
\text{:2::} & \quad \text{M} \\
\end{align*}
\]

Disk 1 is not represented. States which differ only in the position of disk 1 are corepresented: for example, ":2:::" represents ":1,2:::" , ":2:1:::" and ":2::1:" . \( M_1 \) is represented by the identity function (not shown) and \( M_2 \) by move \( M \). Note that \( M_2 \) is undefined on two of the three states that are corepresented. This is typical of the use of corepresentation in representing state-spaces.

The second way in which corepresentation is used is somewhat counterintuitive. Entities (or functions) whose distinction is essential for the task are corepresented. For example, two types of entity, positive integers and sets of integers of the form \{1,...,N\}, can both be represented by positive integers. In this representation the integer \( N \) and the set \{1...N\} are corepresented, and the functions \( N_1 \leq N_2 \), \( N_1 \in \{1...N_2\} \), and \( \{1...N_1\} \subset \{1...N_2\} \) are corepresented. The corepresentation of candidates and sets of candidates is quite common in machine learning. The corepresentation of distinct entities and functions is most evident in [Connell and Brady 1987]. All types of entity in Domain1 are represented with the same type of entity (a Grey code) in Domain2, and entities of different types are corepresented. As a result "the half dozen or so induction heuristics developed by Michalski and others" (p.173) can be corepresented.

REQ.5.3 allows functions \( f_1 \) and \( f_2 \) to be corepresented even though \((\text{DOM} \downarrow f_1) \neq (\text{DOM} \downarrow f_2)\). This use of corepresentation can be illustrated in the Towers-of-Hanoi state-space in Figure 3.2. In this state-space there are six functions, 1A, 2A, 1B, 2B, 1C, and 2C: their names indicate the disk moved and the peg to which it is moved.
This 6-function state-space can be represented in the 2-function state-space (Figure 3.1) by corepresenting 1A, 1B, and 1C with $M_1$, and 2A, 2B, 2C with $M_2$. The opposite is not true. There is no full representation of the 2-function state-space in the 6-function state-space. However there is a sense in which 1A, 1B, 1C collectively represent $M_1$: it is this notion that is captured in distributed representation, defined below.

**Representing Systems With Systems**

The restriction that primitive functions must be represented by primitive functions and that structured functions must be represented by structured functions can be dropped by replacing $R_f$ and $R_s$ by a single relation $R_{sys}$. This substitution results in the definition of full representation.
Domain1 is fully represented (through $R=\langle R_e, R_{sys}, R_c \rangle$) in Domain2 iff

(REG.1.4) $R_e, R_{sys},$ and $R_c$ are relations

(REG.2.4) $(\text{DOM}^* R_e) \subseteq E_1, (\text{DOM}^* R_{sys}) \subseteq F_1 \cup S_1,$ and $(\text{DOM}^* R_c) \subseteq C_1.$

(REG.3.4) $(\text{IMAGE}^* R_e) \subseteq E_2, (\text{IMAGE}^* R_{sys}) \subseteq F_2 \cup S_2,$ and $(\text{IMAGE}^* R_c) \subseteq C_2.$

(REG.4.4) $(\text{COREP} \: X, X_\mu) \Rightarrow \text{for all } Z: (R < X, Z >) \iff (R < X_\mu, Z >)$

(REG.5.4) for all $f \in (\text{DOM}^* R_{sys}) \cup (\text{DOM}^* R_c)$:

- for all $G \in (R^* f)$ and for all $X \in (\text{DOM} \downarrow f)$:
  
  $(R^* X \subseteq (\text{DOM} G)) \& (R^* (f X)) \supseteq \{(G Z) | Z \in (R^* X)\}$

If $R$ is also structure preserving then the representation of Domain1 in Domain2 is a literal representation.

The representation of structured functions in Domain1 by primitive functions in Domain2 is a generalization of the use of macro-operators to change the organization of state-space.

Consider the 3-function version of the Towers-of-Hanoi state-space in Figure 3.3. As before, all moves are in the same direction, from peg A to peg B to peg C. The primitive functions in this state-space, A, B, and C, indicate the peg to which a disk is moved.

![Figure 3.3. The 3-function state-space of the Towers-of-Hanoi puzzle.](image-url)
Unlike the 2-function state-space (Figure 3.1) it is not possible to represent the 3-function state-space in the 1-function state-space

Let AB denote the structured function in which functions A and B are sequentially composed: \((AB \ x) = (B \ (A \ x))\). The effect of AB is to move the disk on peg C to peg A and then to peg B. BA, on the other hand, moves two disks: the disk on A is moved to B and the disk on C is moved to A. The state-space in Figure 3.4 is produced by replacing functions A, B, and C by all two-function sequences, AB, AC, etc. This state-space can be represented in the 1-function state-space. States that differ only in the position of disk 1 are corepresented as before. Functions CA, BC, and AB are corepresented by the identity function. Functions AC, CB, and BA are corepresented by the function M.

\[\begin{array}{c}
\text{M} \\
\text{M} \\
\text{M} \\
\text{M} \\
\text{M}
\end{array}\]

**Figure 3.4.** The macro-operator state-space of the Towers-of-Hanoi puzzle.
3.3 Distributed Representation

In a full representation, X in Domain1 may have alternative representations but the alternatives must be freely substitutable. That is, it must be possible to use any one of them everywhere a representation for X is required. In a distributed representation X is represented by a collection of entities (functions) that are not freely substitutable. The entities in the collection jointly represent X; the representation of X is distributed across the collection. For example, the absolute value function is jointly represented by the functions

\[(f_1 X) = X \text{ if } X > 0 \text{ and undefined otherwise}\]
\[(f_2 X) = -X \text{ if } X \leq 0 \text{ and undefined otherwise}\]

In the following diagram f is a represented function in Domain1 and \(X \in (\text{DOM} \downarrow f)\). \(Z_1, Z_2, \ldots\) are the representations of X, and \(Y_1, Y_2, \ldots\) are the representations of \((f X)\). \(G_i\) is one of \(G_1, G_2, \ldots\), the representations of f.

\[
\begin{array}{ccc}
X & \xrightarrow{R} & Z_1 \\
\downarrow{f} & & \downarrow{G_1} \\
(f X) & \xrightarrow{R} & Y_1 \\
\end{array}
\]

\[
\begin{array}{ccc}
Z_2 & \cdots & Z_j & \cdots \\
\downarrow{G_1} & & \downarrow{G_1} & \cdots \\
(G_1 Z_1) & \cdots & (G_1 Z_j) & \cdots \\
\downarrow{(G_1 Z_2)} & & \downarrow{(G_1 Z_j)} & \cdots \\
Y_2 & \cdots & Y_k & \cdots \\
\end{array}
\]

In a full representation every \((G_i Z_j)\) must be defined and represent \((f X)\). In a distributed representation this is weakened to

(a) at least one \((G_i Z_j)\) must be defined, and

(b) every \((G_i Z_j)\) that is defined must represent \((f X)\).

Formally,

\[\text{(REQ.5.5) for all } f \in (\text{DOM}^* \text{ R}_{\text{sys}}) \cap (\text{DOM}^* \text{ R}_c): \]
\[\text{for all } X \in (\text{DOM} \downarrow f): \]
\[(a) (R^* X) \cap \{Y | Y \in (\text{DOM} \ G) \& (R < f, G>) \neq \{\} \]
\[(b) (R^* (f X)) \supset \{ (G Z) | (R < X, Z>) \& (R < f, G) \& Z \in (\text{DOM} G) \} \]
The domain relation in which 1A, 1B, 1C jointly represent \( M_1 \) and 2A, 2B, 2C jointly represent \( M_2 \) is a distributed representation of the 2-function Towers-of-Hanoi state-space in the 6-function state-space.

In weakening the fidelity requirement in this way an important property has been lost. In a full representation if \( (f_2, f_1, X) \) is defined in Domain1 then \( (g_2, g_1, Y) \) is defined in Domain2, for any choice of \( g_2, g_1, \) and \( Y, \) representing \( f_2, f_1, \) and \( X. \) In other words, the composability of \( f_2 \) and \( f_1 \) in Domain1 guarantees the composability of \( g_2 \) and \( g_1 \) in Domain2. This is because if \( (f_1, X) \) is in \( (\text{DOM } f_2) \) then all representations of \( (f_1, X) \) must be in \( (\text{DOM } g_2). \) A distributed representation requires that some, but not necessarily all, of the representations of \( (f_1, X) \) be in \( (\text{DOM } g_2). \) Composability of \( f_2 \) and \( f_1 \) in Domain1 does not guarantee composability of \( g_2 \) and \( g_1 \) in Domain2, because there is no guarantee that \( (g_1, Y) \) will be one of the representations of \( (f_1, X) \) that is in \( (\text{DOM } g_2). \)

In some learning systems different representations of the same entity are used for different purposes. For example, in the version of ID3 in [Quinlan 1982] a hypothesis is represented as a decision tree for the purpose of classifying examples, and as a set of seen examples (a "window") for the purpose of hypothesis selection. The composability problem is overcome in such systems in one of two ways. First, in these systems functions are composed in specific ways; it is irrelevant whether or not other compositions are possible. In ID3 functions defined only on windows are never applied to the results of functions that produce decision trees. Secondly, these systems employ "identity" functions that translate one representation of an entity to another. ID3 includes a function that translates windows into decision trees.

The model of learning systems defined in Chapter 2 involves the composition of three processes, performance history update, hypothesis selection, and hypothesis application. The composability problem is therefore a significant consideration in choosing representations for performance histories, hypotheses, and performance information. Different approaches to this problem are discussed in Chapter 5.
3.4 The Declarative Representation of Functions

A declarative representation of a collection of functions $f_1,f_2,...$ is a collection of entities $e_1,e_2,...$ and a single function $\Psi$, called an interpreter, such that $(\Psi <e_k,X>)=(f_k X)$ for all $k$ and all $X$. $e_k$ is the declarative form of $f_k$; the set $\{e_1,e_2,...\}$ is a representation language. For example, the Boolean functions on $N$ variables can be represented by binary arrays of length $2^N$ and a "lookup" function that, given array $A$ and arguments $v_1,...v_N$, returns the entry $A[V]$, where $V$ is the positive integer whose binary representation is $v_1,...v_N$. Array $<0,0,0,1>$ represents (AND $<V_1,V_2>$), $<0,1,1,1>$ represents (OR $<V_1,V_2>$) etc.

The declarative form of a function $f_1$ is an entity in Domain2. Like any entity in Domain2 it may have multiple interpretations. It may represent an entity in Domain1 or it may be the declarative form of other functions in Domain1. $f_1$ is partially corepresented with such entities and functions. It is common for an example to be partially corepresented with the function that returns true exactly when given that example.

In a learning system candidates are systems for the performance task, and the set of candidates is almost invariably represented as an interpreter and a set of declarative forms. In ID3, the declarative forms are decision trees and the interpreter is a function that takes a decision tree and an example and produces a classification for that example. In the system INDUCE the declarative forms are expressions in the "Annotated Predicate Calculus" ("APC")[Michalski 1983] and the interpreter is a function that takes two expressions (examples are also represented as APC expressions) and produces a classification. In a two-tiered representation[Michalski 1987] a set of functions is represented by a set of entities $e_1,e_2,...$ and a set of interpreters, and the set of interpreters is represented by a set of entities $I_1,I_2,...$ and an interpreter. The declarative form of a function in a two-tiered representation is a pair $<e_i,I_k>$. 
Declarative forms are usually chosen so that particular functions of functions can be represented by functions of declarative forms. For example, many learning systems use declarative forms such that the relation "function $f_1$ is more specific than function $f_2$"\(^1\) is represented by "$e_1 \leq e_2$" where $e_1$ and $e_2$ are the declarative forms of $f_1$ and $f_2$. As a second example, declarative forms are often chosen so that the "minimal common generalization(s)" of two functions\(^2\) is represented by a function on declarative forms.

3.5 Adequacy and Efficiency

As mentioned at the beginning of this chapter, with every task $T$ there is associated a semantic domain, $D_T$. If Domain2 is a domain and $R$ a domain relation between $D_T$ and Domain2 then $<R,\text{Domain2}>$ is adequate for task $T$ if the following conditions hold.

1. $D_T$ is represented through $R$ in Domain2
2. there exists a system $S$ in Domain2 that performs $T$ in the sense that it represents a system in $D_T$ that satisfies $T$'s criterion of success. $T$ can be performed by $S$ in Domain2 if there exists $R$ such that $<R,\text{Domain2}>$ is adequate for $T$ and $S$ performs $T$.

A representation $<R,\text{Domain2}>$ may be adequate for task $T_1$ but inadequate for task $T_2$ even if $D_{T_1}$ is the same as $D_{T_2}$. For example, [Quinlan 1979] reports a representation that is adequate for one classification task and inadequate for another in the same chess endgame domain. It is not uncommon for a representation of hypotheses that is adequate for the performance task to be inadequate for the learning task; an example is discussed in Chapter 6. For a given task $T$ and domain, Domain2, it is often difficult to find an $R$ such that $<R,\text{Domain2}>$ is adequate for $T$ or to determine that no such $R$ exists. This is illustrated in the chapter that follows.

\(^1\) if $(\text{DOM } f_1)=(\text{DOM } f_2)$ and $(\text{IMAGE } f_1)=(\text{IMAGE } f_2)=(\text{True, False})$ then $f_1$ is more specific than $f_2$ if and only if $(f_1 X) \Rightarrow (f_2 X)$ for all $X$.

\(^2\) $f_3$ is a minimal common generalization of $f_1$ and $f_2$ if for all $f_4$ $(f_1 \leq f_4 & f_2 \leq f_4 \Rightarrow -(f_4 \leq f_3))$, where "$\leq$" is "more specific than"
The informal notion of the generality of a system can be defined in terms of adequacy. For example, if S1 and S2 are systems one could define "S1 is more general than S2" if S1 can perform any task that can be performed by S2. Alternatively, given a set of tasks one could define "S1 is more general than S2" if S1 can perform more of the tasks in the set than S2.

Efficiency

The most fundamental property of a representation is that it can make some types of information explicit, and this property can be used to bring the essential information to the foreground allowing smaller and more easily manipulated descriptions to suffice.3

Efficiency can be defined in many different ways. The following is a brief sketch of a general approach to defining efficiency that fits neatly into the formal framework for representation. Efficiency is defined in terms of the cost of applying a function, where cost indicates, for example, the amount of time, storage or other critical resource that is consumed in applying the function. Corresponding to each primitive function f is a function \( f_c \) such that \((\text{DOM } f_c) = (\text{DOM } f)\) and \((f_c X)\) is the cost of computing \((f X)\). The cost of computing \((f X)\) includes the cost incurred, if any, to store the entity, \((f X)\), produced by \(f\). Corresponding to each composition function \(C\) is a function \(C_c\) such that \((\text{DOM } C_c) = (\text{DOM } C)\). If \(S\) is the structured function defined by \((C X)\) then \((C_c X)\) is a function \(S_c\) such that \((\text{DOM } S_c) = (\text{DOM } S)\) and \((S_c X)\) is the cost of computing \((S X)\). There may also be overhead costs associated with a system, such as the cost of storing the functions and entities that define it. This aspect of cost tends to favour representing a set of entities, such as the set of declarative forms of candidates, as a function that produces the entities as needed. The average efficiency of a system, \(S\), can be defined, for example, as a weighted sum over all inputs \(X\) of \((S_c X)\).

The efficiency of systems that perform a task \(T\) is heavily influenced by the choice of representation \(<R, D>\). That is, there may be very efficient systems that perform \(T\) if \(<R_1, D>\) is chosen but not if \(<R_2, D>\) is chosen. Informally, \(<R, D>\) is well-suited to task \(T\) if there

---

3 p. 298 in [Marr 1982]
exists a system in D that performs T with "high" average efficiency. The primary motivation for using a distributed representation is that different representations of an entity or function are well-suited to different tasks.

The relationship between efficiency and choice of representation is sometimes very clear. If $G_1$ computes a given function more efficiently than $G_2$ it is clear that choosing between $G_1$ and $G_2$ influences efficiency. Less obvious are the effects on efficiency of the representation of entities. For example, the efficiency of a function is often much higher on some entities than others; representations in which only high-efficiency entities have an interpretation exploit this non-uniformity to improve efficiency. Furthermore, there is a dependency between the representation of entities and the representation of functions: $G$ will represent $H$ only if entities are represented in a particular way. For example, list-equality represents set-equality for some representations of sets as lists but not for others. The representation of entities therefore determines which functions in the Domain1, if any, will be represented by the most efficient functions in Domain2. As an extreme example, the identity function in Domain2 will represent function $H$ in Domain1 if $X$ and $(H X)$ are corepresented (for all $X$). Some "exemplar-based" learning systems do precisely this ([Kibler and Aha 1987]). The set of seen examples, i.e. the input to the system, is corepresented with the hypothesis that is output by the system. The following chapter and Chapter 7 examine in detail the relationship between task-specific knowledge, representation, and efficiency.
CHAPTER 4

THE EXPLOITATION OF TASK-SPECIFIC KNOWLEDGE

To apply a system to a task, the semantic domain, in which the task is formulated, must be represented in the syntactic domain, in which the system exists. The first section of this chapter illustrates the obstacles that can arise in attempting to apply a system to a task, and how knowledge of the task can be used to overcome these obstacles. The remainder of the chapter surveys the ways in which knowledge about the task can be exploited to construct a representation that is well-suited to the task.

4.1 Example: Overcoming Difficulties Applying ID3 to Tasks Involving Sequences

In the version of ID3 used in this section there is a fixed, finite number of classifications, attributes, and values for each attribute¹. The input to ID3 is a set of classified examples, where an example is a vector containing one value for each attribute. Because there are a finite number of attributes and a finite number of values for each attribute, there are a finite number of possible examples. The output of ID3 is a decision tree in which each leaf is labelled with a classification, each internal node is labelled with an attribute, and emanating from each internal node is one arc corresponding to each possible value of the attribute labelling the node.

ID3 is most easily applied to concept learning tasks, by representing the candidate classification systems declaratively in the form of a decision tree. But it may be applied to other types of task as well. In [Manago and Kodratoff 1987] ID3 is applied to a clustering

¹ Several versions of ID3 are not limited in this way. For example, [Gams and Lavrac 1987] describes versions that
task: each leaf in a decision tree represents a cluster of the given classified examples. ID3 has also been applied to the task of inducing models of processes. In [van Lehn and Garlick 1987] classified examples represent the actions of a human subject solving a problem and the decision tree represents the mental problem-solving strategy employed by the subject.

The difficulties that can arise in applying a system to a task, and the use of task-specific knowledge to overcome these difficulties, can be illustrated by attempting to apply ID3 to the following tasks.

Mass Spectrometry Domain.
Entities: linear molecules
Performance Task.
Input: a molecule
Output: the bonds in the given molecule that will break when the molecule is analyzed in a mass spectrometer
Learning Objective: to extend the range of molecules for which the system succeeds at the performance task
Reference: METADENDRAL

Protein Folding Domain.
Entities: linear sequences of amino acids (i.e. proteins)
Performance Task.
Input: a protein
Output: a sequence of the elements ALPHA, BETA, and TURN that describes the secondary structure of the given protein.
Learning Objective: to extend the range of proteins for which the system succeeds at the performance task
Reference: [King 1987]

Problem Solving Domain.
Entities: states, a set of "goal" states
Functions: operators mapping one state to another
Performance Task.
Input: a state (the "problem")
Output: a sequence of operators (a "solution") mapping the given state to a goal state
Learning Objective: improve the speed with which solutions are found.
References: [van Lehn and Garlick 1987], LEX, PLS

---

2 likewise, the RULEGEN system in METADENDRAL, which is similar to ID3 in many ways, performs a clustering task (p. 436, [Diezterich et al. 1982]).

3 The restriction to linear molecules is for the purpose of the example and not a restriction of METADENDRAL.
Difficulty #1: Limitations of Decision Trees

In attempting to apply ID3 to these tasks, the first difficulty that arises is that decision trees do not seem adequate for representing systems for the performance task. In all the domains performance-task outputs are unbounded in size. Therefore there are an unbounded number of possible performance-task outputs. In a decision tree, the number of classifications -- even the number of paths -- is fixed and finite.

This difficulty may be circumvented by decomposing the performance task into two subtasks, one of which is a classification task. The possible decompositions depend on the task. The performance task in the protein folding domain may be decomposed as follows. Let AA_k denote the amino acid in position k of the given protein. The classification subtask is "In what type of secondary structure will AA_k occur?", with classes ALPHA, BETA, and TURN. If SS_k is the class of AA_k then the performance task is achieved by calculating <SS_1, ..., SS_P>, where P is the length of the given protein. A similar decomposition may be used in the mass spectrometry domain. The classification subtask is "will bond B_k break?" with classes YES and NO, where B_k is one of the bonds in the given molecule. The performance task is achieved by calculating the set of bonds in the molecule whose classification is YES.

The decomposition of performance task in the problem-solving domain is more complex. A possible decomposition is to use the names of operators as classes in the classification subtask "what operator should be applied to the present state?". Let STATE_1 be the given state. The operator named by classifying STATE_1 may be applied to STATE_1 to produce STATE_2. Then the operator named by classifying STATE_2 may be applied to STATE_2 to produce STATE_3. This process can be iterated until a goal state is reached. A variation on this method of decomposition is used by PLS. Let the current state be STATE_i. PLS iterates over all the operators OP_k the classification subtask "How desirable is the state (OP_k STATE_i)?" with classes corresponding to many different degrees of desirability. The
operator leading to the most desirable next state is then applied. In the most common
decomposition of the performance task in the problem solving domain there is a separate
classification subtask, "should OP\(_k\) be applied to the current state?", for each operator OP\(_k\).
These subtasks are executed sequentially until a classification of YES is made. The
operator associated with the first YES is applied to the current state and the process
repeated.

All of the methods of decomposing the performance task into an iteration of classification
subtasks overcome the potential inadequacies of decision trees for representing systems for
the performance task. However, iterative decomposition can also introduce difficulties. For
example, iterative decomposition results in a sequence of classification subtasks.
Obviously, the result of one subtask cannot depend on the results of subtasks later in the
sequence. In domains in which there are strong constraints on the permissible patterns of
"classes", this aspect of decomposition can be unacceptable.

Consider the domain of speech recognition in which the semantic entities (speech
waveforms) are sequences of amplitude measurements and the performance task is to
produce the phoneme sequence corresponding to the waveform. Decomposing the
performance task into independent classifications of successive segments of the waveform
(the classes correspond to individual phonemes) fails to incorporate any of the known inter-
phoneme constraints in this domain (arising at phonemic, syntactic, and semantic levels). In
some domains, this difficulty can be overcome by providing certain properties of the
classification sequence as pseudo-attributes of the current state. This solution will work only
if the classification subtasks can be ordered in such a way that the earlier classifications can
be made, with certainty, without reference to the later classifications. In the speech domain
it is not clear that any fixed ordering of the classification subtasks satisfies this condition.
Difficulty #2: Limitations of Attribute-Value Vectors

The semantic entities in the mass spectrometry and protein folding domains are sequences consisting of an arbitrary number of "simple" entities (atoms, and amino acids). Many of the natural ways of describing these sequences cannot be represented by attribute-value vectors. ID3’s requirement that each attribute have a fixed, finite set of possible values precludes attributes whose range of values increases with the length of the sequence such as "total length of the sequence", "position of simple entity X in the sequence", or "the number of simple entities with property P in the sequence". ID3’s requirement that there be a fixed, finite number of attributes precludes having the number of attributes grow with the size of the sequence. For example, METADENDRAL describes bond environments with the properties of the atoms within a certain radius of the broken bond, and it can dynamically increase the radius up to the size of the entire molecule (in principle). Because the number of properties increases with the radius, and the maximum radius increases with the size of the molecule, in METADENDRAL the number of properties that can be used to describe a semantic entity increases with the size of the entity.

A finite set of attributes each having a finite number of values can distinguish only a finite number of entities. Consequently, it is not possible to represent with a distinct syntactic entity each of an infinite number of semantic entities. Indeed the number of corepresentation classes cannot exceed the number of syntactic entities. The task dictates whether or not this number of corepresentation classes is adequate. In many learning tasks, for example in virtually all language identification tasks, there does not exist an adequate representation having a finite number of corepresentation classes. Perhaps surprisingly, there does exist an adequate representation of the semantic entities onto attribute-value vectors in the mass spectrometry domain (results in the protein folding domain are inconclusive at present). Although METADENDRAL has access to an unbounded number of properties in principle, in practice there is a parameter whose value (usually very small, and always finite) fixes an upper bound on the number of properties that are actually accessed.
4.2 Providing Task-Specific Knowledge Explicitly

Once an adequate representation is chosen, a system may be described in terms of the entities and functions in the semantic domain. To account for this apparent transition of the system from being oblivious to being knowledgeable about the semantic domain, it is natural to view the act of choosing a representation as an act of providing the system with knowledge specific to the task. There is a trivial side to this phenomenon, namely that task-specific terms (e.g., SIZE, LARGE, ELEPHANT) have been substituted for the syntactic ones (e.g., attribute A003, value 11, class $C01$). But there is also a serious side, a genuine provision of task-specific knowledge to the learning system.

There are often parameters associated with a learning system. Specifying a different combination of values for these parameters results in different learning behaviour. The "learning system" is thus an interpreter, and the parameters the declarative forms, that together represent a set of learning systems. Choosing a particular combination of values selects one of the possible learning systems. For example, some recent versions of ID3 have a pruning threshold parameter, $P$, whose value determines the conditions under which a node will be expanded. Clearly, different values of $P$ correspond to different learning systems.

In "knowledge-intensive" systems, certain parameters are identified as "explicit models" of the semantic domain, and the values assigned to these parameters unarguably provide the learning system with task-specific knowledge. But the effect of assigning values to these parameters is the same as the effect of assigning values to the other parameters, namely to select a learning behaviour that is adequate (and perhaps relatively efficient) for the given task. In this sense, the values assigned to all parameters in all systems constitute task-specific knowledge. The assignment of values of parameters is an explicit provision of task-specific knowledge.

---

4 e.g. domain models are prominent parameters of METADENDRAL, AGAPE, and METALEX.
4.3 Providing Task-Specific Knowledge Implicitly

Task-specific knowledge can be provided to a system implicitly through the choice of the representation of semantic domain. Different representations provide different knowledge that results in different interim behaviour and different rates of success at the learning task. Explicit knowledge affects the behaviour of a learning system at the syntactic level; implicit knowledge does not. It affects only the behaviour of the learning system at the semantic level. All learning systems are provided with knowledge implicitly, and in many systems this is by far the more important source of knowledge.

The different explicit and implicit ways of providing task-specific knowledge to a system can be illustrated in the different ways of indicating to a system that certain syntactic entities are uninterpreted, i.e., that they do not represent any semantic entity or function.

