EXPERIENCE-BASED LEARNING IN DEDUCTIVE REASONING SYSTEMS

by

Joongmin Choi

A dissertation
submitted to the Faculty of the Graduate School
of State University of New York at Buffalo
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

June 1993

Abstract

General knowledge is widely applicable, but relatively slow to apply to any particular situation. Specific knowledge can be used rapidly where it applies, but is only narrowly applicable. We present an automatic scheme to migrate general knowledge to specific knowledge during reasoning. This scheme relies on a nested rule representation which retains the rule builder's intentions about which of the possible specializations of the rule will be most useful. If both general and specific knowledge is available and applicable, a system may be slowed down by trying to use the general knowledge as well as, or instead of, the specific knowledge. However, if general knowledge is purged from the system after migration, the system will lose the flexibility of being able to handle different situations. To retain the flexibility without paying the price in speed, a shadowing scheme is presented that prevents general knowledge from being used when specific knowledge migrated from it is available and applicable. The combination of knowledge migration and knowledge shadowing allows a deductive reasoning system to learn from and exploit previous experience. Experience is represented by the instance relationship between the general knowledge and the specific knowledge migrated from it.

We also present techniques for implementing efficient rules of inference in natural deduction systems by caching and recalling the history of rule activation steps that alleviate duplicate pattern matchings and binding conflict resolutions. To reduce the complexity of manipulating rule activation steps from exponential to polynomial, methods of distributing the information about rule activation steps are developed that minimize the number of activation steps and the number of substitution compatibility tests among shared variables.

An implementation of these schemes in a network-based reasoning system is discussed. Test results are shown that demonstrate the predicted benefits of these ideas.

Acknowledgements

I would like to express many thanks to:

Dr. Stuart C. Shapiro, my advisor, for his support, guidance, criticism, and advice for a long period of time. His endless stream of new ideas always encourages me to think, challenge, and be creative. I've always been very proud of being his student.

Dr. William J. Rapaport, a member of my dissertation committee, for his constructive comments on my research during many research meetings, and for helping me to establish concrete concepts on cognitive and philosophical issues in artificial intelligence.

Dr. Bharat Jayaraman, a member of my dissertation committee, for his helpful comments on the formal definitions of logic-related terminologies. His comments during my dissertation defense made it quite a pleasant experience.

Dr. Valerie Shalin, the outside reader of my dissertation, for her valuable comments on machine learning systems.

SNePS Research Group (SNeRG) colleagues, for their efforts and patience in listening to my problems and suggesting good solutions.

Secretaries of the Computer Science Department, for their kindness, knowledge, and patience for many years which I will not ever forget.

And, most importantly, my family members: my wife Miae, my two sons Daniel (SungYong) and Christopher (SeYong), and my parents. I could never have completed this work without their constant support, love, and sacrifices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Role of Experience in Reasoning

Experience plays a major role in distinguishing an expert from a novice [Kolodner, 1983]. An expert is able to recall and exploit previous experience with already known or familiar situations, and consequently is able to to behave more intelligently and perform more effectively than non-experts in solving problems of certain domains. Roles of experience include refinement and modification of reasoning processes and knowledge, as well as guidance of later decision making and problem solving based on the analogy to previous cases [Kolodner and Kolodner, 1987].

The task of using experience in problem solving can be described in several different aspects. Experience does not simply mean the addition of acquired knowledge to memory (the knowledge base) from a series of problem solvings. The system may even be slowed down by adding new knowledge indiscriminately because of increased search space. What is more important for achieving system efficiency is how to apply and use experience appropriately in subsequent reasoning.

Experience-based problem solving is particularly effective for a problem domain that retains the property of what we call *locality in reasoning*. Locality in reasoning is a phenomenon that the same or similar kind of reasoning is repeated in a particular domain of application. For instance, consider a plan-action system for the blocks world. All actions in this system are decomposed into a series of primitive operations such as *pick up a block*, *put a block on top of another block*, and *put a block on the table*. The number of primitive operations is limited, and a complex operation is built by combining several primitive operations. For example, a plan for achieving a goal *pile three blocks A, B, and C on the table* can be divided into a sequence of primitive operations such as *pick up the block C*, *put C on the table*, *pick up the block B*, *put B on top of C*, *pick up the block A*, and *put A on top of B*. It is easy to anticipate in this system that a small number of rules for primitive operations can be used repeatedly even in a single plan.