Uninterpreted entities of particular interest are those that are elements of a set that represents a semantic set, e.g., the set of (semantic) candidates, the set of possible (semantic) examples, or the set of (semantic) states in a search space. These uninterpreted entities have a peculiar status. On one hand, they are not syntactic candidates (examples, states) because they do not represent semantic candidates (examples, states). On the other hand, they are elements of the set representing the set of semantic candidates (examples, states), and in this sense they are elements of the set of syntactic candidates (examples, states). To make this status clear, these entities will be called uninterpreted candidates (examples, states).

The existence of uninterpreted candidates and examples has different consequences for different learning systems and tasks. For example, performance tasks that involve generating examples implicitly require the examples generated to be interpreted, and therefore require the performance system to distinguish interpreted from uninterpreted examples. Performance tasks that only involve classifying given examples do not require
the performance system to make this distinction. The efficiency of some systems depends on the size of the set of (syntactic) candidates, and is therefore reduced in proportion to the number of uninterpreted candidates. Uninterpreted candidates prevent some systems from detecting that there is a unique (semantic) candidate compatible with the seen examples. The behaviour of some systems, ID3 for example, is not affected by the existence of uninterpreted candidates. Systems that are unaffected by uninterpreted candidates can be used in conjunction with representations in which a very large proportion of the candidates are uninterpreted. For example, in [Quinlan 1983] only $3 \times 10^4$ of the $10^9$ syntactic candidates are interpreted (p. 471).

Knowledge about which entities are interpreted can be provided explicitly in two ways. First, it can be provided in the values of parameters. For example, METADENDRAL is provided with a set of parameter values that constitute a "half-order theory" of mass spectroscopy. Syntactic examples and candidates are uninterpreted if they are inconsistent with the half-order theory. Secondly, knowledge about which entities are interpreted is provided by the seen examples: all the seen examples are interpreted. [Rollinger 1987] proposes that the concept "meaningful example" should be learned in parallel with other concepts, and that uninterpreted examples, classified as "not meaningful", should be provided to a learning system.

There are also two implicit ways to provide knowledge about which entities are interpreted. First, one can choose a representation such that there are very few or no uninterpreted entities. This is the method used in [King 1987] to ensure that the hypotheses produced (in the protein folding domain) correspond to meaningful explanations of the protein folding process. Second, one can choose a representation such that although (many) uninterpreted candidates exist, the learning algorithm always generates interpreted candidates when it is given interpreted examples. Knowledge about which candidates are interpreted is provided to ID3 in this way. [Mitchell 1978] also uses this way of providing knowledge about which candidates are interpreted (p. 140). These different means of implicitly providing task-

---

5 [Bundy 1982, van Someren 1986]. A similar problem arises in Boltzmann machines (p. 9, [Denthick 1984]).
specific knowledge will now be described in detail.

4.3.1 Representing Semantic Entities

Semantic entities can be corepresented if the task permits them to be treated identically. In other words, the task-specific knowledge that a particular set of semantic entities may be treated identically can be provided to a learning system implicitly by choosing a representation that makes it impossible for the learning system to distinguish the semantic entities in the set.

This knowledge is extremely useful to a learning system. The concept learning task is to produce a procedure for deciding which entities are in each of the possible classes; specifying in advance that a large set of semantic entities are in the same class greatly simplifies the learning task. For example, there are roughly 11,000,000 board configurations (semantic entities) in the chess endgame domain of black knight and king versus white rook and king. For the task of classifying a board configuration as a loss by black in 2-ply, the board configurations can be adequately represented with just 30,000 syntactic entities\(^6\). This representation implicitly provides the learning system with the knowledge that certain sets of board configurations (namely, the corepresentation classes) have the same classification. Consequently, each syntactic example given to the learning system informs it of the correct classification of all the board configurations represented by that example (400, on average).

\(^6\) pp. 470-471, [Quinlan 1983]
4.3.2 Representing Semantic Candidates

The choice of representation determines the number of semantic candidates that are represented. Candidates that are not represented, of course, cannot be selected by a learning system. Rendell estimates that in existing concept learning applications 80% of the learning task is accomplished by the choice of representation, leaving only 20% to be accomplished by the learning system.\(^7\)

In addition to determining the number of semantic candidates that are represented, the choice of representation determines which set of semantic candidates is represented. Therefore it determines the proportion of represented candidates that satisfy the selection criterion. As an extreme example, a representation in which every represented candidate satisfied the selection criterion would render the hypothesis selection task trivial. As a less extreme example, suppose the selection criterion is defined in terms of the average cost of a classifying an example, where cost is a function of the attribute measured and the accuracy of measurement required. The knowledge of the costs associated with attributes and the measurement accuracy of values can be provided to some systems explicitly, by assigning appropriate values to parameters. This knowledge can also be provided to any system implicitly, by representing only those candidates whose average cost is less than a certain threshold.

\(^7\) pp. 200-206, [Rendell 1986]. Also pp. 8/12-8/13, [Shapiro 1983].
4.3.3 Exploiting Properties of a Specific Learning System

The uninterpreted aspects of a system are called its bias. For example, the choice of hypothesis is usually underdetermined by the (semantic) selection criterion. Systems are therefore compelled to supplement it with uninterpreted criteria to determine the final choice. The uninterpreted criteria are often systematic at the syntactic level, for example selecting the first entity that satisfies the semantic criterion, or selecting the smallest syntactic entity that satisfies the semantic criterion. Biased selection, being uninterpreted, is tantamount to choosing a candidate blindly in the semantic domain and is correctly described as an "unjustified inductive leap.

No aspect of a system is intrinsically uninterpreted. By choosing an appropriate representation, any aspect of the system can be put into correspondence with an aspect of the semantic domain. For example, the selection criterion used in many systems prefers smaller syntactic entities to larger ones. Some authors do not actively seek to use this syntactic preference to represent a semantic preference. They are content to regard it as a bias, and justify it, as such, on purely syntactic grounds (e.g. efficiency). Others deliberately seek to represent the semantically more desirable candidates with the smaller syntactic entities. In a representation of this type, the size of a syntactic entity represents the desirability of the corresponding candidate, and the preference for smaller strings is not a bias. In choosing a representation in which the syntactic preferences of a system correspond to semantic preferences, these authors are implicitly providing the system with task-specific knowledge.

---

8 "bias" originally [Mitchell 1980] included several interpreted aspects of the system, for example the subset of semantic candidates that are represented in the system. Some authors (e.g. [Rendell 1987]) have extended "bias" to include all "extra-evidential determinants" of a system's behaviour. In this thesis, interpreted aspects of a system are ascribed to its task-specific knowledge (implicit or explicit) and "bias" refers only to the uninterpreted aspects.

9 [Campbell 1960] discusses the distinction between blind and random choice.

10 [Mitchell 1983]. This description is correct only when "bias" is restricted to uninterpreted aspects of a system

11 e.g. [Quinlan 1986, Michalski 1983, Wolff 1982]

12 e.g. pp. 42-45, [Chomsky 1965]; [Gallant 1986]; p. 475, [Georgeff and Wallace 1984]
Choosing a representation such that semantic requirements, preferences, and functions directly correspond to aspects of a system is an extremely effective way of providing task-specific knowledge. This is demonstrated in the following example, and examined in detail in Chapter 7.

This example is based on the notions of "regularity" and "structure" in [Rendell 1987]. The syntactic entities are 4-bit strings. Given a function, U, and a threshold value, T, the system searches for a string, S, such that \((U S) > T\). \((U S)\) is the utility of S. For conducting the search the system has two operators "move left" and its inverse "move right". These operators impose the following linear structure on the search space.

\[
0000 \ 0001 \ 0010 \ 0011 \ 0100 \ 0101 \ 0110 \ 0111 \ 1000 \ 1001 \ 1010 \ 1011 \ 1100 \ 1101 \ 1110 \ 1111
\]

The system's search may be described as "Monte Carlo" hill-climbing. Starting at a randomly chosen string, the system selects the direction with the steeper gradient (unless this leads to a previously visited string) and continues to move in that direction until the utility threshold is exceeded or until doing so would result in a decrease in utility. In the latter case, a new starting point is randomly chosen and the process repeated.

The semantic entities are the integers between 0 and 15, and the semantic utility, \((U^* X)\), is given by the formula 8 - \(|X-7|\). The semantic utility function \((U^* X)\) is very well-behaved; it increases monotonically from \(X=0\) to \(X=7\) and then decreases monotonically. The upper part of Figure 4.1 shows two possible representations, B and G. In each there is a 1-1 correspondence between semantic entities and strings, and the function representing \(U^*\) is the "natural" one, i.e.,

\[
(U_B S) = (U^* (B^{-1} S)) \quad \text{and} \quad (U_G S) = (U^* (G^{-1} S)).
\]

In the lower part of the figure the broken line is the graph of \(U_B\) and the solid line is the graph of \(U_G\).
<table>
<thead>
<tr>
<th>Binary Representation (B X)</th>
<th>Grey Code Representation (G X)</th>
<th>semantic candidate X</th>
<th>utility functions (U* X) = (U_b (B X)) = (U_g (G X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0010</td>
<td>0011</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0011</td>
<td>0010</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>0110</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0101</td>
<td>0111</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0110</td>
<td>0101</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0111</td>
<td>0100</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>1100</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1001</td>
<td>1101</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>1010</td>
<td>1111</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>1011</td>
<td>1110</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>1100</td>
<td>1010</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>1101</td>
<td>1011</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>1110</td>
<td>1001</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1000</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.1.
Representation B results in the task being performed much more efficiently than representation G. In representation B the structure of the syntactic search space and systematicity in the behaviour of the system (its hill-climbing propensity) are very well suited to the task of finding S such that \((U_b S)\) exceeds a given threshold. In choosing representation B one is implicitly providing the task-specific knowledge that the \(U^*\) is well-behaved in a particular way. Using the techniques described in previous sections an improved version of G, \(G'\), can be constructed. In \(G'\) semantic entities with the same utility are corepresented. Several entities have alternative representations, e.g., \(0011\) and \(0100\) are alternative representations of 7. Semantic entity 15 is unrepresented, indicating (implicitly) to the system that it need never be considered. Figure 4.2 shows \(G'\) and the graph of \(U_{G'}\).

Implicit provision of task-specific knowledge involves the construction of a representation well-suited to the particular task of interest. The deliberate construction of task-specific representations is most evident in the learning and problem-solving systems of the late 1960's\(^{13}\). For example, [Amarel 1968] is a thorough study of how the size of a problem-solving search space can be drastically altered by changing the way in which individual states are represented. Equating reduction in the size of the search space with improvement in problem-solving efficiency, [Amarel 1968] observes (p. 168):

> the main improvements in problem-solving power came from the discovery and exploitation of useful properties in the search space\(^{14}\)

The exploitation of these "useful properties" did not involve explicitly describing them to the system. As in the preceding example, properties in the semantic domain were exploited by choosing a representation that put these properties into correspondence with aspects of the system, such as the structure of the syntactic search space and systematicity in the system's behaviour.

\(^{13}\) e.g. [Koffman 1968], [Murray and Elcock 1968]. Current advocates of task-specific representation are [Lenat and Brown 1984] and [Laird et al. 1986] (p. 31).
<table>
<thead>
<tr>
<th>Modified Grey Code Representation (G' X)</th>
<th>semantic candidate X</th>
<th>utility function (U* X) = (U_{G'}, (G' X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0, 14</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1, 13</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0010</td>
<td>3, 11</td>
<td>4</td>
</tr>
<tr>
<td>0110</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>0111</td>
<td>5, 9</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>6, 8</td>
<td>7</td>
</tr>
<tr>
<td>0100</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1100</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>1101</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>1111</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>1110</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>1010</td>
<td>2, 12</td>
<td>3</td>
</tr>
<tr>
<td>1011</td>
<td>4, 10</td>
<td>5</td>
</tr>
<tr>
<td>1001</td>
<td>5, 9</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>5, 9</td>
<td>6</td>
</tr>
</tbody>
</table>

![Graph](Figure 4.2)
CHAPTER 5

REPRESENTING THE MODEL OF THE LEARNING PROCESS

The abstract model of the learning process is shown in Figure 5.1. A system that represented this model literally would consist of two distinct subsystems, corresponding to the performance history update process and the hypothesis selection process, and three distinct bodies of information, the current performance history, the current hypothesis, and the fixed candidate set. The hypothesis selection subsystem would consist of an enumerator of the candidate set and an evaluator of the selection criterion.

![Diagram of the learning process model](image)

*Figure 5.1. Abstract model of the learning process (from Chapter 2).*
In all but the most trivial learning tasks a literal representation of this model is not a practical system. The main practical problems in representing the model are the following.

(1) The candidate set is usually very large, sometimes infinite. This causes two problems. The first is that the hypothesis selection process cannot examine every candidate individually. The second is that the candidate set cannot be represented as a body of information, in the sense of a static object. It must be represented by a process, for example, one that generates candidates at the request of the hypothesis selection process.

(2) The performance history grows steadily with time, eventually exceeding the available storage resources.

(3) The current hypothesis is generated by one process and used by a different process. A problem arises because representations of hypotheses that are well-suited to one of these processes are very often poorly suited to, or inadequate for, the other. In other words there are competing demands on the representation of hypotheses from the processes whose inputs or outputs are hypotheses. The problem is how to satisfy all the demands. A similar problem arises for the representation of performance information and performance histories, and, when the candidate set is represented as a process, for the representation of candidates.

These problems may be approached in several ways. For example, possible approaches to the problem of competing demands on the representation of performance histories are:

(a) there may exist a representation that satisfies all the demands, i.e., is well-suited to both performance history update and hypothesis selection. Finding such a representation may be very difficult.

(b) use two representations, one which is well-suited to performance history update and easily translated to the other, which is well-suited to hypothesis selection.

(c) corepresent the two processes.

(d) partially corepresent performance histories and the hypothesis selection process, i.e. have the declarative forms of hypothesis selection processes represent performance
histories. By doing this the performance history update process directly modifies the selection process. Reconfigurable enumerators (Chapter 7) take this approach.

Systems may be classified according to the approach taken to the practical problems of representing the abstract model. Existing systems fall into a relatively small number of classes, described in the sections that follow. Systems of the same type, i.e., that take the same approach to these problems, tend to have identical architectures. That is, they consist of the same processes and bodies of information interacting in the same way. The grouping of systems produced by this method of classification is very similar to some of the classification schemes in the literature (e.g. [Carbonell et al. 1983]). In some cases the boundary between two classes is somewhat fuzzy. This and other relationships between the classes are discussed fully in the next chapter.

5.1 Enumerative Systems

In the general architecture for enumerative systems there are three components, a candidate enumerator, a tester, and a selector. The candidate enumerator is a procedure representing the set of candidates. It produces candidates one at a time; the order in which they are produced is called the enumeration sequence. The tester and selector represent different parts of the selection criterion. Recall that a selection criterion is a relation, $(\text{SC} \langle \text{HISTORY}_k, \text{CANDIDATE-SET}, \text{CANDIDATE} \rangle)$, between a performance history, the candidate set, and an individual candidate. In an enumerative system the tester represents the part of the selection criterion that depends only on the performance history and the candidate; the selector represents the part that involves the candidate set. Either part may be empty. For example, the selection criterion of an incremental concept learning task is

$$\text{FOR ALL } \langle E, X \rangle \in \text{HISTORY}_k: (\text{CANDIDATE } E) = X$$

This would be represented entirely by the tester. The opposite is true of a clustering task. For example, the selection criterion in Section 2.2,
FOR ALL C ∈ CANDIDATE-SET:

(COMPLEXITY CANDIDATE) + (POORNESS <C, HISTORY, k>) ≤ (COMPLEXITY C) + (POORNESS <C, HISTORY, k>),

would be represented entirely by the selector. The tester thus filters individual candidates; those that pass the tester are accumulated by the selector until a candidate satisfying both parts of the selection criterion is found.

The problem of avoiding examining all candidates is solved, in part, by choosing an enumeration sequence that guarantees the selector need accumulate only a finite number of candidates. For example, some systems select the first candidate passed by the tester (in these systems there is no selector component). [Wharton 1973] strongly advocates this type of enumerative system as a good way of finding the "best" candidate compatible with the classified examples: enumerate candidates in order, best to worst, and select the first that is compatible with the classified examples (pp. 10-13). Other systems select from amongst the candidates preceding the first candidate failed by the tester. In either case the choice of candidate is a function of the enumeration sequence, and so the hypothesis selection process is jointly represented by the tester, selector, and the candidate enumerator.

In simple enumerative systems, the performance history is merely a record of the performance information that has occurred. The tester and selector access this record in assessing candidates. In most systems, the candidate enumerator uses the performance history to determine the point in the enumeration sequence from which to resume; usually the current hypothesis serves as the resumption point. The problem of the growing size of the performance history is not addressed by these simple systems. In reconfigurable enumerative systems, examined in Chapter 7, the performance history and candidate enumerator are partially corepresented, and the performance history is updated by reconfiguring the enumerator.

Enumerative systems are most common in the study of language identification. The grammar enumerators of [Wharton 1977] are described in Chapter 7; other grammar
enumerators are described in [Dietterich et al. 1982] (pp. 503-505). Systems described as “model-driven” or “schema-driven” are invariably enumerative. RULEGEN and SPARC are the best known of these; another is [Waterman 1975].

5.2 Candidate Set Modification Systems

In a candidate set modification system, a performance history is represented as a subset of the candidate set called the active set. Hypothesis selection is based solely on the current active set. Performance history update selects a different subset to be active. To facilitate this operation, candidate set modification systems usually have a language for representing subsets of the candidate set. Thus a two-pronged approach is taken towards the practical problems arising from the sheer size of the candidate set: first, the system's processing is restricted to a subset of the candidate set; second, the subsets have concise descriptions in a representation language well-suited to both update and hypothesis selection. In these systems it is the active set that is modified: the candidate set is an unchanging property of the system.

A candidate set modification system is monotone decreasing (increasing) if each active set is a subset (superset) of the active set preceding it. For example, the Candidate Elimination system is monotone decreasing. The Edinburgh Focussing system is nonmonotonic: the active set is normally a subset of its predecessor but under some conditions the system backtracks and chooses an active set incomparable to its predecessor.

The hypothesis selection process is represented in a candidate set modification system in one of two ways. In some systems, it is jointly represented by the performance history update process that selects the active set of candidates and a subsequent process that selects a candidate within the active set. For example, in the Edinburgh Focussing system the active set is represented by a pair of concepts, the most general and the most specific concept compatible with the seen examples. When new examples become available a new
hypothesis is selected in two steps: the active set is updated and then some candidate in
the active set, typically the most specific concept, is selected. The current hypothesis is
irrelevant to the selection of the next hypothesis. In this type of concept modification
system the candidate set is the union of all possible active sets.

In other systems the representation of an active set is interpreted in two different ways, as a
set of candidates for the purposes of performance history update, and as an individual
candidate for the purposes of testing. That is, hypotheses and active sets of candidates are
corepresented. Also corepresented are the processes of performance history update and
hypothesis selection.

For example, in the Candidate Elimination system a version space is the set of concepts
compatible with the seen examples. A version space is represented by a pair of sets of
concepts, \(<S,G>\), where \(S\) is the set of maximally specific concepts compatible with the
seen examples and \(G\) is the set of maximally general concepts compatible with the seen
examples. The output of the Candidate Elimination system is the current version space, i.e.
the current values of \(S\) and \(G\), and not a particular concept in this version space. In the
problem-solver in LEX there is a version space associated with each problem-solving
operator \(OP\) and a concept is interpreted as a set of problem-states. LEX's search is
guided by the degree of match between the current problem-state \(X\) and the version space
for \(OP\), where the degree of match between \(X\) and \(<S,G>\) is the proportion of concepts in
\(S \cup G\) that contain \(X\) (p. 171, [Mitchell et al. 1983]). LEX's interpretation of \(<S,G>\) as a
concept need not correspond to any of the concepts in the version space that \(<S,G>\)
represents. For example, suppose \(S=\{S_1,\ldots,S_n\}\) and \(G=\{G_1,\ldots,G_n\}\) such that \(S_i \subseteq G_i\) if and
only if \(i=j\). The problem-states in \(P_i = S_1 \cap S_2 \cap \cdots S_{i-1} \cap \neg G_i \cap S_{i+1} \cap \cdots \cap S_n\) (\(\neg G_i\) is the complement
of \(G_i\)) match \(<S,G>\) with high degree, \((n-1)/n\), but no concept in the version space \(<S,G>\)
can contain all the problem-states in all the \(P_i\).

In this type of system each active set is a hypothesis; the set of candidates for such a
system is therefore the set of possible active sets. The elements of active sets - e.g. the
concepts in a version space - have a peculiar status. They are commonly called "candidates": this is appropriate only in systems where an active set - e.g. a version space containing a single concept - exists that corresponds to each element. However, it must be emphasized that in most concept space modification systems of this type the individual elements of active sets are not the only candidates.

5.3 Hypothesis Modification Systems

A hypothesis modification system produces a new hypothesis by modifying the current hypothesis. The system has a set of modification operators mapping one candidate to another and proceeds by applying different sequences of modification operators to the current hypothesis until a candidate satisfying the selection criterion is found. Performance information is used to assess the candidates generated. It may also be used either to determine the sequences of modification operators to apply or as an argument to a modification operator. The hypothesis serves as the performance history, occasionally supplemented with other information. The representation of hypotheses is tailored to the hypothesis application process. The set of modification operators, together with an initial hypothesis defines the set of candidates.

Most learning systems in AI are hypothesis modification systems, for example, Waterman's poker learning system[Waterman 1970], Hedrick's natural language learning system[Hedrick 1976], Shapiro's model inference system[Shapiro 1981], and all "hill-climbing" systems[Langley et al. 1987].

A merging system is a hypothesis modification system in which there is a single modification operator that produces a new hypothesis by "merging" the performance information and the current hypothesis. A simple merging system is a Perceptron: misclassified examples are literally added to the current hypothesis. Systems dealing with conjunctive concepts are very often merging systems. Suppose hypotheses are n-tuples,

\footnote{Negative examples are negated before being added (p.167, [Minsky and Papert 1969]).}
<h_i,...,h_n>, where h_i is drawn from a set D_i whose elements are partially ordered such that every pair of elements (X,Y) has a unique least upper bound, X\ Y. An example is an n-tuple <e_1,...,e_n> in which each e_i is a minimal element of D_i. Systems that propose the hypothesis <h_i,e_i,...,h_n,e_n> given example <e_1,...,e_n> and current hypothesis <h_i,...,h_n> are merging systems. "Algorithm 1" for conjunctive concepts in [Haussler 1987] is like this. [Kodratoff and Ganascia 1986] and [Vrain 1986] extend this type of merging to objects and hypotheses expressed as conjunctive first-order formulae by preceding the merging step with a "structure matching" step in which terms in the example are put into correspondence with terms in the hypothesis.

5.4 Performance History Modification Systems

In a performance history modification system performance histories and hypotheses are partially corepresented and so are the processes of performance history update and hypothesis selection. The representation used for performance histories/hypotheses is tailored to the process of update rather than hypothesis application and is typically closely related to the representation of performance information. Performance history update is invariably a simple process. In some systems it involves little more than recording the new information. In others it involves integrating the new information with the current performance history and/or revising indexing schemes associated with the performance history. The set of candidates is the set of possible performance histories. The (semantic) candidates represented by performance histories are almost always "fuzzy" concepts, i.e. concepts for which membership is a matter of degree. Typically, performance history update systems learn from positive examples only, or learn one concept for each distinct class and classify an example by comparing the degree of its membership in all the classes.

The set of performance history systems can be divided in two on the basis of the nature of the process, called the matching process, for determining the degree of membership of an example in a concept. In an analogy system a performance history is a prototype or set
of prototypes and the matching process involves transforming a given example to minimize the differences between the example and the performance history. The degree of match is a function of the differences remaining after transformation and the transformation sequence used. Analogy systems include [Carbonell 1986, Winston 1980, Kolodner et al. 1985] and [Porter and Bareiss 1986].

In a **constructive system** a performance history is a collection \( \{g_1, \ldots, g_M\} \) of functions from examples to a subset \( R \) of the reals (often \( \{0, 1\} \)). In some systems this collection is a set, in others a multiset. Matching involves applying each \( g_i \) to a given example, \( X \). In contrast to the complex transformation process in analogical matching, constructive systems restrict themselves to \( g_i \) that are efficient to apply. Degree of match is a function of the values \( g_i(X) \), such as the maximum or average or the number that are nonzero. Given example \( e \) in class \( C \), the collection \( H_c = \{g_1, \ldots, g_M\} \) currently associated with \( C \) is updated by adding to \( H_c \) a selection of the functions \( \{G_1 e, \ldots, G_n e\} \), where \( \{G_1, \ldots, G_n\} \) is a fixed set of curried similarity functions, \( G_i : \text{EXAMPLES} \rightarrow \text{EXAMPLES} \rightarrow R \).

Both update and hypothesis application are simple and efficient in constructive systems: [Laird et al. 1986] calls these systems "simple experience learning systems" (p. 13). The preceding abstract description obscures the simplicity of individual constructive systems. In particular, their representations of hypotheses/performance histories are very much simpler than suggested by the description "a collection \( \{g_1, \ldots, g_M\} \) of functions from examples to a subset \( R \) of the reals". For example, in the exemplar systems of [Kibler and Aha 1987] there is only one similarity function, \( G \). In the actual system, update is nothing more than selectively recording the seen examples, and hypothesis application nothing more than comparing the given example with each of the selected seen examples by applying \( G \).

An example of a constructive system having multiple similarity functions is the adaptive image recognition system WISARD [Aleksander and Stonham 1979]. In WISARD each binary image (example) is decomposed into \( K \) \( n \)-tuples. Notionally there are \( K \) projection functions \( p_1, \ldots, p_K \) from images to \( n \)-tuples. A hypothesis consists of one degree-of-
membership function for each class. The degree-of-membership function for class C is represented as a set of K random-access-memories each having n address bits. Let \( \text{RAM}_C(i,N) \) denote the location addressed by binary n-tuple N in the \( i^{th} \) random-access-memory associated with class C. Initially all locations are set to zero. WISARD's update procedure, given example \( e \) of class C, is to set \( \text{RAM}_C(i,(p_i,e)) \) to one, for all \( i \in \{1,\ldots,K\} \). The degree of membership of an example, \( X \), in class C is the sum over \( i \in \{1,\ldots,K\} \) of \( \text{RAM}_C(i,(p_i,X)) \).

In terms of the abstract description given above, WISARD employs K similarity functions, \((G_i(x),y)=1\) if \((p_i,x)=(p_i,y)\) and 0 otherwise. The degree-of-membership function for class C is represented by the set (not multiset) \( H_C=\{(G_i,e)|e \in E, 1 \leq i \leq K\} \), where E is the set of seen examples of C. The degree of membership of example \( X \) in class C is the number of functions \( h \in H_C \) such that \((h,X)=1\); in other words the sum over \( h \in H_C \) of \((h,X)\). This calculation of degree of membership of \( X \) is equivalent to the one previously described. Both count the number of \( G_i \) for which there exists \( e \in E \) such that \((G_i,e)=1\). [Andreea 1977] is a similar constructive system.