Another example of locality in reasoning can be found in the domain of fault diagnosis for electronic devices. The primary objective of fault diagnosis is to check if a device is malfunctioning, and, if it is, to determine which components of the device went wrong. An interesting heuristic in this application is that the components that were diagnosed as faulty in previous testings are more likely to be faulty again, and therefore it is often a good idea to investigate these components first when the same device is being diagnosed later. Thus, a diagnostic rule for this particular component of the device can be repeatedly applied. Furthermore, if we diagnose a sequential circuit, a diagnostic rule for a component with feedback lines may be used as many times as clock cycles since the behavior of that component depends not only on the current input values, but also on the previous output values.

The intuition behind the concept of locality in reasoning is that, since every human expert is able to react efficiently for a situation which is very familiar to him, an expert reasoning system should also solve similar problems quickly.

1.2 Multi-Level Knowledge Structure

Obtaining better performance from experience relies on the capability of determining and applying better knowledge among a large amount of available knowledge. In general, it is not a simple task to establish clear-cut criteria for deciding which knowledge is better than other knowledge, since it depends on several factors including the application domain, the reasoning technique, the architecture of the system, the type of questions, the number of answers desired, and so on.

There are several levels of knowledge in terms of generality or specificity. This multi-level knowledge structure is often emphasized to achieve both system efficiency and generality. In other words, more specific knowledge is applied to known problems for system efficiency, and more general knowledge is applied to unfamiliar and difficult problems for generality.

It is interesting to observe that humans also use multi-level knowledge structure. Suppose an electrical engineer who has little experience in circuit fault diagnosis tries to locate a faulty component in a malfunctioning electrical circuit. This novice first uses general book knowledge such as Ohm's Law E=IR (E:voltage drop, I:current, R:resistance) that relates the output current with voltage drop. Ohm's Law is so general that it can be applied to cases with any voltage drop. However, in the course of locating a faulty component by using Ohm's Law, the engineer may acquire the *Voltage Short Rule* as follows [Milne, 1986]:

```
If one component connects the output and ground,
   and the output is zero,
then that component may be shorted.
```

This rule is more specific than Ohm's Law since it only considers a special case where the voltage drop is zero (short). If the engineer encounters a similar situation where the voltage short causes the device to be faulty, (s)he can simply consult the voltage short rule and not bother with Ohm's Law.

There seem to be two different paradigms in organizing multi-level knowledge structure based on two different definitions for knowledge hierarchy. The first paradigm focuses on the *depth* of knowledge, and is usually called the *deep* knowledge versus *shallow* knowledge distinction. This is an absolute distinction, meaning that a particular level of knowledge is said to be deep or shallow without necessarily comparing to other levels of knowledge. The second paradigm considers the *generality* of knowledge, and is usually called the *general* knowledge versus *specific* knowledge distinction. Unlike the first distinction, this is a relative concept in the sense that a particular level of knowledge is said to be *more general than* or *more specific than*, and hence must always be compared with other levels of knowledge.

1.3 Deep Knowledge versus Shallow Knowledge

The deep versus shallow knowledge distinction is first made in [Hart, 1982]. Deep knowledge has underlying representation of fundamental concepts such as causality or basic physical principles. In many cases, deep knowledge is represented by a model using information about structure, function, and behavior of the system [Davis, 1984, Davis and Hamscher, 1988, Genesereth, 1984, Hamscher and Davis, 1987]. Problem solving by using deep knowledge is called *model-based reasoning*. Model-based reasoning solves problems by reasoning about a model of the behavior of objects in the domain. Modeling provides a different kind of knowledge for reasoning in many domains. Model-based reasoning is generally slow because of longer inference chains.

On the other hand, shallow knowledge directly associates input states with actions, and mostly it is represented by a set of condition-action rules. Problem solving by using shallow knowledge is called *associational reasoning.* Associational reasoning uses heuristics, empirical associations, or rules of thumb, and has the advantage of efficiency. It solves common problems quickly by reducing long chains of inferences in the underlying deep knowledge to shorter links between data and solutions. However, this reasoning is unable to solve unanticipated, peripheral, or difficult problems, because it solves problems by matching the current situation against a set of predetermined situations.