5.5 Mediating Systems

Merging systems and performance history modification systems employ the single representation trick, i.e., they use the same representation language for performance information and hypotheses\(^2\). For example, in concept learning systems the language for representing examples is very often a restriction of that for representing concepts. An example is usually corepresented with the concept containing just that example, but in some cases it is corepresented with a concept containing many other examples\(^3\). In tasks where the natural representations of performance information and hypotheses are distinct, merging

\(^2\) pp. 368-9, 424-5, [Dietterich et al. 1982]

\(^3\) in systems that learn Boolean threshold concepts (\( \langle P_1,\ldots,P_n \rangle \)) represents both the example \( P_1 \& P_2 \& \ldots \& P_i \& P_{i+1} \) and the smallest Boolean threshold concept containing this example. This concept contains all examples in which all of \( P_1,\ldots,P_i \) are true. There are \( 2^{n+1} \) such examples.
or performance history modification systems may be used by first translating the performance information into the hypothesis representation language.4

A mediating system employs two representation languages. Each hypothesis is represented by an entity well-suited to hypothesis application. Each performance history is represented by an entity, called a mediating structure, that is well-suited to update5. Often the mediating structure is closely related to the representation of performance information, but occasionally it is not [Fiann and Dietterich 1986]. The performance history update process is followed by a straightforward process that translates the current performance history into a hypothesis. Hypothesis selection is jointly represented by these two processes. The current hypothesis plays little or no part in the selection of the next hypothesis. The set of candidates is the image under the translation process of the set of possible performance histories.

A simple mediating system is BOXES. The performance task is to control a device whose state changes with time. The controller can sense the state of the device and issue actions that affect the change of state. The objective of the controller is to prevent the device from entering a specific region of state-space; states in this region are called "terminal" states. A hypothesis is a controller for the device. To represent hypotheses, the state-space of the device is partitioned into a number of regions, or "boxes". A hypothesis is a labelling of each region with the name of one of the actions. A hypothesis is applied to the performance task by the issuing the action that labels the box corresponding to the current-state of the device. Performance information indicates

--- the time elapsed between starting and entering a terminal state, and

-- for each region, the times during the run at which the device entered that region.

The performance history accumulates two statistics for each <action,region> pair: the number of times the action has been taken in the region, and the mean time until termination after taking the action in the region. Translation of this information into a

4 METADENDRAL uses this approach

5 the class of mediating systems was first identified in [Rendell 1985]. The learning process is mediated by the representation of performance histories in the sense that this representation "stands between" the representation of
hypothesis is done region by region; the action used for a region is the one whose statistics in that region score highest according to a simple formula. A similar system is the adaptive robot controller in [Raibert 1976].

In PLS a hypothesis is a strategy, represented as a set of coefficients of an evaluation function, for finding solutions to instances of the 15-puzzle. Performance information consists of an instance of the 15-puzzle and a list of puzzle-states classified as "on" or "not on" the solution path. Instead of reasoning directly about coefficients of evaluation functions, PLS manipulates a mediating structure called a "cumulative region set". This is a partition of the puzzle state space into regions such that all the points in a particular region are equally likely to be on a solution path. The derivation of evaluation function coefficients from a cumulative region set is a straightforward matter. [Rendell 1985] summarizes the practical advantages of using a mediating structure (pp. 66-67):

Assignment of credit to individual regions within a cumulative set \( R \) is straightforward, but it would be difficult to do directly in the final evaluation function \( H \), since the components of \( H \), while appropriate for performance, omit information relevant to learning [Rendell's footnote: There are various possibilities for the evaluation function \( H \), but all contain less useful information than their determinantal, the region set \( R \) ...]. ... Retention and continual improvement of this mediating structure relieves the credit assignment problem. This view is unlike that of ... [mainstream machine learning]; learning systems often attempt to improve the control structure [hypothesis] itself, whereas PLS acquires knowledge efficiently in an appropriate structure, and utilizes this knowledge by compressing it only temporarily for performance. In other words, PLS does not directly search rule space for a good \( H \), but rather searches for good cumulative regions from which \( H \) is constructed.

---

performance information and that of hypotheses.
EXAMINING THE RELATIONSHIPS BETWEEN SYSTEMS

The relationships that will be examined in this chapter correspond to the arcs in Figure 6.1. An arc from one class to another indicates there are learning systems in the first class whose behaviour is very closely related to that of a system in the second class. Labels on the arcs indicate the number of the section in which the corresponding relationship is discussed. Relationships between enumerative systems and the others are examined in the next chapter.

![Diagram](Image)

*Figure 6.1.*
6.1 Relationships Based on Transitional Systems

Learning systems are usually described in terms intimately related to the language used by
the system to represent hypotheses, performance information and history and so on. For
example, ID3 is described in terms of operations on trees (pruning, refining a node), PLS in
terms of operations on region sets (splitting regions, revising penetrance values and error
estimates), and RULEGEN in terms of operations on sets of rules describing chemical "bond
environments" (increasing the radius of a bond environment, specifying the value of an
attribute of an atom). This practice can be a considerable impediment to the comparison of
systems, even when they are in the same architectural class.

This obstacle can sometimes be overcome by redescribing both systems in the same
language. An entirely accurate description of ID3, for example, can be given in terms of
operations on a set of production rules. A decision tree can be interpreted as a set of rules
by interpreting each path in it (from root to leaf) as a separate production rule whose
conclusion is the classification in the leaf and whose premises are the attributes/values that
define the path. In terms of production rules, ID3's central learning operation is to replace a
single rule with which no classification is currently associated, \( P \Rightarrow ? \), with a set of rules,

\[
\{ P \&(A_i=1) \Rightarrow ?, P \&(A_i=2) \Rightarrow ?, \ldots, P \&(A_i=V_i) \Rightarrow ? \}
\]

where \( A_i \) is an attribute with values 1,2,...,\( V_i \). Each of these rules is either further refined or
assigned a specific conclusion (classification); different refinements of a rule may or may not
be assigned different conclusions. Described in this way, ID3 can readily be compared to
systems that represent hypotheses as sets of production rules such as SAGE[Langley
1982], [Waterman 1970], and RULEGEN.

A classification system can be interpreted as a partition of the set of possible examples in
which there is a classification associated with each set in the partition. This interpretation
bypasses the language used to represent the classification system, and can therefore serve
as a common basis for the description of learning systems\(^1\). ID3, a hypothesis modification system, PLS, a mediating system, and Edinburgh Focussing, a candidate set modification system, can all be described in terms of partitions of instance space.

It can happen that a system, described in one way, is a member of one class of system and, when described in another way, is a member of a different class. Such a system is **transitional** between the two classes.

### 6.1.1 Transitional Merging/Performance History Update Systems

In both merging systems and performance history update systems the processes of hypothesis selection and performance history update are corepresented as a single, efficient operation. The update operation and the representation of the performance history in some performance history update systems\(^2\) are indistinguishable from the update operation and representation of hypotheses in merging systems. Similarly, merging systems in which a hypothesis is an organized collection of concepts, such as [Langley et al. 1987], are indistinguishable from performance history update systems that maintain an indexed collection of prototypes and specific exemplars, such as [Kolodner et al. 1985]. Constructive systems are particularly similar to merging systems, because both types of system employ a representation that is well-suited to both the learning and performance tasks.

---

\(^1\) [Holte 1985], [Gilmore 1986], and [Thomton 1987] describe several learning systems in this way.

\(^2\) e.g. those that maintain a single prototype such as [Winston 1975]
6.1.2 Transitional Candidate Set/Hypothesis Modification Systems

There are three types of system transitional between candidate set modification and hypothesis modification. First, all candidate set modification systems that corepresent the active set and hypotheses, such as Candidate Elimination, are transitional: the update process can be regarded either as active set modification or as hypothesis modification.

Secondly, in many hypothesis modification systems a new hypothesis is selected from a small, well-defined subset of the candidate set. This active set is defined by the current hypothesis and a strategy restricting the sequences of modification operators that will be considered. A good example of such a system is described in [Solomonoff 1975] (the system will be referred to as SOL). In the learning task for which SOL is designed it is possible to prove that the best new hypothesis lies within a fixed "distance" of the current hypothesis, where distance is the length of the sequence of modification operators in SOL. The candidates within this distance are SOL's active set. Choosing a new hypothesis simultaneously chooses a new active set.

Finally, there are hypothesis modification systems that have, in addition to the usual processes, a process that modifies the language in which hypotheses are represented. For example, the system in [Schlimmer 1987] includes processes that define new terms that are used by a perceptron-like hypothesis modification process. Such a system can be described as a candidate set modification system. The candidates that are representable with the current set of terms constitute an active set which is updated by the processes that define new terms, and the hypothesis modification process constitutes a hypothesis selection algorithm.
6.1.3 Transitional Candidate Set Modification/Mediating Systems

Any candidate set modification system having an explicit hypothesis selection step is also a mediating system. The active set is the mediating structure, and the hypothesis selection process is the translation process. Of particular interest are systems in which part of the active set selection process is shifted into the hypothesis selection process and merged with subprocesses there. This is illustrated in Figure 6.2. Such systems can employ a representation well-suited to the unmerged part of the active set modification process.

![Figure 6.2](image)

**Figure 6.2.** A system that is transitional between candidate set modification (A) and mediating (C). Part of the active set selection process (indicated with dots/colons) is (B) shifted into and (C) merged with the hypothesis selection process.

The "one sided version space" algorithm [Haussler 1987] is a system of this type. It is an adaptation of Candidate Elimination for learning tasks in which the $G$ boundary set grows impractically large. It performs ordinary $S$-update for each positive example but merely records the negative examples. $G$-update is merged with the hypothesis selection process; a hypothesis is chosen by searching (heuristically) from the maximally general concept(s) towards the concepts in $S$ for the first candidate that covers no negative examples.
6.1.4 Transitional Mediating/Merging/Constructive Systems

The distinctive feature of mediating systems is their two process, two representation language nature: first update the mediating structure, then translate it into a hypothesis. But the second process could be shifted into the performance system and merged with processes there. This simple shift results in a single process, single representation system, whose representation is well-suited to being updated. A mediating system has been transformed into a merging or constructive system.

The BOXES system illustrates the process of shifting and merging. Its mediating structure, the performance statistics for each <action,region> pair, is translated into a hypothesis region by region, by using a simple formula to select the action to be taken in a given region. The action selected for one region does not depend on the action selected for any other region. Shifting the translation process into the performance system transforms BOXES from a mediating system to a transitional merging/constructive system whose task is to maintain performance statistics. The translation process can be merged with the classification process by applying the translation formula only to those regions that are actually entered during a run. Merging the two processes in this way results in considerable improvements in efficiency because the number of regions entered in a typical run is a small fraction of the total number of regions.
6.2 Relations Based on Behaviour-Preserving Transformations

A behaviour-preserving transformation maps one system to another that has equivalent input-output behaviour. The shifting and merging of processes illustrated in the previous section are simple types of behaviour-preserving transformation. Yet they are sufficient to transform a system that is transitional between two classes, like BOXES, into a distinct member of either of the two classes. This section and the next chapter employ transformations of a more complex and sometimes task-specific nature.

Behaviour-preserving transformations have a number of important uses in an analytical framework for learning systems.

(1) It is possible to transform a system in one class into a system in almost any other class, and doing so reveals important relationships between the two classes. The relationship between analogy systems and the other classes of system is established by this method in this section.

(2) In identifying the task-specific transformations needed to transform one system into another, one is identifying differences in the task-specific knowledge that is provided implicitly to the two systems. This is illustrated in both examples below.

(3) Behaviour-preserving transformations may have dramatic effects on the overall efficiency of a system and on the distribution of computational effort within a system.

(4) A sequence of transformations produces a sequence of behaviourally equivalent systems. But each system in the sequence can differ from the others in class, task-specific knowledge, and efficiency. This suggests that a transformation-based approach to the design of learning systems might be profitable.

The discussion of transformations in this chapter will concentrate on the first two of these uses; the efficiency effects of the transformations will be pointed out but not discussed. The next chapter is a full analysis of efficiency.

Two demonstrations of the transformation of an analogy system into a system in other
classes follow\(^3\). Both satisfy the following general description. The performance task is a classification task. Input to the overall system consists of a sequence of pairs, each pair consisting of an example and either the name of a class or the special symbol "?". Paired with a "?", an example is called "unlabelled"; paired with a class, it is called "labelled". Pairs may recur in the sequence, but it is assumed that the class of an example is unique. When given an unlabelled example the system outputs a classification, i.e. a set of classes (possibly empty).

6.2.1 Transforming an Analogy System: A Simple Example

In this section, examples are integers. A typical input sequence to the system is

\[<45, A>, <13, A>, <25, B>, <17, ?>, <18, B>, <4, A>, <36, B>, <17, ?>\]

The behaviour of the system is specified as follows. The output required in response to an occurrence of \(<X, ?>\) is the class of the labelled integer \(Y\) preceding this occurrence of \(<X, ?>\) for which the unsigned difference, \(|X-Y|\), is minimum. If there are two such labelled integers the system should output the class of the larger. For example, upon the first occurrence of \(<17, ?>\) in the above sequence, the required output is "A"; upon the second occurrence, the required output is "B".

An analogy system satisfying this specification could represent a hypothesis with an unordered list of the labelled integers that have occurred thus far in the input sequence. Updating a hypothesis represented in this way involves inserting new labelled integers at any convenient position in the list (it is not even necessary to check for duplicates). Using an unordered list in a way that satisfies the behavioural specification involves searching through the entire list. This representation of hypotheses is poorly-suited to the performance task. A representation very well suited to the performance task is a binary decision tree. For example, the hypothesis produced given the input sequence above could

\(^3\) an early version of these demonstrations is published in [Holte 1986].
be represented by

```
  X < 27 ?
   yes
   no
   X < 16 ?
       yes
       no
       X < 41 ?
           yes
           no
           no
           A
           B
           B
           A
```

One transformation of the analogy system into a system that manipulates this type of decision tree will now be examined.

The first step is to transform the representation of hypotheses from an unordered list to a list sorted in increasing order. The learning process may be modified in one of two ways. First, it could be modified to update the hypothesis directly, by inserting new labelled integers in the appropriate position in the sorted list. Modified in this way the system remains a performance history update system. Alternatively, the learning system could be modified by adding a sorting step to translate the performance history, represented and updated as before, into a sorted list. This modification transforms the learning system into a mediating system.

A sorted list is much better suited to the performance task than an unsorted list. Efficient search techniques, such as binary search, can be used to find the integer(s) in the list closest to a given integer. For example, to classify the integer X given the list

```
<4,A>, <13,A>, <18,B>, <25,B>, <36,B>, <45,A>
```

one would first ask, "Is X closer to 18 than to 25?", where 18 and 25 are the two integers in the middle of the list. If X is closer to 18, the search would continue recursively using the first half of the list; if X is closer to 25 it would continue in the second half of the list.
Type 1 Decision Trees

The set of questions used in a binary search of a particular sorted list can be compiled into a balanced binary decision tree. The question "is X closer to 18 than to 25?" can be expressed as "is $|X-18| < |X-25|$?". The decision tree corresponding to the above list is

```
|X-18| < |X-25| ?
  yes /   no
  |X-13| < |X-18| ?   |X-36| < |X-45| ?
  yes /   no   yes /   no
  |X-4| < |X-13| ?   |X-25| < |X-36| ?
  yes /   no   yes /   no
      A       B    A        B
```

The transformation of the list of labelled examples into this type of decision tree is task specific. Its validity depends on the type of "questions" that will be asked during the classification process. A learning system could directly update such a decision tree by inserting a new labelled example under the appropriate node and rebalancing the tree.

Type 2 Decision Trees

The tests in the decision tree can be transformed using the following property

if $Y < Z$, then $|X-Y| < |X-Z|$ if and only if $X < (Y+Z)/2$

Applied to the above decision tree, this transformation produces
Is it possible for a learning system to update this type of decision tree directly and in the manner defined by the behavioural specification of the performance system? The answer would clearly be "yes" if it were possible to transform this type of decision tree into the previous type of decision tree, for which a learning algorithm was known. If there are N labelled integers, $Z_1,...,Z_N$, in the original sorted list, both types of trees will contain the N-1 tests asking whether $X$ is closer to $Z_i$ or $Z_{i+1}$ (1 ≤ i ≤ N-1). In the second type of tree these tests are of the form "$X < (Z_i + Z_{i+1})/2$?" therefore the value of $(Z_i + Z_{i+1})$ is available in this type of tree. These N-1 relations do not completely determine the exact values of the original labelled integers, even taking into account the additional constraint that $Z_i < Z_{i+1}$ (for all i). Exactly the same tree would be produced by the labelled integers:

$<7,A>, <10,A>, <21,B>, <22,B>, <39,B>, <42,A>$

However, the exact values of the labelled integers are crucial for the learning task: if $<19,B>$ is added to these two sets of labelled integers different trees must be produced. When $<19,B>$ is added to the original set of labelled integers, $<15,?>$ should produce the label "A", whereas when it is added to this second set of labelled integers, $<15,?>$ should produce the label "B".

Thus, this type of decision tree is inadequate, on its own, for the learning task; it does not contain enough information to perform the learning task correctly. If the exact value of the smallest labelled integer is stored along with this type of decision tree, there would be
sufficient information for the learning task. The fractional part of the numbers in the tests supplies information that is necessary for the learning task, although unnecessary for the performance task\(^4\). The smallest labelled integer and fractional parts of the numbers in the test are typical of the small amount of additional performance history information kept by Hypothesis Modification systems.

**Type 3 Decision Trees**

The final transformation involves compressing the tree, for example by replacing all subtrees in which all leaves have the same label by a single leaf with that label. Applying this transformation to the above tree produces:

\[
\begin{align*}
X < 21.5 & \quad ? \\
& \quad \text{yes} \quad \text{no} \\
& \quad \begin{align*}
X < 15.5 & \quad ? \\
& \quad \text{yes} \quad \text{no} \\
& \quad \begin{align*}
A & \\
& \\
B & \\
\end{align*} \\
& \quad \begin{align*}
X < 40.5 & \quad ? \\
& \quad \text{yes} \quad \text{no} \\
& \quad \begin{align*}
B & \\
& \\
A & \\
\end{align*} \\
\end{align*}
\end{align*}
\]

This type of tree produces the same classifications as the previous types. But it contains even less of the information required for the learning task than the previous type of decision tree: every test in the previous type of tree contributed information necessary for the learning task.

\[^4\text{if } X \text{ and } Y \text{ are integers then } X < Y + 0.5 \text{ if and only if } X < Y + 1. \text{ Therefore all the non-integers in the tests can be replaced by the nearest larger integer without affecting the classification assigned to any integer.}\]
6.2.2 Transforming An Analogy System: A Full-Scale Example

The previous example introduced the type of transformations that are useful for comparing systems, and illustrated the effects these transformations can have on the class, knowledge content, and efficiency of a system. In this section, similar transformations are applied to a "real" learning system with similar, although more dramatic, effects.

The system that will act as the target of the transformation process is the system for synthesizing finite state machines described in [Biermann and Feldman 1972]. This paper does not provide a detailed description of the learning system, but the description that is provided (especially sections IV and V) conveys the unmistakeable impression that the learning system reasons directly about finite state machines, identifying missing or extraneous states and arcs when a new labelled example is given, and making the appropriate modifications to the finite state machine. In other words, the learning system is a hypothesis modification system in which hypotheses are represented as nondeterministic finite state machines. This system was chosen because the paper includes a formal specification of its input-output behaviour. This has enabled the construction of an analogy system whose behaviour is equivalent to that of the original (hypothesis modification) system without reference to the specific algorithms and data structures of the original system.

Specification of the System

In the learning task described in [Biermann and Feldman 1972] examples are finite strings of characters drawn from a finite alphabet. The specification of the system's behaviour given an unlabelled string may be paraphrased as

the system should output the classes of all labelled strings that can be created by recursively rewriting successively larger initial segments of the given unlabelled string,

where the rewriting process involves replacing a string with an equivalent string.
The motivation for this specification is this. Suppose there is a set of strings, \(\{S_1, \ldots, S_k\}\), whose classes are known and that there is an equivalence relation, \(\equiv\), such that \(X \equiv Y \Rightarrow \text{CLASS}_X = \text{CLASS}_Y\). The class of a given string, \(X\), can be determined by finding a string equivalent to \(X\) whose class is known. Different definitions of equivalence require different procedures for finding a string, in a given set of strings, equivalent to \(X\). The notion of equivalence used in [Biermann and Feldman 1972] is derived from the Nerode equivalence relation:

\[ X \equiv_Y \iff \text{CLASS}_{XW} = \text{CLASS}_{YW} \text{ for all strings, } W \]

A property of Nerode equivalence is \(X \equiv Y \Rightarrow XW \equiv_Y YW\), for any \(W\). In other words, one can prove that \(XW \equiv_Y YW\) by proving that an initial segment, \(X\), of \(XW\) is equivalent to an initial segment, \(Y\), of \(YW\). The following illustrates how this property can be used to find an element of \(\{S_1, \ldots, S_k\}\) that is equivalent to a given string \(S\). Suppose \(S = x_1 \cdots x_N x_{N+1} \cdots x_M\), \(S_1 = x_1 \cdots x_N\), \(S_2 = y_1 \cdots y_P\), \(S_3 = y_1 \cdots y_P x_{N+1} \cdots x_M\), and \(S_1 \equiv S_2\). An initial segment of \(S\) is identical to \(S_1\), which is equivalent to \(S_2\). Substituting \(S_2\) for this initial segment of \(S\) produces a string equivalent to \(S\). This string is identical to \(S_3\); therefore \(S \equiv S_3\).

The notion of equivalence used in [Biermann and Feldman 1972] is a weakened version of Nerode equivalence, called \textit{k-tail equivalence}, defined in terms of a non-negative integer parameter, \(k\):

\[ X \equiv_k Y \text{ if and only if } \]

\[ \text{for all strings } W \text{ of length } k \text{ or less} \]

\[ \text{(LABEL } XW) = \text{(LABEL } YW) \]

\[ \text{where (LABEL } X) = \text{the class of } X \quad \text{if } X \text{ is a labelled string} \]

\[ = 0 \quad \text{otherwise}^5 \]

For example,

suppose alphabet = \{ a, b \} and classes = \{ 1, 2 \}
and the following labelled strings have been given:

\[
\begin{align*}
\text{string} & = a \quad aa \quad ab \quad aba \quad abb \quad abbb \\
\text{class} & = 1 \quad 2 \quad 1 \quad 2 \quad 1 \quad 2
\end{align*}
\]

For strings in this table the function (LABEL string) returns the corresponding class; for all

---

\(^5\) classes will be denoted by integers greater than zero
other strings it returns 0.

Given these labelled strings the k-tail equivalence relation is ("" denotes the empty string)

**equivalence classes under $\equiv_k$, $k=0$**

<table>
<thead>
<tr>
<th>Label X</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\equiv_0$</td>
<td></td>
</tr>
<tr>
<td>(LABEL X) = 1</td>
<td>{ a, ab, abb }</td>
</tr>
<tr>
<td>(LABEL X) = 2</td>
<td>{ aa, aba, abbb }</td>
</tr>
<tr>
<td>(LABEL X) = 0</td>
<td>{ all other strings }</td>
</tr>
</tbody>
</table>

**equivalence classes under $\equiv_k$, $k=1$**

<table>
<thead>
<tr>
<th>Label Xa</th>
<th>Label Xb</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(LABEL X) = 1 &amp; (LABEL Xa) = 0 &amp; (LABEL Xb) = 2</td>
<td>{ abb }</td>
<td></td>
</tr>
<tr>
<td>(LABEL X) = 1 &amp; (LABEL Xa) = 2 &amp; (LABEL Xb) = 1</td>
<td>{ a, ab }</td>
<td></td>
</tr>
<tr>
<td>(LABEL X) = 2 &amp; (LABEL Xa) = 0 &amp; (LABEL Xb) = 0</td>
<td>{ aa, aba, abbb }</td>
<td></td>
</tr>
<tr>
<td>(LABEL X) = 0 &amp; (LABEL Xa) = 1 &amp; (LABEL Xb) = 0</td>
<td>{ &quot;&quot; }</td>
<td></td>
</tr>
<tr>
<td>(LABEL X) = 0 &amp; (LABEL Xa) = 0 &amp; (LABEL Xb) = 0</td>
<td>{ all other strings }</td>
<td></td>
</tr>
</tbody>
</table>

The equivalence class for which (LABEL XW) = 0 for all W of length k or less is called the "all zeroes" equivalence class.

The notion of recursively rewriting successively larger initial segments of a string is captured formally in the definition of the function $T_k(Q, V)$ below. $T_k(Q, V)$ rewrites a string by replacing an initial segment of the string with a string that is equivalent ("$\equiv_k$") to the initial segment, and recursively rewriting the result until the entire original string has been consumed. The rewritten initial segment is the first argument to $T_k(Q, V)$ and the unprocessed remainder of the original string is the second argument.

$$T_k(Q, "") = \{ Q \}$$

$$T_k(Q, V_1 \ldots V_n) = \bigcup_{q \equiv_k Q} \bigcup_{l \in \{1, \ldots, M\}} T_k(qV_1 \ldots V_{l-1} V_{l+1} \ldots V_n)$$

In terms of this definition of $T_k(Q, V)$ the output required of the classification system given an unlabelled example, X, is

$$\text{(OUTPUT}_k X) = \{ (\text{LABEL } t) \mid t \text{ in } T_k("", X) \}$$

Using the equivalence classes shown above ($k=1$), the output required given the unlabelled string "ab" is the classes of the strings in $T_k("", ab)$. These strings can be calculated as
follows:

\[ T_k(\"\",ab) = T_k( a, b ) \]
\[ \cup T_k(ab, \"\") \]

\[ T_k(a,b) = T_k(ab, \"\") \]
\[ \cup T_k(abb, \"\") \]
\[ = \{ ab, abb \} \]

\[ T_k(ab,\"\") = \{ ab \} \]
\[ \cup T_k(abb,\"\") = \{ ab, abb \} \]

Thus, \( \text{OUTPUT}_k ab \) = \{ 1 \}

The output required given the unlabelled string "abb" is the classes of the strings in \( T_k(\"\",abb) \), calculated as follows:

\[ T_k(\"\",abb) = T_k( a, bb ) \]
\[ \cup T_k( ab, b ) \]
\[ \cup T_k(abb, \"\") \]

\[ T_k(a,bb) = T_k(ab, b) \]
\[ \cup T_k(abb, b) \]
\[ \cup T_k(abb, \"\") \]
\[ \cup T_k(abb,bb, \"\") \]

\[ T_k(ab,b) = T_k(abb, \"\") \]
\[ \cup T_k(abb,bb, \"\") \]
\[ = \{ ab, abb \} \]

\[ T_k(abb,b) = T_k(abb,bb, \"\") \]
\[ = \{ abb \} \]

\[ T_k(abb,\"\") = \{ abc \} \]

\[ T_k( a, bb ) = \{ ab, abb, abbb \} \]

\[ T_k(ab,b) = T_k(abb, \"\") \]
\[ \cup T_k( ab, \"\") \]
\[ = \{ ab, abb \} \]

\[ T_k(abb,\"\") = \{ abc \} \]

\[ T_k(\"\",abb) = \{ ab, abb, abbb \} \]

thus, \( \text{OUTPUT}_k abb \) = \{ 1, 2 \}
Following the pattern of the previous example, a sequence of behaviour-preserving transformations will be applied to an analogy system satisfying this specification. The prototypes in the present example are the labelled strings. An unlabelled string is matched against the prototypes and assigned the class of every prototype for which the matching process succeeds. The matching process, of course, is intimately related to the rewriting process, \( T_k(Q,V) \), described in the specification.