As an example, look at a household electric buzzer system described in [Chandrasekaran *et al.*, 1985]. Deep knowledge is mainly represented by function and behavior. A function for "buzzing" can be represented as:

```
   FUNCTION: Buzz: TOMAKE buzzing(buzzer)  IF pressed (switch) by
                   behavior1
```

`Buzz` function says that if the switch is pressed, the buzzer goes to a state called `buzzing` that is accomplished by a series of behavioral states named `behavior1`. The representation of `behavior1` is:

```
BEHAVIOR: behavior1:
          Pressed(switch) BY behavior2
           {Clapper electrical connection alternates}
          USING-FUNCTION mechanical OF clapper
          Repeated-Hit(Clapper)
          USING-FUNCTION acoustical OF clapper
          Buzzing(Clapper)
          Buzzing(Buzzer)
```

`behavior1` says that the buzzer, if the switch is pressed, goes to a state where the electrical connections in the clapper alternately close and open, which results in the state where the clapper is repeatedly hit, which results in the buzzer being in the state of `buzzing`. From this information about the model of the buzzer, some shallow knowledge about how to diagnose the malfunction of the buzzer can be extracted and described by rules as:

```
R1:  If switch is pressed, but the clapper is not alternately
        electrically connected and disconnected,
     problem is in behavior2.

R2:  If switch is pressed, and
        the clapper's electrical connectivity alternates,
        but the clapper doesn't hit repeatedly,
     the cause of buzzer not buzzing is some mechanical
        malfunction of the clapper.
```

A multi-level knowledge system has been suggested to take advantages of both associational reasoning and model-based reasoning. Koton presented a method for overcoming the speed limitation of model-based reasoning by remembering a previous similar problem and making small changes to its solution [Koton, 1985, Koton, 1988, Koton, 1989]. This ends up with a reasoning system that uses associational reasoning for efficiency and uses model-based reasoning for robustness that can combine the advantages of both while complementing their individual limitations. In his system, model-based and associational reasoning is combined through the use of case-based reasoning, and rather than solving the entire problem solving using model based reasoning, the system uses the full power of the causal model only when needed.

Knowledge compilation is another research direction to bridge the gap between the two endpoints of the generality/efficiency spectrum. In a broad sense, knowledge compilation is the process of transforming some of the knowledge structures used by a given reasoning system in order to improve the system's run-time efficiency [Chandrasekaran and Mittal, 1983, Keller, 1990, Neves and Anderson, 1981]. Specifically in Anderson's ACT system [Anderson, 1989, Neves and Anderson, 1981], knowledge compilation transforms domain-general declarative knowledge to domain-specific procedural knowledge. When speed is needed, procedural knowledge is used, and when analysis or change is needed, declarative knowledge is used.

1.4 General Knowledge versus Specific Knowledge

The paradigm of distinguishing general knowledge from specific knowledge has been discussed mostly in rule-based systems. Since this is a relative distinction, categorizing the general level of knowledge is done by defining a *more general than* or *more specific than* relation. An implementation-independent definition for these relations can be described as below [Mitchell, 1982]:

> Rule $\mathbf{r}_1$ is more general than rule $\mathbf{r}_2$ (or $\mathbf{r}_2$ is more specific than $\mathbf{r}_1$), if in any world $\mathbf{r}_1$ can be used to show at least the same results as $\mathbf{r}_2$.

We have found three different but related approaches to rule generality in which the definition of *more general than* or *more specific than* relations are dependent upon specific application domains.

The first definition is used for conflict resolution in production systems [McDermott and Forgy, 1978, Sauers, 1988]. Conflict resolution in production systems is a method by which a rule based interpreter may select one of a set of applicable rules to be applied in some problem solving situation. One of the selection strategies for conflict resolution is the specificity strategy which prefers rules that test more specific features of the environment over rules which test more general features, and are meant to recognize special case relationships between rules. To do this, the relation *more specific than* is defined in [McDermott and Forgy, 1978] as follows:

> Rule $\mathbf{r}_1$ is more specific than rule $\mathbf{r}_2$ if
> (1) the two rules are not equal,
> (2) $\mathbf{r}_1$ has at least as many antecedent clauses as $\mathbf{r}_2$, and
> (3) for each antecedent clause in $\mathbf{r}_2$, with constant elements $\mathbf{c}_1$, $\mathbf{c}_2$, ..., $\mathbf{c}_n$, there exists a corresponding antecedent in $\mathbf{r}_1$, with constant elements $\mathbf{c}'_1$, $\mathbf{c}'_2$, ..., $\mathbf{c}'_m$, such that $\{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n\}$ is a subset of $\{\mathbf{c}'_1, \mathbf{c}'_2, \ldots, \mathbf{c}'_m\}$.

For example, $A(a,b,z) \wedge B(c,w) \wedge C(x) \rightarrow G$ is more specific than $A(a,b,z) \wedge B(c,w) \rightarrow G$ according to (2), which is also more specific than $A(x,b,z) \wedge B(v,w) \rightarrow G$ according to (3), where v, w, x, z are variables.