The target of the transformation process is the hypothesis modification system described in [Biermann and Feldman 1972]. A hypothesis in this system is represented by a nondeterministic finite state machine (NDFSM). There is one class associated with each state in the NDFSM. An unlabelled string is classified by using it as an input sequence to the NDFSM. Because of nondeterminism, each successive character in the string will drive the NDFSM from one set of states to another set of states. The output of the performance system is the set of classes associated with the states reached when the last character of the given string has been input to the NDFSM.

**The Analogy System**

The analogy system implements the performance specification in a more or less literal way. A hypothesis is represented as a set of strings (prototypes) each paired with its class, if known, or "0". String \( X \) is in this set if and only if there exists a string \( W \) of length \( k \) or less (possibly the empty string) such that \( \text{LABEL} \ X \ W \neq 0 \).

Given

\[
\begin{array}{ccccccc}
\text{string} & = & a & a & a & b & a & b & a & b & b \\
\text{class} & = & 1 & 2 & 1 & 2 & 1 & 2 & \\
\end{array}
\]

the hypothesis produced is \( (k=1) \)

\[
\{ \langle '' , 0 \rangle, \langle a, 1 \rangle, \langle aa, 2 \rangle, \langle ab, 1 \rangle, \langle aba, 2 \rangle, \langle abb, 1 \rangle, \langle abbb, 2 \rangle \}
\]

Updating a hypothesis represented in this way is very simple. For each new labelled string
<Z,CLASS₂> remove <Z,0> from the set⁶ and add <Z,CLASS₂> and <X,0> for each X not already in the set such that Z=WXW where W is length k or less. A set maintained in this way contains sufficient information to determine if X ≡ₖ Y holds for any pair of strings⁷.

This type of a list can be interpreted in a way that satisfies the behavioural specification with a literal implementation of the specification of (OUTPUTₖ X). Given string X, the set Tₖ("",X) is calculated in the manner demonstrated above and the class of every string in this set is output. There is one problem implementing (OUTPUTₖ X) literally. Tₖ(Q,X) is infinite when Q is in the "all zeroes" equivalence class. In the example (k=0) above, Tₖ("",Z) is infinite for all Z, because there are an infinite number of strings equivalent to "". However, because all but a finite number of strings in this class are indistinguishable for the purposes of both learning and classification⁸ a single representative (denoted OTHERS) can be used in place of the infinite number of indistinguishable strings.

Using this method, Tₖ("",ba) would be calculated as follows (for k=0):

\[
Tₖ("",ba) = Tₖ(b, a) \cup Tₖ(ba, "") \cup Tₖ(OTHERS, a) \cup Tₖ(OTHERS, "")
\]

Because the class of every string beginning with "b" is "0", the strings "b" and "ba" are replaced by the representative OTHERS, leaving

---

⁶ if labelled strings are presented in order of increasing length <Z,0> will never occur in the set  
⁷ If X is not in the set then X ≡ₖ Y holds if and only if Y is not in the set  
⁸ for all X in this class except a finite number, (LABEL XW)=0 for all W of any length. The exceptions are those strings, X, such that (LABEL XW)=0 for all W of length k or less but (LABEL XW)=0 for some W longer than k. In the example with k=0, "" is the only such string. There are no such strings in the example with k=1.
\( T_k(\"", ba) = T_k(\text{OTHERS}, a) \)
\[ \cup T_k(\text{OTHERS}, \"\") \]

\[ T_k(\text{OTHERS}, a) = T_k(\text{OTHERS}, \") \]
\[ \cup T_k(\ a, \") \]
\[ = \{ a, \text{OTHERS} \} \]

\[ T_k(\text{OTHERS}, \") = \{ \text{OTHERS} \} \]

Thus, \( T_k(\"", ba) = \{ a, \text{OTHERS} \} \), and \( \text{OUTPUT}_k( ba ) = \{ 1, 0 \} \).

When \( V = V_1 \ldots V_m \) is nonempty the calculation of \( T_k(Q, V) \) consists of two steps corresponding to the two union operations in the definition of \( T_k(Q, V) \).

(1) calculate all strings, \( q \), equivalent to \( Q \)

(2) for each \( q \) and for each \( L \in \{ 1, \ldots, M \} \) calculate \( T_k(qV_{1 \ldots L}, V_{L+1 \ldots M}) \).

**The First Transformation: The Table**

The first step in calculating \( T_k(Q, V) \) requires the value of \( (\text{LABEL} \ qW) \) and \( (\text{LABEL} \ qW) \) to be calculated for all strings \( q \) equivalent to \( Q \) and all strings \( W \) of length \( k \) or less. Define

\[ R_k(X) = \{ <W, (\text{LABEL} \ XW)> \mid W \text{ length } k \text{ or less} \} \]

Note that \( (\text{LABEL} \ X) \) is in \( R_k(X) \), paired with \( W = \"\" \). By definition, \( X \equiv_k Y \) if and only if \( R_k(X) = R_k(Y) \).

From the example above,

\[ R_0( a ) = \{ <\"\", 1> \} \]
\[ R_0(abb) = \{ <\"\", 1> \} \]

\[ R_1( a ) = \{ <\"\", 1>, <a,2>, <b,1> \} \]
\[ R_1(abb) = \{ <\"\", 1>, <a,0>, <b,2> \} \]

etc.
It is convenient to abbreviate \( R_k(X) \) as a tuple in which each position corresponds to a different value of \( W \). For \( k=1 \), \( R_k(X) \) will be written as a triple in which the first entry corresponds to \( W=\)", the second corresponds to \( W=a \), and the third corresponds to \( W=b \). Using this notation \( R_1(a)=<1,2,1> \) and \( R_1(abb)=<1,0,2> \).

The transformation of \( \langle X,(\text{LABEL X})\rangle \) into \( \{ R_k(X) \rangle XW\neq0 \text{ for some } W \text{ length } k \text{ or less} \} \) involves duplicating entries: one element of \( \{ R_k(X) \rangle XW\neq0 \text{ for some } W \text{ length } k \text{ or less} \} \) contains all the information needed in the first step of the calculation of \( T_k(Q,V) \). The two basic steps in calculating \( T_k(Q,V_1...V_M) \) are now:

1. find all \( q \) such that \( R_k(q)=R_k(Q) \)
2. for each \( q \) and for each \( L\in\{1,...,M\} \) calculate \( T_k(q,V_1...V_L,V_{L+1}...V_M) \).

Updating the set \( \{ R_k(X) \rangle XW\neq0 \text{ for some } W \text{ length } k \text{ or less} \} \) given a new labelled example \( \langle Z,\text{CLASS}_Z\rangle \) is as simple as updating \( \langle X,\text{LABEL X}\rangle \). For every combination of \( X \) and \( W \) such that \( XW=Z \) (\( W \) is length \( k \) or less), the position corresponding to \( W \) in \( R_k(X) \) is set to \text{CLASS}_Z.

\( \{ R_k(X) \rangle XW\neq0 \text{ for some } W \text{ length } k \text{ or less} \} \) can be regarded as a table indexed by \( X \). For instance, the labelled strings above would result in the following table \( (k=1) \).

<table>
<thead>
<tr>
<th>( X )</th>
<th>( R_k(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabb</td>
<td>(&lt;2,0,0&gt;)</td>
</tr>
<tr>
<td>abb</td>
<td>(&lt;1,0,2&gt;)</td>
</tr>
<tr>
<td>aba</td>
<td>(&lt;2,0,0&gt;)</td>
</tr>
<tr>
<td>ab</td>
<td>(&lt;1,2,1&gt;)</td>
</tr>
<tr>
<td>aa</td>
<td>(&lt;2,0,0&gt;)</td>
</tr>
<tr>
<td>a</td>
<td>(&lt;1,2,1&gt;)</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>(&lt;0,1,0&gt;)</td>
</tr>
<tr>
<td>OTHERS</td>
<td>(&lt;0,0,0&gt;)</td>
</tr>
</tbody>
</table>

\(^9\) for an alphabet of cardinality \( E \), the general formula for the number of positions in this tuple is \( (E^{k+1}-1)/(E-1) \) if \( E > 1 \)
and \( k+1 \) if \( E = 1 \)
The first step in the calculating $T_k(Q,V_1...V_M)$ may be broken into two parts:

1a) given $Q$, determine the value of $R_k(Q)$

1b) given $R_k(Q)$, find all $q$ such that $R_k(q)=R_k(Q)$

2) for each $q$ and for each $L \in \{1,...,M\}$ calculate $T_k(qV_1...V_L,V_{L+1}...V_M)$.

Second Transformation: The Inverted Table

For the purposes of updating the table (learning) and for determining $R_k(Q)$ given $Q$ (step 1a) the values of $R_k(X)$ are indexed by strings. For calculating all strings $q$ for which $R_k(q)=R_k(Q)$ (step 1b), the opposite is true: the strings are indexed by the given value of $R_k(Q)$. For this calculation, an inverted table would be a more appropriate organization:

<table>
<thead>
<tr>
<th>$R_k(X)$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2, 0, 0&gt;</td>
<td>{ aa, aba, abbb }</td>
</tr>
<tr>
<td>&lt;1, 2, 1&gt;</td>
<td>{ a, ab }</td>
</tr>
<tr>
<td>&lt;1, 0, 2&gt;</td>
<td>{ abb }</td>
</tr>
<tr>
<td>&lt;0, 1, 0&gt;</td>
<td>{ &quot;&quot; }</td>
</tr>
<tr>
<td>&lt;0, 0, 0&gt;</td>
<td>{ OTHERS }</td>
</tr>
</tbody>
</table>

An Alternative Definition of $T_k(Q,V)$

Up to this point, the classification procedure has been a literal implementation of the definition of $T_k(Q,V)$. The implementations of $T_k(Q,V)$ do differ for the different data structures used to represent hypotheses, but only in details that directly correspond to the differences in the data structures themselves (for example, in the exact way in which the set of strings equivalent to $Q$ is calculated).

The data structures and learning and classification procedures that will be considered henceforth are based on an alternative, equivalent definition of $T_k(Q,V)$. The new definition is based on the property

$T_k(qV_1...V_L,V_{L+1}...V_M)$ is always a subset of $T_k(qV_1,V_2...V_M)$
In other words, to calculate
\[ \bigcup_{I \in \{1, \ldots, M\}} T_k(qV_1 \ldots V_L, V_{L+1} \ldots V_M) \]
one need only calculate the first term, \( T_k(qV_1, V_2 \ldots V_M) \). Taking this into account, the
previous definition of \( T_k(Q, V) \) becomes
\[ T_k(Q, V_1 \ldots V_M) = \bigcup_{(q \equiv_{k} Q)} T_k(qV_1, V_2 \ldots V_M) \]

According to this definition, an unlabelled string is classified by replacing its first character
with any equivalent string, adjoining its second character to the replacement for the first,
replacing this string with an equivalent string, and so on, consuming one character at a time
from the original string until none remain. The modified procedure for calculating
\( T_k(Q, V_1 \ldots V_M) \) is:

1a) given \( Q \), determine the value of \( R_k(Q) \)
1b) given \( R_k(Q) \), find all \( q \) such that \( R_k(q) = R_k(Q) \)
2) for each \( q \), calculate \( T_k(qV_1, V_2 \ldots V_M) \).

With the new procedure, \( T_k(\"\", \text{abb}) \) would be calculated as follows (compare with the same
example (k=1) used above to illustrate the original definition):
\[
T_k(\"\", \text{abb}) = T_k(a, \text{bb})
\]
\[
T_k(a, \text{bb}) = T_k(\text{ab, b})
\]
\[
\cup T_k(\text{abb, b})
\]
\[
T_k(\text{ab, b}) = T_k(\text{ab, \"\"})
\]
\[
\cup T_k(\text{abb, \"\"})
\]
\[
= \{ \text{ab, abb} \}
\]
\[
T_k(\text{abb, b}) = T_k(\text{abb, \"\"})
\]
\[
= \{ \text{abb} \}
\]
\[
T_k(\"\", \text{abb}) = T_k(a, \text{bb}) = \{ \text{ab, abb, abbb} \}
\]

The significance of this alternative definition of \( T_k(Q, V) \) is that hypotheses can be
represented in ways that exploit its "one character at a time" nature.
Third Transformation: The Hypertable

In step (2) the modified procedure recursively calls itself with \( qV_1 \) as the new first argument. The first step inside this recursive call is to calculate \( R_k(qV_1) \). Because there are only a finite number of possible values for \( V_1 \), there are only a finite number of \( R_k(qV_1) \) for a given value of \( q \). This allows the construction of a hypertable containing \( R_k(qV_1) \) for all possible combinations of \( q \) and \( V_1 \). More precisely, \( \text{HTABLE}_k(X,U) = R_k(XU) \), for any string \( X \), and for any string \( U \) of length 0 or 1. The labelled strings above would result in the following hypertable \((k=1)\).

<table>
<thead>
<tr>
<th>( X )</th>
<th>&quot; &quot;</th>
<th>( U ) a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbb</td>
<td>( R_k(abb) )</td>
<td>( R_k(abbba) )</td>
<td>( R_k(abbb) )</td>
</tr>
<tr>
<td>abb</td>
<td>( R_k(ab) )</td>
<td>( R_k(aba) )</td>
<td>( R_k(abb) )</td>
</tr>
<tr>
<td>aba</td>
<td>( R_k(aba) )</td>
<td>( R_k(abaa) )</td>
<td>( R_k(abab) )</td>
</tr>
<tr>
<td>ab</td>
<td>( R_k(ab) )</td>
<td>( R_k(aba) )</td>
<td>( R_k(ab) )</td>
</tr>
<tr>
<td>aa</td>
<td>( R_k(aa) )</td>
<td>( R_k(aaa) )</td>
<td>( R_k(aab) )</td>
</tr>
<tr>
<td>a</td>
<td>( R_k(a) )</td>
<td>( R_k(aa) )</td>
<td>( R_k(ab) )</td>
</tr>
</tbody>
</table>

OTHERS \( R_k(OTHERS) \) \( R_k(OTHERS) \) \( R_k(OTHERS) \)

The specific contents of this hypertable are

<table>
<thead>
<tr>
<th>( X )</th>
<th>&quot; &quot;</th>
<th>( U ) a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbb</td>
<td>( &lt;2,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
</tr>
<tr>
<td>abb</td>
<td>( &lt;1,0,2&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
<td>( &lt;2,0,0&gt; )</td>
</tr>
<tr>
<td>aba</td>
<td>( &lt;2,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
</tr>
<tr>
<td>ab</td>
<td>( &lt;1,2,1&gt; )</td>
<td>( &lt;2,0,0&gt; )</td>
<td>( &lt;1,0,2&gt; )</td>
</tr>
<tr>
<td>aa</td>
<td>( &lt;2,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
<td>( &lt;0,0,0&gt; )</td>
</tr>
<tr>
<td>a</td>
<td>( &lt;1,2,1&gt; )</td>
<td>( &lt;2,0,0&gt; )</td>
<td>( &lt;1,2,1&gt; )</td>
</tr>
</tbody>
</table>

OTHERS \( <0,0,0> \) \( <0,0,0> \) \( <0,0,0> \)

A comparable extension based on the original definition of \( T_k(Q,V) \) is not possible, because it would involve storing \( R_k(qV_1...V_L) \) for all possible values of \( L \).

A hypertable contains precisely the same information as the simple table and the original list of string-label pairs. However, the hypertable provides the modified procedure for
calculating $T_k(Q,V)$ with immediate access to information that previously had to be calculated from the strings. In terms of a hypertable, the procedure for calculating the alternative definition of $T_k(Q,V_1...V_M)$ is:

(0) initialize $i$ to 1, and $R$ to $HTABLE_k(\"\",\") = R_k(\")$.

(1) find all rows, $X$, in the hypertable in which $HTABLE_k(X,\") = R$

(2) for each such row, set $R = HTABLE_k(X,V_i)$, and

\[ i < M: \text{increment } i. \text{ Iterate steps (1) and (2) with the new } R \text{ values.} \]

\[ i = M: \text{return every string } X \text{ such that } R_k(X) = \text{one of the new } R \text{ values} \]

Fourth Transformation: The (Compressed) Inverted Hypertable

In this procedure for calculating $T_k(Q,V)$ the string information in the hypertable plays two roles. First, $T_k(Q,V)$ is by definition a set of strings, and therefore the string information is essential for the calculation of $T_k(Q,V)$ even if it is only needed at the very end of the calculation, as in this case (step 2, $i=M$). However, the classification system requires only the classes of the strings in $T_k(Q,V)$ and not the strings themselves. The classes of all strings associated with any particular value of $R$ must be the same, since $R_k(X)$ contains (LABEL $X$). For example, the strings having $R_k(X) = \langle 1,0,2 \rangle$ are all in class "1". If instead of returning the strings associated with each $R$ value (step 2, $i=M$), this procedure returned the class associated with each $R$ value, it would be calculating (OUTPUT$_k$ $V$) directly.

The only remaining role for the string information in this procedure is to index the rows. The hypertable can be inverted so that $HTABLE_k(X,\")$ indexes the rows, as was done previously with the regular table. Inverting the above hypertable produces
<table>
<thead>
<tr>
<th>&quot;&quot;</th>
<th>a</th>
<th>b</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2, 0, 0&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>{ aa, aba, abbb }</td>
</tr>
<tr>
<td>&lt;1, 2, 1&gt;</td>
<td>&lt;2, 0, 0&gt;</td>
<td>&lt;1, 2, 1&gt;</td>
<td>{ a }</td>
</tr>
<tr>
<td>&lt;1, 2, 1&gt;</td>
<td>&lt;2, 0, 0&gt;</td>
<td>&lt;1, 0, 2&gt;</td>
<td>{ ab }</td>
</tr>
<tr>
<td>&lt;1, 0, 2&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>&lt;2, 0, 0&gt;</td>
<td>{ abb }</td>
</tr>
<tr>
<td>&lt;0, 1, 0&gt;</td>
<td>&lt;1, 2, 1&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>{ &quot;&quot; }</td>
</tr>
<tr>
<td>&lt;0, 0, 0&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>&lt;0, 0, 0&gt;</td>
<td>{ OTHERS }</td>
</tr>
</tbody>
</table>

The two rows indexed by <1,2,1> reflect the fact that $\text{HTABLE}_k(<1,2,1>,b)$ contains two distinct entries, one associated with string "a" and the other associated with "ab".

Inverting the hypertable in this way renders the string information irrelevant to the classification procedure and allows the inverted hypertable to be compressed. Compressing the string information out of the inverted hypertable above produces

<table>
<thead>
<tr>
<th>&quot;&quot;</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2, 0, 0&gt;</td>
<td>{ &lt;0, 0, 0&gt; }</td>
<td>{ &lt;0, 0, 0&gt; }</td>
</tr>
<tr>
<td>&lt;1, 2, 1&gt;</td>
<td>{ &lt;2, 0, 0&gt; }</td>
<td>{ &lt;1, 0, 2&gt;, &lt;1, 2, 1&gt; }</td>
</tr>
<tr>
<td>&lt;1, 0, 2&gt;</td>
<td>{ &lt;0, 0, 0&gt; }</td>
<td>{ &lt;2, 0, 0&gt; }</td>
</tr>
<tr>
<td>&lt;0, 1, 0&gt;</td>
<td>{ &lt;1, 2, 1&gt; }</td>
<td>{ &lt;0, 0, 0&gt; }</td>
</tr>
<tr>
<td>&lt;0, 0, 0&gt;</td>
<td>{ &lt;0, 0, 0&gt; }</td>
<td>{ &lt;0, 0, 0&gt; }</td>
</tr>
</tbody>
</table>

The following procedure calculates $(\text{OUTPUT}_k V_1...V_M)$ given an inverted hypertable (compressed or uncompressed):

1. (0) initialize $i$ to 1 and $R$ to $R_k("")$.
2. (1) for each entry $r$ in $\text{HTABLE}_k(R,V_i)$:  
   3. $i < M$: increment $i$ and repeat (1) using $r$ as the new $R$ value  
   4. $i = M$: return the class associated with each of the $r$ values

For instance, $(\text{OUTPUT}_k abb)$ would be calculated
initially, $i=1$, $V_1=a$, $V_2=b$, $V_3=b$, and $R=R_k(\"\")=<0,1,0>$. 

$i=1, R=<0,1,0>$:

$H_{TABLE_k}(<0,1,0>,a) = \{<1,2,1>\}$

for $i=2$, use $R = <1,2,1>$

$i=2, R=<1,2,1>$:

$H_{TABLE_k}(<1,2,1>,b) = \{<1,0,2>,<1,2,1>\}$

for $i=3$, use each of these values for $R$

$i=3, R=<1,0,2>$:

$H_{TABLE_k}(<1,0,2>,b) = \{<2,0,0>\}$

return [ 2 ]

$i=3, R=<1,2,1>$:

$H_{TABLE_k}(<1,2,1>,b) = \{<1,0,2>,<1,2,1>\}$

return [ 1 ]

Thus $(OUTPUT_k \text{ abb}) = \{ 2, 1 \}$

The transformation of the analogy system into the hypothesis modification system of [Biermann and Feldman 1972] is now complete. Each distinct value of $R_k(X)$ corresponds to a state of a nondeterministic finite state machine (NDFSM). $R_k(\"\")$ is the initial state. The class associated with state $R_k(X)$ is, of course, $(\text{LABEL X})$ (the entry in $R_k(X)$ paired with $W=\"\")$. The compressed inverted hypertable exactly corresponds to the state transition table of a NDFSM. Finally, the procedure for calculating $(OUTPUT_k V_1\ldots V_M)$ is precisely the algorithm for traversing a NDFSM given the input sequence $V_1\ldots V_M$. The inverted hypertable above corresponds to the following NDFSM:
A NDFSM Alone is Inadequate for the Specified Learning Task

[Biermann and Feldman 1972] gives the impression that their system is a hypothesis modification system operating directly on hypotheses represented as NDFSMs (compressed inverted hypertables). If this impression is accurate, information that is not available in an NDFSM alone, such as the information about specific labelled strings, must be irrelevant to the learning task. The impression is a false one. The NDFSM alone is inadequate for the specified learning task. This can be demonstrated in exactly the same way the inadequacy of decision trees was demonstrated in the previous section. In that section, the same decision tree was produced by two sets of examples, but the correct changes to the tree, given a new example, depended on which set of examples had actually been presented. In the following, the same NDFSM is produced by two sets of examples, but the correct changes to the NDFSM, given a new example, depends on which set of examples has actually been presented. In the NDFSMs in this example all arcs labelled "b" lead to the
state <0,0,0> and are not drawn in the figures.

If k=1 the following NDFSM is produced given <aa,1> or given <aa,1> and <baa,1>.

\[
\begin{array}{c}
\text{state } <0,0,0> \\
\text{output: 0} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{state } <1,0,0> \\
\text{output: 1} \\
\end{array}
\]  

It is impossible to determine on the basis of the NDFSM alone whether or not <baa,1> has been presented. However, if the labelled example <aaa,1> is now presented, different NDFSMs must be produced depending on whether or not <baa,1> has been presented. The two NDFSMs, shown in the following diagram, are identical except for the diagonal arc from state <0,1,0> to <1,0,0>. This arc is present if <baa,1> has been presented and absent if it has not.

\[
\begin{array}{c}
\text{state } <0,0,0> \\
\text{output: 0} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{a} \\
\downarrow \\
\text{state } <1,0,0> \\
\text{output: 1} \\
\end{array}
\]

The hyptable representation is adequate for the learning task and gives insight into the inadequacy of the NDFSM. The hyptable produced given <aa,1> is
In the NDFSM there is a diagonal arc labelled "a" from <0,1,0> to <1,0,0> if and only if there is a row in the hypertable with an entry of <0,1,0> in the first column and an entry of <1,0,0> in column "a". If the hypertable contained no row satisfying these requirements, the NDFSM would contain no arc from state <0,1,0> to state <1,0,0>. Only the second row in the hypertable satisfies these requirements. It does because the following conditions hold:

— the first column of this row is <0,1,0> because the classes of "a" and "ab" are unknown and the class of "aa" is known to be 1; and

— column "a" of this row is <1,0,0> because the class of "aa" is known to be 1 and the classes of "aaa" and "aab" are unknown.

If <baa,1> has not been presented, these conditions are necessary conditions for there being an arc from state <0,1,0> to state <1,0,0> in the NDFSM. If any of them were to change, e.g. if the class of "aaa" became known, the NDFSM would no longer have an arc from state <0,1,0> to state <1,0,0>.

The effects of presenting <baa,1> is indicated in the hypertable in parentheses. Although presenting <baa,1> does not change the columns that define the NDFSM it does create a second set of sufficient conditions for the arc from state <0,1,0> to state <1,0,0> namely,

— the first column is <0,1,0> because the classes of "ba" and "bab" are unknown and the class of "baa" is known to be 1; and

— column "a" is <1,0,0> because the class of "baa" is known to be 1 and the classes of "baaa" and "baab" are unknown.

Having several sets of sufficient conditions for an arc means that no one set of conditions is necessary. For example, after <aaa,1> is presented the first set of conditions no longer holds. But <aaa,1> does not affect the second set and therefore the arc from <0,1,0> to <1,0,0> remains in the NDFSM. This explains why different NDFSMs must be produced.
depending on whether or not \(<\text{bba}, 1>\) has been presented. In conclusion, an NDFSM does not represent all the information necessary to update it correctly.

6.2.3 The Relation Between Analogy and Hypothesis Modification Systems

The preceding examples have shown that at least some analogy systems can be transformed into mediating systems. The mediating structure in the first example was a sorted list, and in the second example an uncompressed inverted hypertextable. These structures are sufficiently well-suited to both the learning and performance tasks that the systems employing them are transitional between mediating and hypothesis modification systems.

The examples have also shown that the representations ideally suited to the performance task are sometimes specialized to the extent that they are inadequate for the learning task. [Porter and Bareiss 1986] makes the general claim that such "highly compiled" representations are always too inflexible (i.e. adequate for very few tasks), and concludes that analogy systems are the only choice when flexibility is required. The examples presented in this chapter show this claim is true only in the most extreme cases. Mediating structures demonstrate that the tradeoff between efficiency and flexibility is not strictly linear: there are representations that are both flexible and very efficient. A useful effect of behaviour-preserving transformations is that a sequence of systems can be produced with different proportions of flexibility and efficiency. Furthermore, the transformations compress and re-organize the performance history, sometimes on the basis of task-specific knowledge. Studying the transformations exposes "where" and "how much" flexibility has been traded for efficiency.

In an analogy system the matching process is, of course, a function of the both the set of prototypes (performance history) and the current performance-task input. Often there are aspects of the matching process that depend only on the set of prototypes. The learning system's processes and representations can be modified to exploit these aspects of the
matching process. Sometimes whole subprocesses can be shifted into the learning system and merged with processes there. This type of modification to the matching and learning systems can transform an analogy system into a mediating system or hypothesis modification system. It is called the compilation of the matching process.

Both examples illustrate the compilation of the matching process. In the first example, matching consists of finding the integer in a sorted list closest to a given integer. The sequence of comparisons involved in this process is almost entirely a function of the sorted list. The given integer determines the outcome of the comparison, but because there are only two possible outcomes of each comparison it is possible to enumerate all possible sequences of comparisons on the basis of the sorted list alone. The construction and simplification of this sequence could therefore be shifted into the learning system and a representation (decision tree) devised that is well-suited to the performance task.

A similar line of reasoning applies to the second example. Matching consists of successively rewriting initial segments of a given string to produce a labelled string. The allowable rewritings are determined largely by the set of labelled strings. As in the first example, the effects of the given string on the allowable rewritings could be reduced to a finite number of possibilities defined by the set of possible characters and the set of possible equivalence classes of strings (states of the NDFSM). Thus it was possible to enumerate all possible sequences of rewritings on the basis of the set of labelled strings alone. This enumeration process was shifted into the learning system, and a representation devised (the hypertable) that was well-suited to both the learning and performance tasks.

Having seen the matching processes in analogy systems compiled, in a literal sense, into prominent processes in a hypothesis modification system, one is led to inspect the resemblance between the matching techniques in existing analogy systems and the generalization techniques in existing hypothesis modification systems. In some cases, the similarities are very strong.
For example, the analogy system of [Porter and Bareiss 1986] uses "knowledge-based pattern matching" to decide if the current P-task instance should be classified in the same way as a particular prototype. The generalization technique called "structure matching" [Vrain 1986] is a very similar process. The major differences between these systems are that structure matching returns a formula that matches both objects whereas knowledge-based pattern matching returns a strength of match, and no formula. These differences are historical and do not reflect a fundamental difference between the two types of system. [Anderson 1986] has shown that by simply recording (and "variabilizing") specific applications of the matching process analogy systems can create formulae equivalent to the generalizations produced by hypothesis modification systems. And very recent extensions of structure matching do, in fact, incorporate an exact counterpart to the strength of match 10. Generalization and analogical matching are thus very closely related process.

6.3 Important Relationships Not Covered in this Chapter

The relationships between classes established in this chapter have been of the form "there exist systems in one class whose behaviour is very closely related to that of systems in another class". It has certainly not been established that there are no important differences between systems in different classes, or indeed between systems in the same class.

Two important differences not covered in the present chapter are the following. First, the efficiency of systems can vary tremendously. Secondly, the performance history information and explicit task-specific knowledge can be used in significantly different ways. In the most efficient systems the performance information and task-specific knowledge provides a direct indication of the the next hypothesis. In the least efficient systems this information is used as an explicit test of the quality of a candidate, and an extensive search of candidate space is required to find the next hypothesis. Between these two extremes are systems that analyze the performance history and/or the current hypothesis and extract patterns of

10 personal communication, Michel Manago (summer 1987)
behaviour required of subsequent hypotheses. The next chapter investigates the efficiency of and the different ways of using information in enumerative systems and extends the results to systems of other types.
CHAPTER 7

THE EFFICIENCY OF THE LEARNING PROCESS

7.1 Using Performance Information to Improve Efficiency

Efficiency can be studied at any level in the semantic context and measured in many different ways. Formal investigations typically study the efficiency of a learning task and measure efficiency in terms of coarse classes, such as learnable in the limit, learnable in nondeterministic polynomial time, and learnable in deterministic polynomial time. A few investigations, e.g. [Bitner 1977], study the efficiency of specific implementations of systems, measured in terms of CPU cycles and bits of storage.

The efficiency problem addressed in this chapter is a restricted version of a general question highlighted by the abstract model of the learning process. In this model, performance information is used to evaluate candidates, and candidate set is re-enumerated each time new performance information is presented. The general question is "how can a system select a hypothesis without evaluating every candidate?".

The proposed answer has two parts. The first, described in Chapters 3 and 4, is that regularities in the selection criterion, a type of task-specific knowledge, may be provided to the system. For example, a hypothesis in BOXES consists of a large number of components ("boxes"), and a hypothesis is selected by independently selecting values for each of its components. This hypothesis selection strategy exploits the decomposability of the selection criterion. The second part of the answer is that in the model performance information is used only to evaluate individual candidates. Performance information could also be used to modify the sequence in which candidate set is enumerated.
Efficiency is measured in terms of the number of "partially specified candidates" evaluated in the course of selecting a new hypothesis. The notion of "partially specified candidate" is made more precise later. It is introduced in order to make the efficiency measure appropriate for systems, such as BOXES, that select a hypothesis by successive refinement. In BOXES a partially specified candidate is one fully-specified box. Because BOXES evaluates all of the A possible actions in each of the B boxes in a hypothesis its efficiency is $A \times B$.

This chapter makes two restrictive assumptions in its investigation of the use of performance information to guide the selection process. First, it assumes performance information is a constraint of the form "all future hypotheses must (must not) satisfy P", where P is either a predicate over candidates or a relation between a candidate and the sequence of hypotheses proposed up to the time of evaluation of the relation. Performance information is in this form in many learning tasks. For example, in most concept learning tasks each classified example, $\langle X, \text{CLASS}_X \rangle$, provides the information "all future hypotheses must classify X as $\text{CLASS}_X$". Nevertheless there are many tasks satisfying the requirements of Chapter 2 in which performance information is not in this form. For example, the requirements in Chapter 2 are satisfied by learning tasks in which performance information indicates preferences or functions over candidates to be optimized. Also, there are tasks in which performance information is in the form "all solutions must (must not) satisfy P", which is not always equivalent to "all future hypotheses must (must not) satisfy P". The second restriction is that the investigation only considers the use of a single constraint to guide the selection process. Difficult issues arising from the interactions or tradeoffs between different constraints and/or preferences are left as future work. What remains to be investigated is this: to what extent and under what conditions can a learning system improve its efficiency upon being given one constraint from a family of possible constraints.
7.1.1 Example Problem

The following example will be used to illustrate the techniques and main issues associated with the problem of using performance information to improve the efficiency of a system. A candidate is a pair \(<q, r>\) of predicates over objects, where \(q = \{q_1...q_5\}\) and \(r = \{r_1...r_5\}\). Object \(X\) is a member of \(<q, r>\) if and only if \((q \ X) \& (r \ X)\). The sets of predicates may be thought of as attributes and the predicates they contain as values. Predicates are partially ordered: \(p_1 \leq p_2\) (\(p_2\) is "more general" than \(p_1\)) if and only if \((p_1 \ X) \Rightarrow (p_2 \ X)\). \(p_2\) is called a minimal generalization of \(p_1\) if and only if \(p_1 \leq p_2\) \& \((p_1 \leq V \leq p_2 \Rightarrow (V = p_1 \text{ or } V = p_2))\). The partial order on predicates in this example is

\[
\begin{array}{ccc}
q_1 & \quad & x_1 \\
q_2 & \quad & x_2 \\
q_3 & \quad & x_3 \\
q_4 & \quad & x_4 \\
q_5 & \quad & x_5 \\
r_1 & \quad & \text{(maximal)} \\
r_2 & \quad & \text{(minimal)} \\
r_3 & \quad & r_4 \\
r_5 & \quad & r_5 \\
\end{array}
\]

\(<q, r>\) will be abbreviated \(q_r\). Figure 7.1 shows the partial order on the candidate set, \(q_r \leq q_m\) \(r_n\) if and only if \((q_n \leq q_m) \& (r_n \leq r_n)\), induced by the partial order on predicates.

This example will compare the uses of two different constraints. The first constraint is "No future hypotheses may involve \(q_5\) or \(r_5\)." In other words, none of the candidates \(\{q_5 r_1, q_5 r_2, \ldots, q_5 r_5, q_5 r_1, \ldots, q_5 r_5\}\) may be proposed as a hypothesis. The second constraint is "no future hypothesis may be more specific than (or equal to) \(q_2 r_2\)." This specifies that none of \(\{q_2 r_2, q_3 r_3, q_2 r_4, q_3 r_4, q_3 r_3, q_2 r_5, q_4 r_2, q_3 r_5, q_4 r_3, q_4 r_4\}\) may be proposed as a hypothesis.

Each of these constraints specifies nine candidates that need never be evaluated in the hypothesis selection process. By eliminating the need to evaluate more than one third of the candidates each constraint promises considerable improvement in the efficiency of hypothesis selection. The questions addressed in this chapter are "how much improvement in efficiency can actually be achieved?" and "what factors affect the amount of improvement that can be achieved?".
7.1.2 Techniques for Using Performance Information

Performance information is task-specific knowledge. Chapter 4 discusses methods by which a human designer may provide task-specific knowledge to a system at the time of choosing a representation of the semantic domain in some syntactic domain. The methods discussed need not be applicable to the present problem because in the present problem a specific representation and system already exist. That is, the present problem may be regarded as one of incremental redesign, i.e., redesign given new task-specific knowledge and the existing design but not the task-specific knowledge used to produce the existing design. Useful insight into the incremental redesign task can result by considering how the new knowledge would have changed the original design had it been available. For example, these considerations can produce an upper bound on the amount of improvement in efficiency that will result from redesign and they can produce a rough estimate of the
amount of effort and "restructuring" required to obtain this gain in efficiency.

**Data-Driven Learning**

The first method described in Chapter 4 is to provide task-specific knowledge explicitly, for example, by providing a value for a parameter. The automatic counterparts of this method are called **data-driven** techniques. One data-driven technique is to extract from performance information values for the parameters to hypothesis modification (or other) operators. Another technique is to use performance information to define a small subset of the candidate set, i.e. an active set, and restrict the selection process to candidates in the active set. As seen in section 6.1.2 this is used in many systems, not just candidate set modification systems. Perhaps the most powerful of the data-driven techniques is to use performance information directly as the specification of (part of) the hypothesis, as is done in performance history update systems.

**Test Compilation**

Chapter 4 briefly mentioned the idea of using task-specific knowledge to specialize a general system, e.g. by inserting the knowledge as a test at an appropriate point in the system. For example in most hypothesis modification systems tests are applied to the candidate produced by an operator sequence in such a way that extensions of the operator sequence are evaluated only if the candidate is sufficiently promising. As a second example, in some systems a hypothesis is selected (constructed, modified) component by component and explicit tests are applied to individual components.

Tests that occur early in the hypothesis selection process evaluate an entire set of candidates each time they are applied. In hypothesis modification systems candidates are corepresented with subsets of the candidate set: a test of a candidate can be interpreted as a test of the set of candidates represented by that candidate. As a general rule, the earlier in the process a test is applied the greater the efficiency. Thus, task-specific knowledge provided by inserting tests, will result in greater improvements in efficiency if the tests are
inserted early in the hypothesis selection process. Inserting the test as the last step in the hypothesis selection process results in no improvement in efficiency: exactly the same number of candidates are evaluated before and after inserting the test at this point.

The insertion of a test may involve merging it with existing processes. For example, explanation-based generalization [Mostow and Bhatnagar 1987] regresses a new test through a sequence of operators. This permits the test to be applied before the operator sequence rather than afterwards.

**Constructive Induction**

Chapter 4 introduced the principle that efficiency could be obtained by choosing a representation such that regularities in the domain correspond to regularities in the behaviour of the system. The most straightforward applications of this principle are based on what might be called unconditional regularities. For example the unconditional regularity "under all conditions X and Y require identical processing" permits X and Y to be corepresented, and the regularity "under no condition will candidate H be selected" permits H to be unrepresented in the system. Applications of the principle can also be based on conditional regularities such as "if P holds, X and Y require identical processing" or "if Q holds, H will never be selected". A conditional regularity that is very commonly exploited is "if H misclassifies a seen example it will never be selected". This regularity permits the use of monotone decreasing active sets.

There is an exact parallel between these types of regularity and the two types of constraint: constraints involving predicates over candidates are unconditional regularities and constraints involving relations between a candidate and the sequence of hypotheses proposed up to the time of evaluation of the relation are conditional regularities. However, because a learning system only has access to a specific representation of the semantic domain and not to the domain itself, it cannot implement this principle exactly. It can partially implement the principle by creating a representation of one syntactic subdomain in another syntactic subdomain. The general problem of change of representation is called
"constructive induction".

7.1.3 Applying These Techniques to the Example Problem

A very efficient data-driven technique is to evaluate only those candidates expressible using combinations of terms that occur in the seen examples. This technique naturally avoids the candidates specified by the first (or second) constraint in the example problem as long as no examples occur in which q5 or r5 (or q2&r2) are true. It may be a property of the task that no such examples will occur, e.g. there may be no objects in the semantic domain for which q5 or r5 (or q2&r2) are true. In this case the constraint provides no task-specific knowledge beyond that provided by mapping this property of the semantic domain onto the behaviour produced by this data-driven technique. On the other hand, it may be that such examples can occur but that hypotheses involving these predicates are unacceptable in the task for "extra evidential" reasons. If examples in which q5 or a5 (or q2&r2) are true can occur, a technique is needed that explicitly uses the constraint.

One technique is to use the constraint in a preprocessing step that translates each classified example into the set of minimal candidates that satisfy the constraint and are compatible with the classified example. For instance, \{q4r1,q1r4\} is the set of minimal candidates that include q4r4 and satisfy the constraint "not more specific than or equal to q2r2". The sets of minimal candidates are presented to the system in lieu of the classified examples. This technique is only applicable if either each classified example translates into one candidate or the learning system can cope with "disjunctive" examples, such as \{q4r1,q1r4\}.

The constraint could, of course, be inserted as a test in the evaluation of candidates. As noted earlier, using a constraint in this way does not improve the efficiency of a system.

The applicability of other techniques depends on the way the set of candidates is represented in the system. Suppose it is represented in a way that reflects the partial order, for example, as the following set of "specialization" operators.
\[
(1) \ q1R \rightarrow q2R \qquad (5) \ Qr1 \rightarrow Qr2 \\
(2) \ q1R \rightarrow q5R \qquad (6) \ Qr1 \rightarrow Qr5 \\
(3) \ q2R \rightarrow q3R \qquad (7) \ Qr2 \rightarrow Qr3 \\
(4) \ q2R \rightarrow q4R \qquad (8) \ Qr2 \rightarrow Qr4
\]

The change in behaviour specified by a constraint can be achieved by making appropriate modifications to this set of rules. Candidates involving either q5 or r5 can be eliminated from the candidate set by deleting rules (2) and (6). Candidates involving both q2 and r2 cannot be eliminated by deleting rules. They can be eliminated by adding a test (precondition) to rules (1) and (5) to produce

\[(1') \ q1R \rightarrow q2R \textbf{ unless } R \leq r2 \qquad (5') \ Qr1 \rightarrow Qr2 \textbf{ unless } Q \leq q2\]

The only values of R for which rule (1') is applicable are r1 and r5. Similarly, the only values of Q for which rule (5') is applicable are q1 and q5. Thus (1') and (5') are equivalent to

\[(1a) \ q1r1 \rightarrow q2r1 \qquad (5a) \ q1r1 \rightarrow q1r2 \\
(1b) \ q1r5 \rightarrow q2r5 \qquad (5b) \ q5r1 \rightarrow q5r2 \]

Other systems, such as Edinburgh Focussing, represent a candidate set by operators that take a partial order as a parameter. In such systems, constraints can be put into effect by modifying the partial orders. The first constraint requires deleting q5 and r5 from the partial orders to produce

\[
\begin{array}{c}
q1 \\
/ \\
q2 \\
/ \\
q3 \quad q4 \\
/ \quad / \\
q3 \quad q4 \\
\end{array}

\begin{array}{c}
r1 \\
/ \\
r2 \\
/ \\
r3 \quad r4 \\
\end{array}
\]

The second constraint cannot be put into effect so easily. [van Someren 1987] proposes merging the two predicate partial orders and eliminating the combinations prohibited by the constraint. In the partial order produced by this technique, shown below, there are predicates, e.g. q5r3, having several minimal generalizations. Thus the technique is not applicable in systems such as Edinburgh Focussing whose operators require each predicate

- 113 -
to have a unique minimal generalization.

Another technique is to replace the single focussing space with a set of focussing spaces analogous to the multiple version spaces used in [Mitchell 1978] (pp. 105-130). Using this technique candidates in which q2&r2 is true are eliminated by using two spaces, one in which r2 exists but q2 does not and another in which q2 exists but r2 does not.

Focussing occurs in both spaces simultaneously. Note that candidates involving neither q2 nor r2 are in both spaces.
7.1.4 The Effectiveness with which Performance Information is Used

The **effectiveness** with which a system uses performance information is defined to be the improvement in efficiency achieved by the system given that information. The preceding example demonstrates that effectiveness is influenced by specific properties of the performance information and of the internal organization of the learning system. The two example constraints are similar in many ways. Both specify nine candidates and the candidates specified are connected to the others by a similar number of arcs (8 for the first constraint, 6 for the second). And both have the property that if \( X \geq Y \) and \( Y \) satisfies the constraint then \( X \) satisfies the constraint. Despite these similarities techniques applicable to the first constraint are very often inapplicable to the second. The applicability of techniques is also limited to learning systems with specific properties. Several of the techniques substituted a set for a single item; not all systems can cope with such a substitution.

The remainder of the chapter defines a property of a system called generative structure and investigates the three-way relationship between this property, constraints, and the effectiveness with which constraints can be used. The investigation is carried out in detail for enumerative systems and the results transferred to other types of system. There are two reasons for using enumerative systems to study efficiency problems. First, both the behaviour and internal organization of enumerative systems are simple and amenable to analysis. Second, there is a widespread belief that enumerative systems are inherently inefficient. The investigation that follows successfully challenges this belief. The techniques for using performance information, and the improvements in efficiency that result from its use, are similar in all types of system. Enumerative systems employing these techniques tend to be transitional with other types of system. The general rule that emerges is that the effectiveness with which performance information can be used by a system is determined by the system's generative structure and not by its gross architectural classification.
7.2 Effectiveness and the Structure of Enumerative Systems

The general enumerative system defined in Chapter 5 consisted of three components, a candidate enumerator, a tester that eliminates some of the candidates produced by the enumerator, and a selector that chooses the next hypothesis from those candidates passed by the tester. In keeping with the assumption that performance information is in the form of a constraint, this section shifts the selector component into the performance system and considers only the enumerator and tester components.

What techniques for using performance information to improve a system's efficiency are applicable to enumerative systems? Performance information could be inserted as a test; if inserted early enough in the enumeration process this would be an effective use of the information. Alternatively a resumption point could be calculated from the performance information. For example, many grammar enumeration systems construct a "canonical grammar" from the performance information and use this grammar as a resumption point. A dynamically determined resumption point is a simple form of data-driven enumeration. There are other applicable techniques: these are illustrated in the case study below and summarized in section 7.2.4.

7.2.1 Reconfigurable Enumerators

It is one thing to calculate a resumption point; it is quite another to arrange for an enumerator to proceed from an arbitrarily chosen point. For example, consider algorithms for enumerating the sequence of moves in the solution of the N-disk Towers-of-Hanoi puzzle. A resumption point could be specified either by the state of the puzzle or by the move number. Some algorithms can very easily be adapted to initiate their enumeration from an arbitrarily specified state but not from an arbitrarily specified move number. For others, the opposite is true. The best-known algorithm cannot be easily adapted to accept either type of resumption point. Being recursive it requires the resumption point to be
specified in terms of the contents of a stack of procedure calls!

This example illustrates the idea of a reconfigurable enumerator. An enumerator is reconfigurable if it can modify its enumeration sequence in response to "specifications" expressed in an algorithm-independent way. A specification, in this context, is a description of an enumeration sequence, or of a change to an (implied) enumeration sequence, that is expressed in terms of semantic entities (e.g. "enumerate only integer multiples of 2"), syntactic entities (e.g. "enumerate only strings in which the right-most bit is OFF"), or properties of sequences (e.g. "skip every second item in the present enumeration sequence").

The notion of a reconfigurable enumerator arises elsewhere in the Artificial Intelligence literature. DENDRAL's design, described as the "plan-generate-test" paradigm, explicitly calls for enumerators that can be reconfigured by the user and/or the output of the planning step.¹ Hierarchical enumerative systems always include reconfigurable enumerators. In this type of system the output of an enumerator at one level is used to reconfigure the enumerators at the next lower level; actual candidates are enumerated only at the lowest level. For example, in [Waterman 1975] and SPARC the candidates describing periodic sequences are enumerated in two stages. One enumerator produces period lengths. A second enumerator produces all the candidates describing sequences with a given period length. If none of the candidates with the given period length are selected, the next period length is enumerated and the process repeats.

Performance information in the form of constraints is a specification to a reconfigurable enumerative system. As witnessed in the Towers-of-Hanoi example, there is considerable variability in the effectiveness with which different systems can use specifications. The property of the system that determines which constraints can be used effectively by a system is its generative structure.

¹ pp. 53-63 in [Lindsay et al. 1980]
7.2.2 The Generative Structure of an Enumerative System

An enumerator of a set $S$ is a procedure that produces a sequence of elements of $S$ in which every element of $S$ eventually occurs. The intuitive idea underlying generative structure is that the internal organization of an enumerator constitutes a declarative form of the enumeration sequence. Generative structure will sometimes be regarded as a property of the enumerator and at other times as a property imposed by the enumerator on the sequence or set that is enumerated. To use information effectively a system must change the enumeration sequence. The general observations in Chapter 3 indicate that the representation of the enumeration sequence, i.e. generative structure, will determine the efficiency and even the applicability of techniques for using information. Chapter 3 did not consider the influence of generative structure on effectiveness.

The following example suggests generative structure has a very great influence on the effectiveness with which a given constraint can be used. Suppose $S = A \cup B$ and $S = C \times D$. One enumerator of $S$ might proceed by enumerating the elements of $A$ and the elements of $B$; another might proceed by enumerating elements of $C$ and of $D$ and combining these elements in all possible ways. The first enumerator can easily be reconfigured to make very effective use of the constraint "do not consider candidates in $A$". If it happens that $A = C' \times D'$ ($C' \subseteq C$, $D' \subseteq D$), the second enumerator can also be reconfigured to make effective use of the constraint. If $A$ is not be expressible as $C' \times D'$ it may be very difficult to reconfigure the second enumerator to make use of the constraint.

This intuitive notion has been formalized for three types of enumerative system. The first type enumerates set $S$ by enumerating the elements of each of $S_1$, $S_2$, ..., $S_k$, a collection of sets covering $S$. The order in which elements are enumerated is unrestricted except for the requirement that every element must eventually be enumerated. This type of system has (and imposes on $S$) the generative structure $[\text{UNION} : S_1, \ldots, S_k]$. 

- 118 -
The second type of enumerative system involves a total, onto function \( F : S_1 \times S_2 \times \ldots \times S_k \rightarrow S \). It enumerates the elements of \( S_1, \ldots, S_k \) and applies \( F \) to every combination of elements. This type of system imposes on \( S \) the generative structure \([F:S_1, \ldots, S_k]\). It is natural to view some enumerators of this type as imposing a structure on each element of \( S \). For example, if \( A \) and \( B \) are sets of strings, it is natural to view each string in the set with structure \([\text{concatenation}:A,B] \) as consisting of two parts, an initial segment from \( A \) and a final segment from \( B \).

The third type of enumerative system involves a function, called a test, \( F : S_1 \times S_2 \times \ldots \times S_k \rightarrow \{\text{true}, \text{false}\} \). Values mapped to false fail \( F \); those mapped to true are said to pass. \( F \) is applied to every combination of elements in \( S_1, \ldots, S_k \) and only the combinations which pass are enumerated. This type of system imposes on \( S = \{s \in S_1 \times S_2 \times \ldots \times S_k \mid F(s) \text{ is true} \} \) the generative structure \([F:S_1, \ldots, S_k]\). Although the notation is the same as for the second type of enumerative system, the two types of system can always be distinguished by the range of \( F \). Enumerators other than these three types will be regarded as primitive enumerators imposing on the sets \( S \) they enumerate the null generative structure, denoted \([S]\).

The \( S_i \) in the generative structure \([F:S_1, \ldots, S_k]\) (or \([\text{UNION}:S_1, \ldots, S_k]\)) of set \( S \) are called \( F \)-constituents (\( \text{UNION}-\text{constituents} \)) of \( S \), abbreviated constituents in contexts where the meaning of \( F \) is clear. Constituents of \( S \) may or may not be subsets of \( S \).

The generative structure of \( S \) includes (recursively) the generative structures imposed on its constituents by their respective enumerators. For example, if the enumerator of \( S \) imposes \([\text{UNION}: X,Y] \), and the enumerator of \( Y \) imposes \([\text{concatenation}:A,B] \), and the enumerators of \( X \), \( A \), and \( B \) are primitive, then the generative structure of \( S \) is fully specified as \([\text{UNION}: [X], [\text{concatenation}:A],[B]]\).

In a hierarchical enumerator several distinct constituents are generated by the same reconfigurable enumerator. A set of specifications is enumerated sequentially, and the constituent corresponding to one specification is enumerated in entirety before the next
specification is considered. The generative structure imposed on a set by a hierarchical
enumerator is

\[ \text{[UNION \{constituent specifications\}: reconfigurable enumerator].} \] When fully
specified this generative structure includes both the generative structure of the set of
constituent specifications and the generative structure imposed on the constituents by the
reconfigurable enumerator.

The reconfigurable enumerator may itself be hierarchical, imposing the generative structure

\[ \text{[UNION \{SPEC}_1\} [\text{UNION \{SPEC}_2\}: R-GEN] ]} \]

Each \( \text{SPEC}_1 \) in \( \{ \text{SPEC}_1 \} \) is the specifications of a "top level" constituent. Each \( \text{SPEC}_2 \) in
\( \{ \text{SPEC}_2 \} \) is the specification of a constituent within a particular top level constituent. That
is, the set enumerated by R-GEN is determined by both \( \text{SPEC}_1 \) and \( \text{SPEC}_2 \). This is
sometimes made explicit by describing the elements of \( \{ \text{SPEC}_2 \} \) as specifications of the
form "\( \text{SPEC}_1 \& \text{SPEC}_2 \)."