The second definition of rule generality is used in concept learning where a general description of a class of objects is derived from a set of examples and non-examples [Mitchell, 1982]. In this work, each instance is described by an unordered pair of feature vectors, each of which specifies the size, color, and shape of an object. For example, {(Large Red Square) (Small Yellow Circle)} is an instance. Generalizations of these instances are represented in a similar way, except that some of the properties are replaced by question marks indicating that they are unimportant. For example, {(Small ? Circle) (Large ? ?)} is a generalization representing the set of all instances containing one small circle and one large object. The matching predicate M(g,i) is defined for a generalization g and an instance i. Here, g matches i if and only if there is a mapping from the pair of feature vectors of g onto the pair of feature vectors of i such that either (1) the feature restriction in g is identical to the feature value in i, or (2) the feature restriction in g is a question mark. For example, the above example generalization {(Small ? Circle) (Large ? ?)} matches the example instance {(Large Red Square) (Small Yellow Circle)}. Here, the *more specific than* relation is defined as follows.

> Given two generalizations, G1 and G2, G1 is more specific than G2 if and only if $\{i \in I \mid M(G1, i)\} \subseteq \{i \in I \mid M(G2, i)\}$, where I is the set of all instances, and M is the matching predicate.

In other words, G1 is more specific than G2 if and only if G1 matches a proper subset of the instances that G2 matches. For example, both {(Large Red Circle) (Large ? ?)} and {(? ? Circle) (Large Blue ?)} are more specific than {(? ? Circle) (Large ? ?)}. This *more specific than* relation imposes a partial ordering in the generalization hierarchy, and generalization is characterized as a search problem.

The third definition of rule generality is in the domain of logic programs in which *more general than* is equivalent to *subsumes*, and *more specific than* is equivalent to *is subsumed by.* In the θ-subsumption theory [Plotkin, 1970], clause C is more general than clause D if there exists a substitution θ that satisfies D $\supseteq$ Cθ. In this theory, the subsumption relation between two clauses is determined without consulting any other clause. For example, $Q(x) \rightarrow P(x)$ subsumes $Q(x) \wedge R(x) \rightarrow P(x)$ since the former must succeed for any value of x for which the latter does. In [Buntine, 1988], the inadequacy of θ-subsumption is discussed, and a generalized subsumption is suggested.

> Clause C subsumes (or is more general than) clause D with respect to logic program P if for any Herbrand interpretation I such that P is true in I, and for any atom A, C covers A in I whenever D covers A. Here, C is a generalization of D, and D is a specialization of C.

Here, $S(x) \rightarrow P(x)$ subsumes $Q(x) \wedge R(x) \rightarrow P(x)$ with respect to $Q(x) \rightarrow S(x)$. In this method, rules can be organized into a structure called a *specialization* (*generalization*) *hierarchy.* Induction can be achieved by searching through those clauses more general than a known specialization of a clause, for instance, a ground fact. Alternatively, a search can be made of those of clauses more specific than a known generalization.

We are interested in generating a specific rule from a general rule during a deduction. Some of variables will be replaced by constants, and the specificity relationship between these two rules is determined based on this substitution. Therefore, our definition of rule generality is based on a substitution that unifies two rules.

> Rule $\mathbf{r}_1$ is more specific than rule $\mathbf{r}_2$ if there exists a non-empty substitution $\sigma = \{c1/v1, c2/v2, \ldots\}$ such that $\mathbf{r}_1 = \mathbf{r}_2\ \sigma$. Here, v1, v2, ... are variables, and c1, c2, ... are constants.

Note that σ is non-empty, meaning that two rules should not be the same. This is obvious since the same rule will never be generated from itself. Also σ does not contain a binding v1/v2 where both v1 and v2 are variables. Details are in Chapter 3.

1.5 Knowledge Redundancy

The multi-level knowledge structure is closely related to the issue of knowledge redundancy in which the existence of both general and specific knowledge causes the same conclusion to be reached by more than one way.

One of the issues that arises in applying knowledge representation/ reasoning systems to real problems is the prevention of performance degradation caused by the addition of a large amount of redundant knowledge in the course of reasoning. In automated reasoning, redundant knowledge has been considered a main culprit of performance degradation [Wos, 1987] from the fact that it increases the search space by providing more alternatives during an inference, while the same goal can still be reached without it. Hence, research in this paradigm has focused on preventing redundant knowledge from being asserted during inference [Wos, 1987] or eliminating it after inference [Markovitch and Scott, 1988].