The generative structure \([F:S_1, \ldots, S_n]\) (or \([U\text{NION:S}_1, \ldots, S_n]\)) describes a set, not a sequence.
For example, \([U\text{NION:S}_1, \ldots, S_n]\) does not convey any information about the sequence in
which the elements of \( S_1 \) and \( S_2 \) are enumerated. It is imposed by systems that enumerate
\( S_1 \) in entirety (assuming \( S_1 \) is finite), then \( S_2 \). It is also imposed by systems that alternate
between enumerating \( S_1 \) and \( S_2 \). In the present framework sequence information is
conveyed only for hierarchical enumerators. \([\text{UNION is \{1,2\}: (enumerator of \( S_j \))}\] describes
a system that enumerates all the elements of \( S_1 \) followed by all the elements of \( S_2 \).
The Generative Structure Imparted By a General Enumerator of Tuples

Of particular importance in the following case study are enumeration sequences of \( S_i \times T \) with the property that if \( X \) occurs before \( Y \) in the enumeration of \( S_i \) then all \( <X,t> \) pairs occur before the first \( <Y,t'> \) pair. If \( T=S_2 \times T' \), \( T'=S_3 \times T'' \) and so on are enumerated in the same fashion, a sequence of k-tuples is produced having the property that, for all \( i \), if \( X \) occurs before \( Y \) in the enumeration of \( S_{i+1} \) then all k-tuples of the form \( <a_1,a_2,...,a_i,X,...> \) occur before the first k-tuple of the form \( <a_1,a_2,...,a_i,Y,...> \).

A general purpose enumerator for this type of sequence is

\[
\text{TUPLES Spec} = \\
\text{IF} \ (\text{END_OF_TUPLE Spec}) \text{ THEN} \ [< ] \\
\text{ELSE UNION over} \ (\text{Element,New_Spec}) \in (\text{EXPAND Spec}) \text{ of} \\
\quad \{ \ (\text{CONS Element}) \ (t) \ | \ t \in (\text{TUPLES New_Spec}) \} \\
\text{where} \\
\quad (\text{CONS e}) \ (<t_1,...,t_k>) = <e,t_1,...,t_k>
\]

"Spec" specifies a set, \( S_i \). "EXPAND" maps a specification onto a set of pairs. The first entry in each pair is the element of \( S_i \) that will occupy the \( i \)th position in the tuple. The second entry is a specification of \( S_{i+1} \); thus \( S_{i+1} \) may depend on the first entry in the pair.

A simple example that will be used in the case study is the following algorithm for enumerating all tuples of a particular size of elements of a given set.

\[
\text{ALL_TUPLES} <\text{Size,Set}> = \\
\text{IF} \ (\text{Size}=0) \text{ THEN} \ [< ] \\
\text{ELSE UNION over} \ [<s,<\text{Size-1,Set}>] \ | \ s \in \text{Set} \text{ of} \\
\quad \{ \ (\text{CONS s}) \ (t) \ | \ t \in (\text{ALL_TUPLES} <\text{Size-1,Set}>) \} \\
\]

\text{ALL_TUPLES is simply TUPLES with the definitions}

\[
\text{Spec} = <\text{Size,Set}> \\
\text{END_OF_TUPLE} <\text{Size,Set}> = \text{Size}=0 \\
\text{EXPAND} <\text{Size,Set}> = \{ <s,<\text{Size-1,Set}> | s \in \text{Set} > \}
\]
As a second example, TUPLES will enumerate all permutations of the elements of set S given the following definitions.

Spec = {elements remaining to be placed}, initially all of S.
END_OF_TUPLE = no elements remain to be placed (Spec = {}).
EXPAND \{s_1, \ldots, s_k\} = \{ <s_1, [s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k]> | i \in \{1, \ldots, k\}\} \]

The sequence in which the tuples (permutations) are enumerated has the property that, for all i, if X occurs before Y in the enumeration of S_{i+1} then all tuples (permutations) of the form <a_1, a_2, \ldots, a_i, X, \ldots> occur before the first tuple (permutation) of the form <a_1, a_2, \ldots, a_i, Y, \ldots>.

The generative structure imposed by TUPLES on the sequence it enumerates is

\[
[\text{UNION} \{<e_1, S_2>\}):
\quad \text{(CONS } e_1):\]
\quad [\text{UNION} \{<e_2, S_3>\}):
\quad \quad \ldots
\quad [\text{UNION} \{<e_i, S_{i+1}>\}):
\quad \quad \text{(CONS } e_i):\]
\quad \quad \quad \ldots
\quad \text{[(CONS } e_k):[<>]\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ld
constituent of the set of tuples beginning \(<e_i,...,e_p>\). Because they are elements of the same constituent, all tuples beginning \(<e_i,...,e_i>\) occur consecutively in the enumeration sequence.

Tests may be inserted in this generative structure in two different ways, around \([\{CONS \ e_i\}:...]\), as in

\[
[\text{UNION} \ \{<e_i, S_{i+1}^>\}:
\{CONS \ e_i\}:
\ldots
[\text{UNION} \ \{<e_i, S_{i+1}^>\}:
\{CONS \ e_i\}:
\ldots
\{CONS \ e_k\}:(<>)]\ldots]\ldots]
\]

or around \(<e_i,S_{i+1}^>\), as in

\[
[\text{UNION} \ \{<e_i, S_{i+1}^>\}:
\{CONS \ e_i\}:
\ldots
[\text{UNION} \ \{TEST: \{<e_i, S_{i+1}^>\}\}:
\{CONS \ e_i\}:
\ldots
\{CONS \ e_k\}:(<>)]\ldots]\ldots]
\]

A test inserted in the first way is applied to individual k-tuples in the final stage of construction: values for all the entries in the k-tuple have been determined and the rightmost entries have been assembled into a (k-i+1)-tuple. Except for the negligible cost of assembly (CONSing), inserting a test in this way is no more effective than applying it to fully constructed individual k-tuples, as in \([\text{TEST: [UNION} \ \{<e_i, S_{i+1}^>\}:\{CONS \ e_i\}:...]]\).

A test inserted in the second way is applied to specifications of the form "all k-tuples whose first i entries are \(<e_i,...,e_p>\". Each specification that fails the test eliminates, in a single stroke, all the k-tuples beginning \(<e_i,...,e_i>\). The smaller the value of i the more effective the use of the test.
7.2.3 Case Study: Systems for Enumerating Context Free Grammars

Working in the domain of context free grammars, [Wharton 1977] begins with an enumerator that is extremely inefficient, in the sense that it enumerates many grammars that cannot possibly be solutions. Two further enumerators are presented in succession, each derived by reconfiguring its predecessor (by hand) to eliminate classes of grammar that need not be enumerated. In this case study\textsuperscript{2}, the enumerators in [Wharton 1977] are described in terms of the generative structure they impose on the set of context free grammars. The definition of a class of grammars that need not be enumerated is viewed as performance information.

[Wharton 1977] offers a unique opportunity to study how the difficulty of making effective use of performance information varies with different generative structures for the same set. In [Wharton 1977] some classes of grammar are eliminated very effectively after minor modifications to the original system whereas some other classes are eliminated effectively only after major modifications and other classes are eliminated only ineffectively, by testing individual grammars. Furthermore, different classes of grammar were difficult to eliminate effectively for different enumerators. Both phenomena can be explained in terms of the generative structures of the different enumerators.

In this study a grammar is defined to be an ordered set (i.e. tuple) of production rules of the form, \( L \Rightarrow R \), where \( L \), the left hand side, is a nonterminal symbol and \( R \), the right hand side, is a nonempty string of terminal and nonterminal symbols. Nonterminals are denoted \( N_1, N_2, \ldots, N_n \). There is assumed to be a fixed finite set of possible terminal symbols, TERMINALS. The following are considered distinct grammars.

\[
\text{GRAMMAR}_1: \\
\begin{align*}
N_1 &\Rightarrow N_2 N_3 \\
N_2 &\Rightarrow a \\
N_2 &\Rightarrow a a N_2 \\
N_3 &\Rightarrow b N_1 
\end{align*}
\]

\[
\text{GRAMMAR}_2: \\
\begin{align*}
N_1 &\Rightarrow N_2 N_3 \\
N_2 &\Rightarrow b N_1 \\
N_2 &\Rightarrow a \\
N_2 &\Rightarrow a a N_2
\end{align*}
\]

\textsuperscript{2} an early version of this case study is published in [Holte and Wharton 1986]
7.2.3.1 An Enumerator of Context Free Grammars

The first enumerator, GEN_1, imposes on the set of context free grammars the generative structure \([\text{UNION } \{<n,m,p>\}: \text{size class enumerator}]\), defined as follows. A size class is specified by a triple of non-negative integers. A grammar is in size class \(<n,m,p>\) if and only if it involves exactly \(n\) nonterminals, \(\{N_1,\ldots,N_n\}\), and contains exactly \(p\) productions the longest of which has a right hand side exactly \(m\) symbols long. \(\text{GRAMMAR}_1\) is in size class \(<3,3,4>\). For present purposes the enumerator of triples of non-negative integers may be regarded as primitive.

Associated with a given size class is \(\text{RHS-SET}\), the (finite) set of right hand sides of length \(m\) or less involving no nonterminals other than \(N_1,\ldots,N_n\). \(\text{RHS-SET}\) is enumerated by

\[
\text{UNION over length}\in\{1,\ldots,m\}
\quad \text{of} \quad (\text{ALL_TUPLES } \text{<length,TERMINALS}\cup\{N_1,\ldots,N_n\}>)
\]

The enumerator of size classes imposes on each the generative structure \([\text{UNION } \{<q_1,\ldots,q_n>\}: \text{shape class enumerator}]\), defined as follows. In size class \(<n,m,p>\) a shape class is specified by an \(n\)-tuple of positive integers, \(<q_1,q_2,\ldots,q_n>\), for which \(\Sigma q_i = p\). A grammar is in shape class \(<q_1,\ldots,q_n>\) if and only if it contains exactly \(q_i\) productions whose left hand side is \(N_i\) (\(1\leq i\leq n\)). \(\text{GRAMMAR}_1\) is in shape class \(<1,2,1>\). For present purposes the enumerator of \(\{<q_1,\ldots,q_n>\}\) may be regarded as primitive.

The enumerator of shape classes imposes on each shape class the generative structure \([\text{LR: LPS,RPS}]\), defined as follows. \(\text{LR}\) is a function of two arguments, a \(p\)-tuple \(<L_1,\ldots,L_p>\) of nonterminals and a \(p\)-tuple \(<R_1,\ldots,R_p>\) of elements of \(\text{RHS-SET}\). Given \(<L_1,\ldots,L_p>\) and \(<R_1,\ldots,R_p>\) \(\text{LR}\) produces the grammar

\[
\begin{align*}
L_1 & \Rightarrow R_1 \\
L_2 & \Rightarrow R_2 \\
\ldots \\
L_p & \Rightarrow R_p
\end{align*}
\]
Associated with LR are projection functions, \((\text{LHS} (\text{LR} <x,y>))=x\) and \((\text{RHS} (\text{LR} <x,y>))=y\). The LR-constituents of shape class \(H\) are \(\text{LPS}=(\text{LHS} h)\,h \in H\) and \(\text{RPS}=(\text{RHS} h)\,h \in H\). LPS is the set of all \(p\)-tuples of nonterminals in which, for all \(i\), \(N_i\) occurs \(q_i\) times. For present purposes the enumerator of LPS may be regarded as primitive. RPS is enumerated by \((\text{ALL\_TUPLES} <p,\text{RHS\_SET}>)^3\).

7.2.3.2 A Second Enumerator of Context Free Grammars

An enumerator, GEN_2, that imposes a different generative structure on the set of grammars enumerated by GEN_1 will now be described\(^4\). Both GEN_1 and GEN_2 impose the generative structure [UNION [\(<n,m,p>\): [UNION [\(<q_1,...,q_n>\): shape class enumerator]] on the set of context free grammars. In GEN_2 the enumerator of shape classes imposes on each the generative structure [UNION [production-tuple tuples]: merge class enumerator], defined as follows. There are \(n\) entries in a production-tuple tuple, one corresponding to each nonterminal. The \(i\)th entry is a \(q_i\)-tuple of production rules all of which have \(N_i\) as their left hand side. There is a total of \(\Sigma q_i=p\) productions in a production-tuple tuple. For example, in size class \(<2,2,4>\) and shape class \(<1,3>\) a typical production-tuple tuple is

\[
\langle N_1 \Rightarrow bN_2, \ N_1 \Rightarrow a, \ N_2 \Rightarrow a, \ N_2 \Rightarrow aN_1 \rangle
\]

Define PRODUCTIONS to be a function mapping a nonterminal \(N_i\) and a grammar \(G\) to the \(q_i\)-tuple of productions in \(G\) having \(N_i\) as their left hand side; the productions in (PRODUCTIONS \(N_i\ G\)) are ordered as in \(G\). For example,

\[
(\text{PRODUCTIONS } N_2 \text{ GRAMMAR}_1) = \langle N_2 \Rightarrow a, \ N_2 \Rightarrow aN_2 \rangle
\]

\(^3\) actually, RPS excludes right hand side \(p\)-tuples in which there is no right hand side of length \(m\). The means by which this requirement is enforced are not discussed in [Wharton 1977]. The additional generative structure involved is almost identical in all the enumerators and does not affect the conclusions or comparisons that follow.

\(^4\) The systems in [Wharton 1977] correspond to GEN_1, GEN_3, and GEN_4 in this chapter. The set enumerated by GEN_3 is very much smaller than that enumerated by GEN_1. GEN_2 has been constructed, for the purposes of comparison, by making minor modifications to GEN_3 so that it enumerates the same set as GEN_1.
Grammar $G$ is in the merge class corresponding to production-tuple tuple $<P_1, \ldots, P_n>$ if and only if $(\text{PRODUCTIONS} \ N_i, G) = P_i$ for all $i$. For example, the merge class corresponding to

$$<N_1 \Rightarrow bN_2, \ N_2 \Rightarrow a, \ N_2 \Rightarrow a, \ N_2 \Rightarrow aN_1>$$

consists of the grammars

$$\begin{align*}
N_1 & \Rightarrow bN_2 & N_2 & \Rightarrow a & N_2 & \Rightarrow a & N_2 & \Rightarrow a \\
N_2 & \Rightarrow a & N_1 & \Rightarrow bN_2 & N_2 & \Rightarrow a & N_2 & \Rightarrow a \\
N_2 & \Rightarrow a & N_2 & \Rightarrow a & N_1 & \Rightarrow bN_2 & N_2 & \Rightarrow aN_1 \\
N_2 & \Rightarrow aN_1 & N_2 & \Rightarrow aN_1 & N_2 & \Rightarrow aN_1 & N_1 & \Rightarrow bN_2
\end{align*}$$

The grammar

$$\begin{align*}
N_1 & \Rightarrow bN_2 \\
N_2 & \Rightarrow a \\
N_2 & \Rightarrow aN_1 \\
N_2 & \Rightarrow a
\end{align*}$$

is not in this merge class because the productions with left hand side $N_2$ do not occur in this grammar in the order specified in the production-tuple tuple. As a second example, GRAMMAR$_1$ and GRAMMAR$_2$ are in the same merge class.

By treating each production-tuple $P_i$ as a stack, a grammar in the merge class corresponding to production-tuple tuple $<P_1, \ldots, P_n>$ can be enumerated by popping all the productions, one at a time, off all the stacks. At any time, any nonempty stack can be popped; different choices create different grammars. More precisely, a merge class is enumerated by TUPLES given the following definitions.

$$\begin{align*}
\text{Spec} & = \text{a production-tuple tuple, } <Q_1, \ldots, Q_n>, \\
& \text{initially the specification } <P_1, \ldots, P_n> \text{ of the merge class}
\end{align*}$$

$$\begin{align*}
\text{END	extunderscore OF	extunderscore TUPLE} & = \text{for all } i, \ Q_i = <> \\
\text{EXPAND } <Q_1, \ldots, Q_n> & = \{\text{POP } i \mid Q_i \neq <>\} \\
\text{POP } i & = <\text{HEAD}(Q_i), \ Q_1, \ldots, Q_{i-1}, \text{TAIL}(Q_i), Q_{i+1}, \ldots, Q_n> \\
\text{HEAD } <t_1, \ldots, t_k> & = t_1 \\
\text{TAIL } <t_1, \ldots, t_k> & = <t_2, \ldots, t_k>
\end{align*}$$

The specifications produced by EXPAND indicate which production-tuple to pop. A top level constituent in the generative structure imposed by TUPLES on the merge class corresponding to $<P_1, \ldots, P_n>$ is specified by $<\text{HEAD}(P_i), \ <P_{i+1}, \ldots, P_n>, \text{TAIL}(P_i), P_{i+1}, \ldots, P_n>$, and
contains all grammars in the merge class whose first production is HEAD(P_i). There is a k_{th} level constituent corresponding to each sequence <a_1,...,a_i>, where a_i names the production-tuple to be popped to provide the i^{th} production in the grammar.

A merge class is specified by an production-tuple tuple. The set of specifications for the merge classes is therefore the set, PTT-SET_i, of distinct production-tuple tuples. PTT-SET is enumerated by TUPLES given the following definitions.\(^5\)

\[
\text{Spec} = \text{an integer } i, \text{ initially } i=1 \\
\text{END\_OF\_TUPLE} = i=n \\
\text{EXPAND } i = \{(LR <N_1,\ldots,N_i> R), \; i+1> \mid \text{Re (ALL\_TUPLES } <q_i,\text{RHS\_SET}>) \} \\
\]

where LR, defined earlier, is used to produce a q_i-tuple of productions all of which have N_i as their left hand side.

Every i-tuple of production-tuples, <P_1,...,P_i>, specifies a constituent in the generative structure imposed on PTT-SET by this enumerator. If P_i precedes P_{i'} in the enumeration of (EXPAND i) then every production-tuple tuple beginning <P_1,...,P_i> is enumerated before the first production-tuple tuple beginning <P_1,...,P_{i'}>. Therefore all grammars G in the same shape class having the same value of (PRODUCTIONS N_{a G}) for (1\leq a\leq i) will occur consecutively in GEN_2's enumeration sequence.

The generative structure imposed by (ALL\_TUPLES <q_i,\text{RHS\_SET}>) on the set of production-tuples, \{P_i\}=(EXPAND i), is such that all production-tuples beginning with the same k productions will occur consecutively in the enumeration of \{P_i\}. This additional structure results in GEN_2 enumerating consecutively all grammars G in the same shape class having the same value of (PRODUCTIONS N_{a G}) for (1\leq a\leq(i-1)) and whose values of (PRODUCTIONS N_i G) have the same first k entries.

\(^5\) analogous to RPS, PTT-SET actually excludes production-tuple tuples in which none of the productions have a right hand side of length m.
For example, the production-tuple tuples in size class <2,1,3> and shape class <2,1> would be enumerated by GEN_2 in the following sequence (assuming RHS-SET={(a,b,N₁,N₂)}).

\[
\begin{align*}
<N₁ \rightarrow a, N₁ \rightarrow a>, & \quad <N₂ \rightarrow a> \\
<N₁ \rightarrow a, N₁ \rightarrow a>, & \quad <N₂ \rightarrow b> \\
<N₁ \rightarrow a, N₁ \rightarrow a>, & \quad <N₂ \rightarrow N₁> \\
<N₁ \rightarrow a, N₁ \rightarrow a>, & \quad <N₂ \rightarrow N₂> \\
<N₁ \rightarrow a, N₁ \rightarrow b>, & \quad <N₂ \rightarrow a> \\
<N₁ \rightarrow a, N₁ \rightarrow b>, & \quad <N₂ \rightarrow b> \\
<N₁ \rightarrow a, N₁ \rightarrow b>, & \quad <N₂ \rightarrow N₁> \\
<N₁ \rightarrow a, N₁ \rightarrow b>, & \quad <N₂ \rightarrow N₂> \\
<N₁ \rightarrow a, N₁ \rightarrow N₁>, & \quad <N₂ \rightarrow a> \\
<N₁ \rightarrow a, N₁ \rightarrow N₁>, & \quad <N₂ \rightarrow b> \\
<N₁ \rightarrow a, N₁ \rightarrow N₁>, & \quad <N₂ \rightarrow N₁> \\
<N₁ \rightarrow a, N₁ \rightarrow N₁>, & \quad <N₂ \rightarrow N₂> \\
<N₁ \rightarrow a, N₁ \rightarrow N₂>, & \quad <N₂ \rightarrow a> \\
<N₁ \rightarrow a, N₁ \rightarrow N₂>, & \quad <N₂ \rightarrow b> \\
<N₁ \rightarrow a, N₁ \rightarrow N₂>, & \quad <N₂ \rightarrow N₁> \\
<N₁ \rightarrow a, N₁ \rightarrow N₂>, & \quad <N₂ \rightarrow N₂> \\
\end{align*}
\]

This pattern repeats once for each of the remaining possibilities for the first production, \(N₁ \rightarrow b\), \(N₁ \rightarrow N₁\), and \(N₁ \rightarrow N₂\).

That all the grammars having productions \(N₁ \rightarrow a\) and \(N₁ \rightarrow b\) (in that order) are enumerated consecutively is a property of the generative structure imposed by TUPLES on the set of production-tuple tuples, a property that is independent of the way in which (EXPAND i) is enumerated. The generative structure imposed by TUPLES on the set of production-tuple tuples does not guarantee the consecutive enumeration of all the grammars in which \(N₁ \rightarrow a\) is the first production with \(N₁\) as a left hand side; this occurs because TUPLES, in the form of ALL_TUPLES, is also used to enumerate (EXPAND i). A different enumeration of (EXPAND i) could result in an enumeration of PTT-SET that does not exhibit this pattern, such as one beginning...
7.2.3.3 A Comparison of Alternative Generative Structures

Included in the set of grammars produced by GEN_1 and GEN_2 are classes of grammars that may be deemed extraneous for one reason or another. For example, this set includes grammars that contain no terminal symbols, such as

\[ N_1 \Rightarrow N_2 \\
N_2 \Rightarrow N_1 \]

A definition of one of these extraneous classes is performance information in the form of a constraint, "do not consider grammars satisfying this definition". Given such information a system should reconfigure itself to eliminate the grammars in that class from the enumeration sequence. This constraint can always be used as a test of individual grammars but this is not an effective use of the constraint. The constraint can be used effectively if it can be applied earlier in the enumeration process. For example, it is sometimes possible to reconfigure a hierarchical system so that the constraint is used as a test of the specification of each constituent. This is much more effective than testing each individual element of each constituent. It is the generative structure imposed by a system that determines the how early in its enumeration process a constraint may be applied. Because the generative structures imposed by GEN_1 and GEN_2 are different, there are tests that can be applied early in one but not in the other. This is illustrated in the following with some of the classes of extraneous grammars defined in [Wharton 1977].
Eliminating Grammars Containing Duplicate Productions

GEN_1 and GEN_2 enumerate many grammars containing two or more copies of the same production rule. For example, they enumerate

\[ N_1 \Rightarrow bN_2 \]
\[ N_2 \Rightarrow a \]
\[ N_1 \Rightarrow a \]
\[ N_2 \Rightarrow aN_1 \]

Let NO_DUPLICATES be a function that is false of grammars containing duplicate copies of a production rule and true otherwise. Both GEN_1 and GEN_2 can be reconfigured to apply the NO_DUPLICATES to the output of the shape class enumerator. Reconfigured in this way they impose the generative structure \([\text{UNION} [{n,m,p}]: [\text{UNION} [{q_1,\ldots,q_m}]: [\text{NO_DUPLICATES}: \text{shape class enumerator}]]]\).

This is not an effective use of the NO_DUPLICATES definition: it is being applied to each individual grammar. Whether there is a more effective use depends on the generative structure imposed by the shape class enumerator.

GEN_1's shape class enumerator imposes the generative structure \([\text{LR}:\text{LPS},\text{RPS}]\). Because NO_DUPLICATES cannot be applied to elements of LPS or RPS, it cannot be used in GEN_1 except to test each individual grammar. GEN_2's shape class enumerator imposes the generative structure \([\text{UNION} [\text{PTT-SET}]: \text{merge class enumerator}]\), in which PTT-SET has the generative structure

\([\text{UNION} \{P_i\}:
\quad ((\text{CONS} P_i) :
\quad [\text{UNION} \{P_j\}:
\quad \quad ((\text{CONS} P_j) : \{<>\} ) \ldots )))\]

where \(\{P_i\}\) denotes \((\text{EXPAND} i)\). An element of \(\{P_i\}\) specifies the tuple of productions with left hand side \(N_i\) that occurs in all grammars in the constituent corresponding to \(\{P_i\}\). In GEN_2, NO_DUPLICATES can be used in an effective way, as a test of each \(\{P_i\}\).
Reconfigured in this way PTT-SET has the generative structure

\[
\text{[UNION \{NO\_DUPLICATES: \{P_i\}\}:}
\]
\[
\text{\{(CONS \ P_i):}
\]
\[
\text{[UNION \{NO\_DUPLICATES: \{P_i\}\}:
\]
\[
\text{\ldots [UNION \{NO\_DUPLICATES: \{P_n\}\}:
\]
\[
\text{\{(CONS \ P_n): \{<>\}\}]]\ldots]]}
\]

The next section will discuss how NO\_DUPLICATES may be applied earlier still, during the process of enumerating the elements of \{P_i\}.

Eliminating Grammars With Identical Sets of Productions

For every multiset containing \(p\) production rules, GEN\_1 and GEN\_2 enumerate all \(p!\) ways of ordering these productions. For example, GRAMMAR\_1 and GRAMMAR\_2 are different orderings of the same multiset of productions. For most purposes, only one of the orderings need be enumerated. \cite{Wharton 1977} orders the elements of RHS-SET, \(RHS_1, RHS_2, \ldots, RHS_{|RHS-SET|}\), and requires production \(N_i \Rightarrow RHS_x\) to occur before production \(N_k \Rightarrow RHS_y\) if and only if \(i \leq k\) and, when \(i=k\), \(x < y\). The two conjuncts in this definition can be restated as separate tests of grammars. LHS-ORDER is a function that is true of grammar \(G\) if and only if the productions in \(G\) whose left hand side is \(N_{i+1}\) occur before those whose left hand side is \(N_i\) (for \(2 \leq i \leq n\)). RHS-ORDER is a function that is true of grammar \(G\) if and only if the right hand sides of productions with the same left hand side occur in the required order. By requiring "\(x < y\)" rather than "\(x \leq y\)", RHS-ORDER will be false on all grammars with duplicate productions.

LHS-ORDER can be used effectively in both GEN\_1 and GEN\_2. In GEN\_1 LHS-ORDER can be applied to the elements of LPS; doing so eliminates all of them but one. Because \[\text{[LHS-ORDER:LPS]}\] has only one element, \(X\), the generative structure \[\text{[LR: [LHS-ORDER:LPS],RPS]}\] is equivalent to \[\text{[LR': RPS]}\] where \((LR' Y) = (LR X Y)\).
In GEN_2 LHS-ORDER is applied to constituents of merge classes in much the same way as it is applied to LPS in GEN_1. A merge class has one constituent corresponding to each sequence \( <a_1, ... , a_k> \), where \( a_i \) names the production-tuple to be popped to provide the \( i \)th production in the grammar. LHS-ORDER can be applied to each \( <a_1, ... , a_k> \), eliminating all except those such that \( a_i = b \) if and only if \( q_1 + ... + q_{b-1} < i \leq q_i + ... + q_b \). LHS-ORDER eliminates all sequences of length \( p \) but one. In other words, LHS-ORDER specifies precisely which production-tuple to pop at each step. Having applied LHS-ORDER, there is but one grammar in each merge class, i.e., one grammar corresponding to each production-tuple. Because LHS-ORDER requires all productions whose left hand side is \( N_i \) to occur before those whose left hand side is \( N_{i+1} \), the grammar corresponding to production-tuple \( P \) is (FLATTEN \( P \)), where

\[
\text{FLATTEN } <A_1, ... , A_z>, <B_1, ... , B_z>, ..., <Z_1, ... , Z_z> \\
= <A_1, ... , A_z>, B_1, ... , B_z, ..., Z_1, ... , Z_z >
\]

e.g. \( \text{FLATTEN } <a, b>, <1, 2> = <a, b, 1, 2> \)

The generative structure \([\text{UNION } \{P\text{T-TSET}\}: \text{LHS-ORDER: merge class enumerator}]\) is therefore equivalent to \([\text{FLATTEN}; \text{PTT-SET}]\). When GEN_2 is reconfigured with LHS-ORDER, grammars in the same size and shape class that are identical in the first \( k \) productions occur consecutively in the enumeration sequence. Furthermore, this set of grammars is a constituent in this generative structure. This is exploited in section 7.2.3.4.

In GEN_1, RHS-ORDER cannot be applied to the elements of LPS or RPS, and therefore it can only be used as a test of individual grammars. In GEN_2, RHS-ORDER can be applied, like NO-DUPLICATES, to the elements of each \( \{P_i\} \). \( \{P_i\} \) has imposed on it by \( \text{(ALL_TUPLES }<q_i, R\text{HS-SET}>\) the generative structure

\[
\{\text{UNION } \{<z_1, \text{RHS-SET}>\}:
\quad [(\text{CONS } z_1)]:
\quad \{\text{UNION } \{<z_2, \text{RHS-SET}>\}:
\quad \quad [(\text{CONS } z_2)]:
\quad \quad \quad ... [(\text{CONS } z_q):[<>]]...]]\}
\]
where \( r_k \) specifies the right hand side for the \( k^{th} \) production in the elements of \( \{P_i\} \). RHS-ORDER can be applied to each \(<r_k, \text{RHS-SET}>\), eliminating \(<r_k, \text{RHS-SET}>\) unless \( r_k > r_{k-1} \).

Figure 7.2 illustrates the effect of reconfiguring GEN_2 in this way on the generative structure imposed on \( \{P_i\} \). The figure assumes \( q = 4 \) and that only ten right hand sides are possible in the current size and shape classes. Under these assumptions 4-tuples of decimal digits correspond to the elements of \( \{P_i\} \). For example, if \( i = 1 \) and the right hand sides are

<table>
<thead>
<tr>
<th>RHS-SET</th>
<th>a</th>
<th>N_1</th>
<th>aN_1</th>
<th>aa</th>
<th>bb</th>
<th>b</th>
<th>N_1b</th>
<th>bN_1</th>
<th>ab</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

the number 2356 corresponds to

**GRAMMAR_3**

\[ N_1 \Rightarrow aN_1 \]
\[ N_1 \Rightarrow aa \]
\[ N_1 \Rightarrow b \]
\[ N_1 \Rightarrow N_1b \]

Only a segment of the enumeration sequence of \( \{P_i\} \) is shown. Preceding this segment are the numbers whose first digit is 1, and succeeding this segment are the numbers whose first digit is 4 or greater. Elements are enumerated sequentially from left to right and top to bottom (i.e. the last five elements shown are 3678, 3679, 3689, 3789, 4567).
A top level constituent of \([P_1]\) contains all 4-tuples whose first entry has a particular value. A k\(^{th}\) level constituent contains all 4-tuples whose first k entries have a particular combination of values. 3\(^{rd}\) level constituents are depicted in the figure as a line of 4-tuples; 2\(^{nd}\) level constituents are depicted as a group of consecutive lines; and top level constituents are depicted as blocks of consecutive groups.

There are 55 4-tuples in the two top level constituents shown in the figure. Before reconfiguring GEN_2 with the RHS-ORDER test there were 2000 4-tuples in these constituents. Each of the eliminated 4-tuples contains duplicate entries or has the same entries, in a different order, as one of the 55 that remain.
Eliminating Other Classes of Extraneous Grammars

Several other classes of extraneous grammars are defined in [Wharton 1977]. In GEN_1, all of the definitions must be used as tests applied to individual grammars. In GEN_2, some can be applied to the specifications of constituents of \( \{P_i\} \); others can only be applied to individual grammars. The tests associated with some of these classes of grammars are

**Positive-Example-Compatibility:** is true of a grammar if and only if it accepts all the positive examples.

**Negative-Example-Compatibility:** is true of a grammar if and only if it accepts none of the negative examples.

**Sufficient-Terminals:** is true of a grammar if and only if each terminal symbol that occurs in a positive example occurs in the right hand side of a production in the grammar. For example, this test eliminates the following grammar as soon as one positive example is known.

\[
N_1 \Rightarrow N_2 \\
N_2 \Rightarrow N_1
\]

**Ambiguity-Heuristic#2:** it is undecidable to determine if an arbitrary context free grammar is ambiguous, but some classes of grammar are certain to be ambiguous. One such class (the second defined in [Wharton 1977]) contains grammars in which there occur a left recursive production, \( N_1 \Rightarrow N_0 \alpha \), and a right recursive production, \( N_1 \Rightarrow \beta N_1 \), with the same left hand side (\( \alpha \) and \( \beta \) are strings of terminals and nonterminals). For example, GRAMMAR_3 contains the left recursive production \( N_1 \Rightarrow N_1 b \) and the right recursive production \( N_1 \Rightarrow a N_1 \). Ambiguity-Heuristic#2 is true of a grammar if and only if the grammar is not this class. This test is heuristic because some ambiguous grammars pass it.

**Connected:** is true of a grammar if and only if all of its nonterminals are reachable, where a nonterminal is **reachable** if it is the initial nonterminal or it is in the right hand
side of a production whose left hand side is already known to be reachable. If \( N_1 \) is the initial nonterminal, the following grammar is disconnected because \( N_2 \) is unreachable.

\[
\begin{align*}
N_1 & \Rightarrow b \\
N_1 & \Rightarrow bbN_1 \\
N_2 & \Rightarrow bbbN_1
\end{align*}
\]

The second system described in [Wharton 1977], GEN_3, is GEN_2 reconfigured with tests associated with all the definitions of extraneous classes of grammars. The NO_DUPLICATES and RHS-ORDER tests are applied as described above; all others are applied to individual grammars. The generative structure imposed by GEN_3 on the set of context free grammars is

\[
\text{[TESTS: [UNION \{<n,m,p>: <\text{UNION \{<q_1,\ldots,q_n>: \text{FLATTEN:PTT-SET}\}\}\}\}]}
\]

TESTS refers to all tests except NO_DUPLICATES and RHS-ORDER. PTT-SET has the structure

\[
\text{[UNION \{P_1\}:}
\text{[CON P_1]:}
\text{[UNION \{P_2\}:}
\text{... [UNION \{P_1\}:}
\text{[CON P_1]:}
\text{... [UNION \{P_n\}:}
\text{[CON P_n]: [\{<\}\] ...] ...]]}
\]

and each \( \{P_i\} \) has the structure

\[
\text{[UNION \{<r_1,\text{RHS-SET}>\}:}
\text{[CON r_1]:}
\text{[UNION \{RHS-ORDER:\{<r_2,\text{RHS-SET}>\}:}
\text{[CON r_2]:}
\text{... [UNION \{RHS-ORDER:\{<r_k,\text{RHS-SET}>\}:}
\text{[CON r_k]:}
\text{... [UNION \{RHS-ORDER:\{<r_q,\text{RHS-SET}>\}:}
\text{[CON r_q]: [\{<\}\] ...] ...]]]]}
\]

where \( r_k \) specifies the right hand side for the \( k \)th production in the elements of \( \{P_i\} \).
In the final enumerator described in [Wharton 1977], GEN_4, every test produces as output the index of the production rule at which the outcome of the test (pass or fail) becomes known. This information is called a failure point or success point depending on the outcome of the test. For example, if "ab" is a negative example, GRAMMAR_3

\[
N_1 \Rightarrow aN_1 \\
N_1 \Rightarrow a a \\
N_1 \Rightarrow b \\
N_1 \Rightarrow N_1 b
\]

fails the Negative-Example-Compatibility test because it accepts a negative example. Its failure point for this test is 3, because the outcome is established after examining the first three productions of the grammar.

The information that arises as feedback from tests is comparable in every way to the performance information provided by external sources. Therefore the opportunities and obstacles for improving efficiency through the use of feedback are the same as those for improving efficiency through the use of performance information. In particular, adding feedback to a system will improve its efficiency only if its generative structure permits effective use to be made of the the information provided by feedback. GEN_4 is simply GEN_3 with the feedback of failure/success point information added. GEN_4 is considerably more efficient than GEN_3\(^6\) because the generative structure imposed by GEN_3 permits this information to be used effectively, as will now be described.

\(^6\) Empirical results on p. 271 of [Wharton 1977] suggest that the difference in efficiency between GEN_4 and GEN_3 is even greater than the difference between GEN_3 and GEN_1.
The Use of Failure Point Information

The failure points of some tests have the property that if grammar G fails with failure point f all grammars identical to G in the first f productions are guaranteed to fail. The failure points of the Ambiguity-Heuristic#2 and Negative-Example-Compatibility tests have this property. Given failure point information with this property a system could reconfigure itself by adding the test "is the grammar currently being enumerated identical to G in the first f productions?". If such a test can be applied early enough in the enumeration process, this may be an effective way to use the failure point information.

The failure points of other tests have the weaker property that all grammars identical to G in the first f productions are guaranteed to fail only if they are also in the same shape and size class as G; grammars in other size and shape classes that are identical to G in the first f productions may or may not fail these tests. For example, a grammar with p+1 productions can pass the Sufficient-Terminals test even if the grammar consisting of its first p productions fails. The same is true of the Positive-Example-Compatibility and Connected tests. Given failure point information with the weaker property a system could reconfigure itself by adding the test "is the grammar currently being enumerated identical to G in the first f productions and in the same size and shape class as G?".

GEN_4 treats every failure point as if it had only the weaker property. If grammar G fails a test with failure point f, GEN_4 creates the new test TESTf, "is the grammar currently being enumerated identical to G in the first f productions and in the same size and shape class as G?". If f=q_1+q_2+...+q_{a+1}+b, TESTf can be inserted in the generative structure of \(P_a\) around [RHS-ORDER:{<r_b,RHS-SET>}], i.e.,
Because G is the most recently enumerated grammar, the most recently enumerated element of [RHS-ORDER:{<r_b,RHS-SET>}] in \{P_a\} is the specification of the first f productions in G. This specification will fail TESTf and a new element of this [RHS-ORDER:{<r_b,RHS-SET>}] will be enumerated. Thus, an immediate consequence of inserting TESTf into the generative structure is that the very next grammar enumerated will differ from G in the first f productions. The grammars in the enumeration sequence between G and this grammar are all identical to G in the first f productions and are all eliminated immediately upon inserting TESTf. In fact, in [RHS-ORDER:{<r_b,RHS-SET>}] of \{P_a\} of the current size and shape class all the elements pass TESTf except the one specifying the first f productions in G. Since this one is eliminated immediately upon inserting TESTf, there is no reason for TESTf to remain. Therefore, GEN_4 applies TESTf once only, to determine where to resume enumeration, and then discards it. In other words, GEN_4 uses failure point information to leap forward in the enumeration sequence across all the grammars in the current size and shape class that are identical to the present grammar up to its failure point.

For example, having enumerated GRAMMAR_3 and received a failure point of 3, GEN_4 would resume its enumeration with the first subsequent grammar that differed from GRAMMAR_3 in the first 3 productions, or in size or shape. Figure 7.3 illustrates the generative structure imposed by GEN_4 on the shape class containing GRAMMAR_3, assuming that RHS-SET is

<table>
<thead>
<tr>
<th>RHS-SET</th>
<th>a</th>
<th>N_1</th>
<th>aN_1</th>
<th>aa</th>
<th>bb</th>
<th>b</th>
<th>N_1</th>
<th>bN_1</th>
<th>ab</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- 140 -
The arrows indicate the leaps forward in the enumeration sequence made by GEN_4 in response to different failure points. Different types of arrow correspond to different failure points: "I" corresponds to a failure point of 1, "\n" to 2, and the dotted arrow to 3. Arrows corresponding to a failure point of 4 join every 4-tuple to its immediate successor and have not been shown. An arrow of type k from X to Y indicates that all 4-tuples between X and Y (excluding Y) are identical to X in the first k positions. Because the 4-tuples are tested in sequence, if an arrow of type k connects X to Y no arrow of type k can originate from a 4-tuple between X and Y.
GRAMMAR$_3$ corresponds to number 2356. Receiving a failure point of 3, GEN$_4$ follows the dotted arrow leading from 2356 and enumerates the grammar 2367, i.e.,

\[
N_1 \Rightarrow aN_1 \\
N_1 \Rightarrow a a \\
N_1 \Rightarrow N_1 b \\
N_1 \Rightarrow b N_1
\]

bypassing all the grammars that differ from GRAMMAR$_3$ only in the fourth production.

Figure 7.3 gives an indication of the number of grammars eliminated from the enumeration sequence by this use of failure point feedback. In the worst case, no grammars are eliminated because the next grammar in the sequence is not identical to the present grammar up to its failure point. In GEN$_4$ this happens when the failure point is the last production in a grammar (a failure point of 4 in the figure). For some tests, such as Sufficient_Terminals, the failure point is always the last production in the grammar; in GEN$_4$ feedback from such tests is of no use. However, for many tests the failure point varies from grammar to grammar, and considerable gains in efficiency result from GEN$_4$'s use of the failure point information.

The Use of Success Point Information

Success point information cannot be used to eliminate grammars from the enumeration sequence, but it can be used to reduce the computational effort involved in testing a sequence of grammars. On the basis of success point information GEN$_4$ reconfigures itself to eliminate unnecessary tests (ones that are certain to succeed) in an exact parallel to the elimination of unnecessary grammars on the basis of failure point information. An arrow of type k from X to Y in Figure 7.3 indicates that Y is the next grammar after X that differs from X in the first k productions. Therefore, if grammar X passes test TESTs with success point k, TESTs need not be applied to any grammar between X and Y. If k=$q_1+q_2+\ldots+q_{a-1}+b$, TESTs can be removed from its present position in the generative structure and inserted, as TESTI was, in the generative structure of (P$_a$) around [RHS-
ORDER: [{r_b,RHS-SET}], i.e.,

[UNION {<r_1,RHS-SET>}]:
  [(CONS r_1):
      [UNION [RHS-ORDER:{<r_2,RHS-SET>}]]]:

  ...

  [UNION TESTS[{{RHS-ORDER:{<r_k,RHS-SET>}}]}]:
  [(CONS r_k):

  ...

  [UNION [RHS-ORDER:{<r_q,RHS-SET>}]]]:
  [(CONS r_q):[<>]][...][...]]]]

Because G is the most recently enumerated grammar, the mostly recently enumerated element of this [RHS-ORDER:{<r_b,RHS-SET>}] is the specification of the first k productions in G. G having passed TESTs with success point k, this specification must also pass TESTs. But TESTs will not be applied again until the next element of this [RHS-ORDER:{<r_b,RHS-SET>}] is enumerated; that is, TESTs will not be applied again until all the grammars in the current size and shape class that are identical to G in the first k productions have been enumerated.

The Use of Other Feedback Information

A failure (success) point of f for grammar G on test T provides the information "an outcome of failure (success) for test T is guaranteed by the property that a grammar is in the same size and shape class as G and its first f productions fail (pass) T". A general notion of feedback is that it provides information of the form "outcome X of test T is guaranteed by outcome Y of test U". Such feedback will result in efficiency gains only if U can be applied earlier in the generative structure than T. A general technique for using feedback effectively is to remove T and insert U until outcome Y fails to obtain, at which time T is restored.

Whether or not U can be applied earlier than T depends on the generative structure of the system. In the generative structure of GEN_4, the test U corresponding to a failure/success point often could be applied earlier than the test T that produced the failure/success point. Thus GEN_4's generative structure permits the effective use of failure/success point information. Its generative structure does not permit some other types of information to be
used effectively. For example, instead of failure point information indicating that some change is necessary in the first production, several of the tests in GEN_4 could describe precisely what changes are needed in the first production. The Sufficient-Terminals test could indicate the set MTS of terminal symbols that are missing; the Connected test could indicate the set MNTS of nonterminal symbols that are unreachable; and the Ambiguity-Heuristic#2 test could indicate whether it is right recursive or left recursive productions that are inadmissible. As useful as this additional information may seem, in GEN_4 it cannot be used any more effectively than failure point information alone.

The information that the failure point of a grammar on a test is f gives rise to tests that are applied to the right hand side of the fth production. For the additional information to improve on this, it must give rise to tests that can be applied even earlier, that is, inside the enumerator of right hand sides. In GEN_4 this enumerator has the generative structure

\[
\text{[UNION } k \in \{1, \ldots, m\]: }
\text{[UNION } \{\text{TNT}_1\}:
\text{[(CONS } \text{TNT}_1):
\text{[UNION}\{\text{TNT}_2\}:
\ldots
\text{[(CONS } \text{TNT}_k):\{<>\} \ldots]]}
\]

where \( \{\text{TNT}_i\} = \text{TERRNALS} \cup \{N_1, \ldots, N_n\} \)

That is, a length \(k\) is specified for the right hand side, and right hand sides of that length are produced one symbol at a time, left to right. A test for left recursive productions can be applied early in this generative structure, but the tests corresponding to the other feedback information cannot be applied earlier than the selection of the final symbol in the right hand side; i.e., they can only be applied to entire right hand sides.

The following enumerator of right hand sides imposes a generative structure in which it is possible to make effective use of the sets MTS and MNTS that can be provided by the Sufficient-Terminals and Connected tests. After a length \(k\) is chosen sets \(\text{ST} \subseteq \text{TERRNALS}\) and \(\text{SN} \subseteq \{N_1, \ldots, N_n\}\) are chosen so that \(|\text{ST}| + |\text{SN}| \leq k\). A right hand side satisfies the specification \(<k,\text{ST} \cup \text{SN} \rangle\) if it is length \(k\), contains every symbol in \(\text{ST} \cup \text{SN}\), and contains
only symbols in ST∪SN. This set can be enumerated, for example, by TUPLES given the definitions,

\[ \text{Spec} = \text{a pair } <\text{Length, Set}> \text{ containing an integer and a set,} \]
\[ \text{initially } \text{Length} = k \text{ and } \text{Set} = \text{ST∪SN} \]
\[ \text{END_OF_TUPLE } <\text{Length, Set}> = (\text{Length}=0) \]
\[ \text{EXPAND } <\text{Length, Set}> = \begin{cases} & \text{if } \text{Length}=|\text{Set}| \\ & \text{then } [<s, <\text{Length-1, Set}/s>> \text{ if } s \in \text{Set} ] \\ & \text{else } [<s, <\text{Length-1, Set}>] \text{ if } s \in \text{Set} ] \end{cases} \]

where S/s = S with the element s removed

In this enumerator the test that the right hand side of the i\textsuperscript{th} production includes all the symbols in MTS can be applied to the specification ST and the test that the right hand side includes all the symbols in MNTS can be applied to the specification SN.

7.2.4 Techniques For Effectively Using Information in Enumerative Systems

The efficiency of an enumerative system depends equally on two factors: the information that is available, and the effectiveness with which available information is used. The case study illustrates the influence of both factors. GEN_4 is more efficient than GEN_3 only because the feedback from the tests provides useful information to GEN_4 that is not available to GEN_3. On the other hand, although the same information is available to GEN_1 and GEN_2, GEN_2 is more efficient because it makes more effective use of the information. Techniques of analysis, such as credit assignment and explanation-based generalization, may be regarded as techniques for amplifying information. Although these techniques are normally associated with other types of system, it is clear from the case study that the information they provide can be used effectively in enumerative systems\(^7\). The following is a list of some of the techniques by which an enumerative system may make effective use of information, whatever its origin.

\[ \text{{\footnotesize \cite{Mostow and Bhatnagar 1987}} uses explanation-based generalization to amplify failure point information in an enumerative system} \]
Early Testing

It is not an effective use of information to apply it as as a test of individual candidates. However it not always necessary to wait until a candidate is fully formed to apply a test. The outcome of a test depends on certain properties of a candidate. In the process of enumerating a candidate some properties become definite long before the candidate is fully formed. A test may be applied as soon as the properties upon which its outcome depends have become definite. If the generative structure permits a test to be applied sufficiently early, the test will constitute an effective use of information. Early testing is well illustrated in the case study.

Test Insertion or Deletion

Test insertion is the primary reconfiguration technique used in the case study. Its main effect is to remove candidates or subspaces, but it also can change the point at which enumeration resumes and, by moving a test to an earlier point in the enumeration process, reduce the number of applications of a test. Test deletion, on its own, has the effect of increasing the number of candidates enumerated.

Modifying a Constant

Constant values occur in various places in enumerators. For example, the constant \(<X>\) occurs in every enumerator based on TUPLES. TUPLES can be reconfigured to enumerate all tuples ending with X by changing this constant to \(<X>\). Similarly, an enumerator of ST (or SN) that starts with the empty set \(\{\} \) and adds terminal (nonterminal) symbols one at a time can be reconfigured to enumerate supersets of MTS (or MNTS) by changing the starting value from the constant \(\{\} \) to MTS (MNTS). The number of sets enumerated increases as the cardinality of MTS (MNTS) decreases.

As a different example, some enumerators explicitly store a resumption point and proceed by applying a successor function to the current value of the resumption point. By changing
this value an enumerator of this type can be reconfigured to resume enumeration from any
point. An enumerator of this type for the set $\langle r_k, \text{RHS-SET} \rangle$ in production-tuple $P_i$ could
be reconfigured to incorporate the RHS-ORDER requirement by setting its resumption point
to the current value of $\langle r_{k-1}, \text{RHS-SET} \rangle$.

Reordering Constituents

Reordering the constituents of $[\text{UNION } a \in A : [\text{UNION } b \in B : F(a,b)]]$ to
$[\text{UNION } b \in B : [\text{UNION } a \in A : F(a,b)]]$ has two main effects. First, it changes the sequence in
which $\{F(a,b)\}$ is enumerated. Second, it reverses the order in which $a \in A$ and $b \in B$ are
determined, with the result that tests applied to $B$ occur earlier in the enumeration process
after the reordering. For example, testing whether a production is right recursive can only
be done after the rightmost symbol in the production's right hand side has been determined.
The enumerators of right hand sides described above have a constituent corresponding to
each position in the right hand side. These are organized in such a way that the rightmost
symbol is the last to be determined. Consequently the test for right recursiveness cannot be
applied until all the symbols in a right hand side have been determined. Reordering the
constituents so that the rightmost symbol is determined first permits this test to be applied
much earlier, to specifications of large sets of right hand sides rather than individual right
hand sides.

Test Compilation

There may exist more efficient ways of enumerating the set $\{s \mid \text{TEST}(s) \& s \in S\}$ than actually
generating and individually testing all the elements of $S$. For example, if $\text{TEST}(s) = \text{"is } s = k?"$
for constant $k$, the set $\{s \mid \text{TEST}(s) \& s \in S\}$ is simply $\{k\}$, a set that can be enumerated very
efficiently. The LHS-ORDER test is of this form. In GEN_1 this permitted $[\text{LR:[LHS-}
\text{ORDER:LPS}, \text{RPS}]$ to be compiled first to $[\text{LR:[X]}, \text{RPS}]$ and then to $[\text{LR:\text{RPS}}]$, where $X$
is the one element of LPS that passes LHS-ORDER. Applied to $q_n$ the test "is
$(q_1 + q_2 + \ldots + q_n) = p$?" is also of this form: only the value $q_n = p \land (q_1 + \ldots + q_{n-1})$ passes this test.
A second type of test compilation is possible if the test TEST₁ is applied to elements of a set \( \{ F(x) \mid x \in X \} \) and there exists a second test, TEST₂, such that \( \text{TEST}_1(F(x)) \equiv \text{TEST}_2(x) \) for all \( x \in X \). Under these conditions the generative structures \([\text{TEST}_1,F:X]\) and \([F:\text{TEST}_2:X]\) are equivalent, but not necessarily equally efficient: for example, in \([F:\text{TEST}_2:X]\) \( F \) is calculated only when necessary. To insert early in the enumeration process a test defined in terms of candidates usually requires a compilation of this type. For example, reconfiguring \([\text{LHS-ORDER}:[\text{LR}:\text{LPS},\text{RPS}]] \) to \([\text{LR}:[\text{LHS-ORDER}:\text{LPS}],\text{RPS}] \) is a compilation of LHS-ORDER. The two "LHS-ORDER" tests have different domains: the first is a test of grammars, the second a test of p-tuples of nonterminals.

A more complicated type of test compilation is possible if \( \text{TEST}(s) = \"\text{Is REL}(s,h) \text{ true for some } h \in H?\" \) (where REL is any relation) and there exists an efficiently computable function \( F \) such that \( F(h)=\{s|\text{REL}(s,h) \text{ is true}\} \). Under these conditions \([\text{TEST}:S]\) can be compiled to \([\text{UNION H:F}]\). A simple example of this type of compilation is the enumeration of squares of integers. Instead of enumerating integers (S) and testing if they are squares one may enumerate integers (H) and square them (F in this case produces just one \( s \in S \) for each \( h \in H \)). A more complex example, whose details are given in the next section, involves the test "does grammar \( s \) accept positive example \( p \)?". This test can be formulated "does there exist a unlabelled parse tree for \( p \) that, when labelled, corresponds to \( s \)?". Here \( S=\{\text{grammars}\}, \ H=\{\text{unlabelled parse trees for } p\} \), and \( \text{REL}(s,h) \) is true if any of the labellings of \( h \) corresponds to \( s \). The required function \( F \) does exist: all the unlabelled parse trees of a string can be enumerated efficiently, as can all the labellings of a parse tree involving a given set of nonterminals.
Large-Scale Restructuring

Unlike all the previous techniques, the third type of test compilation eradicates the enumeration sequence and generative structure of the original system S and installs in its place [UNION H:F]. The changes in generative structure brought about by test compilation can be arbitrarily complex. For example, test compilation might be expected, at least in principle, to produce GEN_2 given [RHS-ORDER:GEN_1]. [Dietterich and Bennett 1986] formulates the general problem of reconfiguring an enumerative system as the problem of test compilation.

The automation of reconfiguration techniques that result in large-scale restructuring has been explored in only a few experimental systems. [Tappel 1980] and [Dietterich and Bennett 1986] are initial studies of test compilation. [Mostow 1981] and [Keller 1987] are initial studies of the general problem of "operationalizing" advice in the context of problem-solving and learning systems, respectively. A recent workshop\(^8\) introduced the problem of enumerator reconfiguration to researchers in the fields of automatic programming and program transformation, but no concrete results of this cross-fertilization have yet appeared.

\(^8\) [Dietterich 1986]
7.3 General Discussion

7.3.1 The Relationship Between Enumerative and Other Types of System

we close this report with a proposal that we consider particularly compelling - the development of a procedure which can construct precisely that grammar which an enumerative method would discover. Such a procedure would be far removed from existing constructive methods since, by its ability to construct the same grammars that an enumerative technique would enumerate, it would retain all the theoretical advantages that follow from the use of enumerative procedures. The concept behind such a procedure suggests a quantum leap in intuition, for it implies discarding the distinction, which so far we have consistently maintained, between constructive and enumerative inference methods. Although a development of this kind would be a major departure from previous work, we believe it is entirely within the range of possibility.\footnote{p.163, [Wharton 1973]}

It has long been held that enumerative systems are fundamentally different than (and distinctly inferior to) other types of system. Typical of this sentiment is the statement in [Biermann and Feldman 1972a] (p. 42),

\begin{quote}
In contrast to the exhaustive search methods [enumerative algorithms] which produce optimum performance often at the cost of astronomical computational effort, a few constructive inference procedures have been developed, which produce useful if not optimal grammars in a reasonable length of time
\end{quote}

The preceding section has laid to rest the idea that an enumerative system must rigidly follow a fixed enumeration sequence. Reconfiguration techniques can change the resumption point, the set of candidates that are enumerated, and the order in which they are enumerated. Furthermore these changes can often be implemented in a highly efficient manner. It is only between applications of reconfiguration operators that an enumerative system follows a fixed enumeration sequence: this sequence is best regarded as a default that will be followed until performance information dictates otherwise. To examine in detail the relationship between enumerative and other types of system, the reconfiguration techniques will be compared with the techniques described in section 7.1.2.
The same test compilation techniques apply to both enumerative and other types of system. The technique of corepresenting a subspace and a candidate corresponds to the technique of testing a constituent by applying the test to the first candidate in the constituent and reconfiguring if the test fails (e.g. failure points). In component-by-component testing a test of set \( \{C_1, \ldots, C_n\} \) is applied, in different forms perhaps, to some or all of the components, either in parallel, as in \( ([T_1;C_1]x \ldots x[T_k;C_k]x \ldots x[T_k;C_k]) \), or sequentially, as in \( \text{UNION} c_i e [T_1;C_i] \text{CONS } c_i \text{UNION } c_2 e [T_2;C_2] \ldots [\text{CONS } c_k : (<>)] \ldots ] \). If \( [T_i;C_i] \) contains only one element the system is transitional between enumerative and constructive. For example, BOXES selects its components in parallel; each component is selected by enumerating and testing all its possible values. ID3 selects attributes sequentially for nodes at different levels in the decision tree and in parallel for nodes at the same level. The attribute for each node is selected by enumerating all the possibilities.

Broadly speaking, the class of constructive induction techniques corresponds to the class of large-scale restructuring techniques. Certainly the aim of both classes is the same: to accelerate learning by altering the structure of the set of candidates. Some techniques, such as constructing new terms, apply equally well to enumerative and other types of system.

Some systems using data-driven techniques can be transformed into enumerative systems by the type of test compilation in which \( [\text{TEST}_1;F:X] \) is reconfigured as \( F:[\text{TEST}_2;X] \). If \( F \) a function that maps a mediating structure \( (x \in X) \) to a candidate then \( [\text{TEST}_1;F:X] \) corresponds to a system that tests hypotheses whereas \( F:[\text{TEST}_2;X] \) corresponds to a system that "analyzes data" and constructs a hypothesis on the basis of the analysis. This is one of the ways in which RULEGEN and ID3 differ. RULEGEN extends the tree it is constructing in all possible ways and tests each of the resulting trees, whereas ID3 tests the possible ways of extending the tree before actually extending it. Systems that are related in this way are transitional between enumerative and data-driven.
The essential feature of data-driven techniques is that performance information (data) is used not as a test, however early, but as a partial specification of the hypothesis to be selected. For example it may specify the exact value of a component of the hypothesis, as in the updating of hypertables in section 6.2.2, or it may specify the region of candidate space in which to find the hypothesis, as in the "active set" technique. Several of the reconfiguration techniques described in the previous section use performance information directly as a specification for constituents. Modifying the value of a constant and converting a set to a constant (the first type of test compilation) are two "enumerative" techniques for using information directly as a specification.

The conclusion, then, is that data-driven techniques are applicable to enumerative systems. Like any reconfiguration technique, the possibility of applying a data-driven technique in an enumerative system is determined by the generative structure of the system. The enumerators in the case study have generative structures that prevent the effective use of information that can be derived from example data. The following is a brief description of an enumerative system whose generative structure permits the use of data-driven techniques.\(^{10}\)

In the generative structure of this system a set of parse trees specifies a constituent, namely, the set of grammars that can generate one or more of the parse trees in the given set. Roughly speaking, the system proceeds as follows. It first maps each positive example onto the set of possible parse trees of the example, thereby mapping the example onto the specification of a constituent. It then enumerates all the grammars in the intersection of the constituents specified by the positive examples. Grammars accepting negative examples are eliminated by explicit test.

More precisely, a reduced grammar for a set of strings is defined as any set of context free production rules that accepts the strings such that no subset of the production rules accepts the strings. The set of reduced grammars for a single string can be enumerated efficiently, by enumerating all the parse trees of the string and all the ways of labelling with

\(^{10}\) [van Lehn and Ball 1987] may be consulted for a full description of a very similar system and a discussion of the definitions and claims that follow.
nonterminals the arcs in each parse tree.

Two grammars, \( g \) and \( h \), involving distinct sets of nonterminals, \( \{V_1, \ldots, V_a\} \) and \( \{W_1, \ldots, W_b\} \), can be combined into a single grammar, \( G_{g,h} \), as follows.

(a) initially, let \( G_{g,h} \) be the union of the sets of productions in \( g \) and \( h \).

(b) replace all occurrences of \( W_1 \) in \( G_{g,h} \) with \( V_1 \).

(c) for each \( V_i \) (\( i > 1 \)):
   either do nothing,
   or choose some \( W_j \) and replace all occurrences of \( W_j \) in \( G_{g,h} \) with \( V_i \).

Different choices in step (c) result in different grammars. Define \( \text{JOIN}(g,h) \) to be the set of all grammars that can be produced by combining \( g \) and \( h \) in the manner just described. \( \text{JOIN}(g,h) \) can be enumerated efficiently, by enumerating the different choices in step (c). If \( g \) is a reduced grammar for \( S_g \) and \( h \) is a reduced grammar for \( S_h \), \( \text{JOIN}(g,h) \) includes all reduced grammars for \( S_{g \cup h} \) and possibly other grammars as well.

A system for enumerating the reduced grammars consistent with a set of examples could proceed using an adaptation of the STAR methodology [Michalski 1983], as follows.

1. Initially, let \( S \), the set of reduced grammars, be the singleton containing the empty grammar.
   Repeat 2 until every \( G \in S \) accepts all the positive examples.

2. Execute 3-6 separately for every \( G \in S \) that does not accept all the positive examples.

3. Choose \( p \), a positive example not accepted by \( G \).

4. Generate the reduced grammars for \( p \);
   eliminate those that accept a negative example.

5. For each reduced grammar \( R \):
   calculate the set \( JR = \text{JOIN}(G, R) \);
   eliminate from \( JR \) grammars that accept a negative example;
   eliminate \( R' \in JR \) if it is not a reduced grammar for the set of positive examples it accepts;
   add the grammars in \( JR \) to \( S \).

6. Remove \( G \) from \( S \).
7.3.2 Efficient Hypothesis Selection and the Structure of the Candidate Set

Notions of candidate set structure have occasionally been defined and used to analyze the efficiency of (non-enumerative) systems. To complete the discussion of the role of generative structure in the analysis of a system's efficiency, generative structure will be compared to existing notions of structure.

The most common notion of the structure of candidate space is based on the (semantic) general-to-specific partial ordering of candidates. Candidate X is a neighbour of candidate Y if Y is a minimal generalization of X. Figure 7.1 depicts the general-to-specific structure of the candidate set of the system described in section 7.1.1. A system imposes this structure on the set of candidates if it proceeds by generalizing and/or specializing candidates that have previously been considered. This structure has served as the basis for systems of most types, enumerative (RULEGEN), candidate set modification (Edinburgh Focussing), and hypothesis modification (SOL [Solomonoff 1975]).

There are systems, for example those in which all context free languages are candidates, that do not impose the general-to-specific structure. A universally applicable notion of structure may be defined in terms of syntactic operations on syntactic candidates: candidate X is a neighbour\(^{12}\) of candidate Y if there exists in the learning system an operation mapping X to Y. For example, in Figures 4.1 and 4.2 syntactic neighbours are depicted as adjacent points on the x-axis. If the system includes operations corresponding to (semantic) generalization and specialization, the syntactic structure will include the general-to-specific structure.

This syntactic notion of structure is closely related to generative structure. In the syntactic structure of an enumerative system candidate X is a neighbour of candidate Y if it is

\(^{11}\) van Lehn and Ball 1987 omit this step and call the resulting grammars "derivational" rather than reduced.

\(^{12}\) "neighbour" normally connotes symmetry, but in this definition X may be a neighbour of Y without Y being a
possible for $Y$ to be enumerated immediately after $X$. If an enumerative system does not employ large-scale restructuring techniques its syntactic structure can be inferred from its generative structure. For example, the arrows in Figure 7.3 depict the syntactic structure imposed by the enumerator of tuples of right hand sides.

[Rendell 1987] employs syntactic structure to analyze the efficiency of hypothesis selection in hypothesis modification systems. The hypothesis selection task is formulated as the task of finding a candidate that is a local optimum of the credibility surface, where the credibility of a candidate depends on the performance information that is available. Thus, the credibility surface changes as new information becomes available. The analysis is based on a hill-climbing model of the hypothesis selection process: determine all the neighbours of a candidate by applying all possible operators and proceed to the neighbour with greatest credibility. The process involves no constructive inductive operators. Based on this model it is clear that the "neighbour" relation, i.e. the syntactic structure, heavily influences which candidate is selected and the number of candidates examined in the course of selection.

The analysis in [Rendell 1987] suffers from two serious shortcomings. First, it does not account for the effects of strategies that might be employed to determine the operators to attempt and the order in which to attempt them. In particular, it does not account for the effect of data-driven techniques. Yet such strategies and techniques can improve the efficiency of hypothesis selection dramatically. Secondly it fails to account for the nature of the performance information that is available. For example, it does not anticipate the phenomenon observed in section 7.1.3, that the constraint "no future hypotheses may involve q5 or r5" can be used more effectively in most systems than the constraint "no future hypothesis may be more specific than or equal to q2r2".

These shortcomings reflect inherent limitations in the analytical usefulness of syntactic structure. Syntactic structure is a relation between candidates, described directly in terms of candidates: $X$ and $Y$ are neighbours if $X$ immediately precedes $Y$ in some circumstances. Generative structure describes this relation in a much different way, in terms of an

 neighbour of $X$
organization of constituents (sets of candidates), specifications, and functions. The additional information provided by generative structure is precisely the information needed to overcome the limitations of syntactic structure as a basis for the analysis of efficiency.
CHAPTER 8

CONCLUSION

An analytical framework for learning systems must be applicable to a diverse set of systems and for these systems it must facilitate analytical activities such as

(1) analysis of the task-specific knowledge used by a system;

(2) analysis of the efficiency of specific systems and whole families of systems;

(3) transferring techniques and analyses from one system or family of systems to another;

(4) comparative analysis: analyzing or explaining the differences in behaviour of two (families of) systems in terms of differences in the (families of) systems.

To satisfy these requirements it has proved necessary to extend the analytical framework to include the semantic context of a learning system, that is, the relation between the system and the specific semantic domain and task to which the system is being applied (see Figure 8.1). By including the semantic context, representation and related concepts have a natural, indeed prominent, role in the analytical framework.

Certain specific analytical activities can be carried out entirely at the level of the "Hypothesis Selection Task". For example, the convergence characteristics of a system and its computational complexity depend only on the choice of candidate set and selection criterion, and not on the way in which they are represented. Similarly some hypothesis selection techniques, such as the use of an active subset of candidate set, can be expressed at this level. These specific analytical activities are thus facilitated for any system that can be described in terms of a candidate set, selection criterion, and performance history.
The problem of applying the framework to a wide variety of systems is treated as a problem of identifying the different ways of representing the components of the framework. Some general alternative methods of representation have been discussed, including the declarative representation of a function, and corepresentation, in which several components in the model are represented by a single component in the system. Alternative ways of representing the components of the "Hypothesis Selection Task" correspond to different architectures of learning systems. The relationship between the "Hypothesis Selection Task" model and the most common architectures has been described, establishing that the framework may be applied to most existing systems.

Knowing the relationship between the "Hypothesis Selection Task" model and the architectures provides useful guidance for the comparative analysis of, and the transfer of techniques and analyses between, systems having different architectures. These analytical activities are further facilitated by the general notions of transitional subclass and behaviour-preserving transformation, and the specific transitional subclasses and transformations given in the Chapter 6. Using these techniques it has been established that analyses of the behaviour of systems with one architecture may be transferred to systems with different architectures.

All analytical activities are greatly facilitated by including in the framework concepts related to the representation of the semantic domain, that is, the relation between the semantic
domain and the system being analyzed. The choice of representation determines the task-specific knowledge available to the system, and the means by which it is used by the system. The choice also determines the efficiency and even the adequacy of the system for the task. Various properties of the system, particularly its generative structure, determine whether or not there exists a representation under which the system is performs the task efficiently. These insights are not entirely new, but in the present framework they are stated with greater generality and precision, and are demonstrated more clearly, than previously.

In conclusion, an analytical framework for learning systems must include concepts related to representation and the general semantic context of the systems. This is necessary for some analytical activities, and of great benefit to others. An analytical framework that consists of a related set of sub-frameworks - the different levels in Figure 8.1 - enables analyses and discussions to be carried out in one sub-framework and their results applied in another. This should be of great assistance to the practitioners of machine learning whose thinking, at present, is largely confined to the syntactic ("systems") level.

**Future Directions**

The analytical framework presented is, of course, incomplete in many ways, and the list of desirable extensions and refinements is correspondingly long. One of the intriguing questions that is raised but not addressed by the current framework is, under what conditions does there exist a representation well-suited to two or more tasks? This phenomenon was observed twice in the thesis: in the discussion of constructive systems (Chapter 5), and in the examples of transforming analogical to mediating systems (Chapter 6).

Incomplete as it is, the analytical framework could be applied usefully to several current issues. For example, notions related to "constructive induction", "bias", and "change of representation" are currently formulated imprecisely and at the syntactic level. Investigations and discussions are considerably impeded by both the imprecision of these
formulations and the inappropriateness of the syntactic level for these notions.

Although the analytical framework has been developed to facilitate analytical activities, it happens to facilitate, to some extent, the activity of designing a learning system that efficiently solves a given problem in a semantic domain. The different levels in Figure 8.1 correspond to different stages of the design process. The analytical framework identifies analyses and representation techniques that can be applied at each level, and stresses the importance of semantic design decisions, such as the choice of candidate set and the formulation of the selection criterion. The conclusions of Chapter 7 suggest that an efficient learning system can be designed by first designing a generative structure that enables the effective use of the type of performance information that is likely to become available. The transformations that are used for comparative analysis serve equally well as tools for design: they can be used to transform an inefficient case-based system into an efficient mediating system. Transformation approaches to design are advocated in [Korf 1980] and [Jonckers 1986].

The analytical framework is not restricted to tasks in which performance information takes the form of classified examples. It may be applied to a broad range of "advice taking" tasks including ones bearing little resemblance to learning tasks. For example, in a design task the performance information may take the form of a critique of the present design (hypothesis) or of a specific refinement of the present design. An efficient design system is one that can use this information effectively. [Tong 1986] and [Araya and Mittal 1987] approach design tasks in this way. The present analytical framework can be easily extended to tasks in which the advice is subject to sudden change, for example, tasks in which a human expert is exploring a large space of alternatives and provides speculative advice about the quality of subspaces. [Holte et al. 1986] is an initial attempt to apply the framework to the task of scheduling a semiconductor fabrication plant. By unifying well-studied learning tasks with a range of advice-taking tasks the analytical framework permits the techniques and insights developed by the machine learning community to be transferred to tasks with which AI has little direct experience.
REFERENCES FOR LEARNING SYSTEMS CITED BY NAME

AGAPE — [Vrain 1986, Kodratoff and Ganascia 1986]

BOXES — [Michie and Chambers 1968]


Edinburgh Focussing — [Bundy et al. 1985]

ID3 — [Quinlan 1982, Quinlan 1983, Quinlan 1986]

INDUCE — [Michalski 1980]

LEX — [Mitchell et al. 1983]

METADENDRAL — [Lindsay et al. 1980], [Buchanan et al. 1976]

METALEX — [Keller 1987]

PLS — [Rendell 1983]

RULEGEN — [Lindsay et al. 1980], [Buchanan et al. 1976]

SPARC — [Dietterich and Michalski 1986]
REFERENCES

[Aleksander and Stonham 1979]

[Amarel 1968]

[Anderson 1986]

[Andreae 1977]

[Angluin 1982]

[Angluin and Smith 1983]

[Araya and Mittal 1987]

[Barto and Sutton 1981]

[Bauer 1979]

[Biermann and Feldman 1972]

[Biermann and Feldman 1972a]

[Binner 1977]
[Buchanan et al. 1976]  

[Buchanan et al. 1977]  

[Bundy 1982]  

[Bundy 1984]  

[Bundy et al. 1985]  

[Campbell 1960]  

[Carbonell et al. 1983]  

[Carbonell 1986]  

[Chomsky 1965]  

[Connell and Brady 1987]  

[Derthick 1984]  

[Dieterich and Bennett 1986]  

[Dieterich and Michalski 1986]  
[Dietterich and Michalski 1981]

[Dietterich et al. 1982]

[Dietterich 1986]
Dietterich, T. G. (editor), Proceedings of the Workshop on Knowledge Compilation, Computer Science Department, Oregon State University, 1986.

[Flann and Dietterich 1986]

[Gallant 1986]

[Gams and Lavrac 1987]

[Garey and Johnson 1979]

[Georgeff and Wallace 1984]

[Gilmore 1986]

[Gold 1967]

[Haralick 1978]

[Haralick and Kartus 1978]

[Haussler 1987]
[Hedrick 1976]

[Holland 1975]

[Holte 1985]

[Holte 1986]

[Holte and Wharton 1986]

[Holte et al. 1986]

[Holte and MacDonald 1987]

[Hopcroft and Ullman 1979]

[Jonckers 1986]

[Kearns et al. 1987]

[Keller 1987]

[Kibler and Aha 1987]

[King 1987]

- 165 -
[Kodratoff and Ganascia 1986]

[Koffman 1968]

[Kolodner et al. 1985]

[Korf 1980]

[Laird et al. 1986]

[Langley 1982]

[Langley 1983]

[Langley et al. 1987]

[Lenat and Brown 1984]

[Lindsay et al. 1980]

[Manago and Kodratoff 1987]

[Marr 1982]

[Michalski 1980]

[Michalski 1983]
[Michalski and Stepp 1983]

[Michalski 1987]

[Michie and Chambers 1968]

[Minsky and Papert 1969]

[Mitchell 1978]

[Mitchell 1980]

[Mitchell 1982]

[Mitchell 1983]

[Mitchell et al. 1983]

[Mitchell et al. 1986]

[Mostow 1981]

[Mostow and Bhatarag 1987]

[Murray and Elcock 1968]


[Schlimmer 1987]

[Shapiro 1983]

[Shapiro 1981]

[Simon and Lea 1974]

[Smith et al. 1977]

[Solomonoff 1975]

[Tappel 1980]

[Thornton 1987]

[Tong 1986]

[Utgoff 1986]

[Valiant 1984]

[Vrain 1986]

[Waterman 1970]
[Waterman 1975]

[Wharton 1973]

[Wharton 1977]

[Winston 1975]

[Winston 1980]

[Wolff 1982]

[van Lehn and Garlick 1987]

[van Lehn and Ball 1987]

[van Someren 1986]

[van Someren 1987]

- 170 -
INDEX OF DEFINED TERMS

active set .......................................................... 65
adequate representation .......................................... 43
admissible presentations .......................................... 6
alternative representations ....................................... 31
Ambiguity-Heuristic#2 ............................................. 136
analogy system ...................................................... 68
analytical framework .............................................. 2
applied, system to a task .......................................... 9
approximation in polynomial time ................................ 7
bias ........................................................................ 56
candidate set ........................................................ 12
candidates, of a learning task ..................................... 12
candidates, uninterpreted .......................................... 52
classes .................................................................. 9
classification task .................................................. 9
classifications ........................................................ 9
compatible, candidate .............................................. 12
composition functions, of a domain ......................... 23
concept .................................................................. 12
concept learning ..................................................... 4
concept learning, incremental ..................................... 18
Connected ............................................................ 136
constituent of a set .................................................. 119
constructive system ................................................ 69
COREP ................................................................. 33
corepresentation classes .......................................... 34
corepresented ........................................................ 33
corepresented, partially .......................................... 42
criterion of success ................................................ 7
data-driven ............................................................ 110
declarative form ...................................................... 42
declarative representation of functions ....................... 42
descriptive framework ............................................. 2
distributed representation ........................................ 40
domain ................................................................. 23
domain isomorphism ............................................... 25
domain relation ..................................................... 24
domain, semantic ................................................... 22
domain, syntactic ................................................... 22
effectiveness .......................................................... 115
etities, of a domain ............................................... 23
enumeration sequence ............................................ 63
example, classified ................................................ 12
example, seen ........................................................ 12
examples .................................................................. 9
fail ........................................................................ 119
failure point, of a grammar ....................................... 138
fidelity requirement ................................................ 28
full representation .................................................. 38
functions, of a domain ........................................... 24
generality, of a domain ........................................... 44
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>generative structure</td>
<td>118</td>
</tr>
<tr>
<td>grammar, reduced</td>
<td>152</td>
</tr>
<tr>
<td>hierarchical enumerator</td>
<td>119</td>
</tr>
<tr>
<td>hypertable</td>
<td>95</td>
</tr>
<tr>
<td>hypotheses, set of</td>
<td>6</td>
</tr>
<tr>
<td>hypothesis</td>
<td>12</td>
</tr>
<tr>
<td>hypothesis selection task</td>
<td>16</td>
</tr>
<tr>
<td>hypothesis selection task, sequential</td>
<td>18</td>
</tr>
<tr>
<td>identification in the limit</td>
<td>7</td>
</tr>
<tr>
<td>interpretation</td>
<td>29</td>
</tr>
<tr>
<td>interpreter</td>
<td>42</td>
</tr>
<tr>
<td>JOIN of two grammars</td>
<td>153</td>
</tr>
<tr>
<td>jointly represent</td>
<td>40</td>
</tr>
<tr>
<td>k-tail equivalence</td>
<td>86</td>
</tr>
<tr>
<td>literal representation</td>
<td>38</td>
</tr>
<tr>
<td>mediating structure</td>
<td>71</td>
</tr>
<tr>
<td>merge class</td>
<td>127</td>
</tr>
<tr>
<td>merged functions</td>
<td>27</td>
</tr>
<tr>
<td>merging system</td>
<td>67</td>
</tr>
<tr>
<td>minimal generalization</td>
<td>108</td>
</tr>
<tr>
<td>monotone decreasing</td>
<td>65</td>
</tr>
<tr>
<td>monotone increasing</td>
<td>65</td>
</tr>
<tr>
<td>Negative-Example-Compatibility</td>
<td>136</td>
</tr>
<tr>
<td>pass</td>
<td>119</td>
</tr>
<tr>
<td>perform</td>
<td>9</td>
</tr>
<tr>
<td>performance history</td>
<td>18</td>
</tr>
<tr>
<td>performance information</td>
<td>18</td>
</tr>
<tr>
<td>performance task</td>
<td>15</td>
</tr>
<tr>
<td>(PERFORMS S T)</td>
<td>9</td>
</tr>
<tr>
<td>Positive-Example-Compatibility</td>
<td>136</td>
</tr>
<tr>
<td>primitive enumerators</td>
<td>119</td>
</tr>
<tr>
<td>primitive functions, of a domain</td>
<td>23</td>
</tr>
<tr>
<td>PRODUCTIONS</td>
<td>126</td>
</tr>
<tr>
<td>production-tuple tuple</td>
<td>126</td>
</tr>
<tr>
<td>PTT-SET</td>
<td>128</td>
</tr>
<tr>
<td>reachable nonterminal symbol</td>
<td>136</td>
</tr>
<tr>
<td>reconfigurable enumerator</td>
<td>117</td>
</tr>
<tr>
<td>represent, jointly</td>
<td>40</td>
</tr>
<tr>
<td>representation, adequate</td>
<td>43</td>
</tr>
<tr>
<td>representation, distributed</td>
<td>40</td>
</tr>
<tr>
<td>representation, full</td>
<td>38</td>
</tr>
<tr>
<td>representation language</td>
<td>42</td>
</tr>
<tr>
<td>representation, literal</td>
<td>38</td>
</tr>
<tr>
<td>representations, alternative</td>
<td>31</td>
</tr>
<tr>
<td>represents</td>
<td>29</td>
</tr>
<tr>
<td>RHS-SET</td>
<td>125</td>
</tr>
<tr>
<td>selection criterion</td>
<td>12</td>
</tr>
<tr>
<td>semantic</td>
<td>9</td>
</tr>
<tr>
<td>semantic context</td>
<td>11</td>
</tr>
<tr>
<td>semantic domain</td>
<td>22</td>
</tr>
<tr>
<td>shape class</td>
<td>125</td>
</tr>
<tr>
<td>simple isomorphism</td>
<td>24</td>
</tr>
<tr>
<td>single representation trick</td>
<td>70</td>
</tr>
<tr>
<td>size class</td>
<td>125</td>
</tr>
</tbody>
</table>