However, we claim that recognizing and avoiding knowledge redundancy is not a simple task, and often more complicated than doing reasoning itself. There is no easy way of determining whether a given knowledge base is redundant, since that is known to be undecidable [Greiner, 1991]. Furthermore, redundant knowledge may become necessary in most real reasoning systems with limited resources, because there may be a goal that can only be reached by using redundant knowledge that requires fewer resources [Buntine, 1988]. From these observations, instead of attempting to remove redundancy, we tackle the issue of improving deduction efficiency in a redundant knowledge base by using techniques of extracting experience information from redundant knowledge and applying it to future problem solving.

Our motivation for improving deduction efficiency by producing and using redundant knowledge can be explained by the following two examples.

First, consider an example knowledge base consisting of a rule $\forall$x [`man`(x) $\rightarrow$ `mortal`(x)] and a fact `man`(Socrates). The question `p`(Socrates), for variable `p`, asked to this knowledge base is answered by deriving a new fact `mortal`(Socrates), and the addition of `mortal`(Socrates) to the knowledge base causes fact redundancy. Then, the same question `p`(Socrates) asked in a subsequent deduction can be answered more quickly if we can block the rule from the activation and directly retrieve `mortal`(Socrates), since the rule is not deriving any new result other than `mortal`(Socrates) which is already in the knowledge base. So, even though `mortal`(Socrates) is redundant, it is certainly useful.

The second example considers rule redundancy. Suppose a knowledge base contains a generic rule for the transitive relation $\mathtt{r}_g$ = $\forall$r [`trans`(r) $\rightarrow$ $\forall$x,y,z [`r`(x,y) $\wedge$ `r`(y,z) $\rightarrow$ `r`(x,z)]] and facts `trans`(on), `on`(a,b), `on`(b,c), and `on`(c,d). A natural deduction derivation of `on`(a,c) from this knowledge base generates a more specific transitive rule $\mathtt{r}_s$ = $\forall$x,y,z [`on`(x,y) $\wedge$ `on`(y,z) $\rightarrow$ `on`(x,z)] as an intermediate result by applying universal instantiation and modus ponens.

This specific rule creates redundancy since the next query on(b,d) can now be solved in two ways, either by using the generic transitive rule $\mathtt{r}_g$ or by using the specific on transitive rule $\mathtt{r}_s$. If the system is intelligent enough to select $\mathtt{r}_s$ over $\mathtt{r}_g$, this particular query on(b,d) can be answered more quickly since $\mathtt{r}_s$ produces fewer inference steps than $\mathtt{r}_g$ does. On the other hand, for a more general query r(b,d) for variable r, $\mathtt{r}_g$ should be preferred to $\mathtt{r}_s$, since the query is looking for all possible relations between b and d, not restricted to the on relation. The question now is how to make a choice systematically according to the current situation.

1.6 Approach

In this section, we describe our approach to achieving system efficiency in a multi-level knowledge structure by using the general vs. specific knowledge paradigm and to avoiding performance degradation caused from knowledge redundancy.

Our objective is to develop deductive learning mechanisms to control deductive inference by using past problem-solving experience and by avoiding duplicate deduction steps for similar problems, and eventually to improve problem solving time in subsequent inferences. Generally, we are interested in a system where experience monotonically adds knowledge to the knowledge base. In this environment, the general issue would be how maximal advantage can be taken of old partial results in solving new problems and how the regeneration of partial results can be avoided when solving new problems. We are also concerned with achieving both generality and system efficiency by providing systematic decision-making procedures to select the proper level of knowledge.

Redundant knowledge that is produced during a deduction is treated as a piece of experience, and the information about how the redundant knowledge was generated is represented and stored in the *expertise base*. Learning and deduction are integrated so that the expertise information produced through learning can be recalled and exploited during future deduction to solve similar problems more efficiently.

The task of using experience to allow similar problem to be solved more efficiently is tackled at two different levels, one at the *rule selection* level and the other at the *rule activation* level.[1]

Learning at the rule selection level focuses on developing techniques of deciding the best possible branch in a deduction tree by choosing an appropriate rule among many applicable rules. A selection criterion is the specificity relationship among rules that is obtained from previous reasoning. Two learning modules, *knowledge migration* and *knowledge shadowing*, are provided for experience-based rule selection [Choi and Shapiro, 1991a, Choi and Shapiro, 1991b].

Knowledge migration, formally defined in Section 3.3, is a process of acquiring specific knowledge from general knowledge during a deduction. During this process, redundant knowledge is generated, and experiential information is stored in the expertise base. In rule-based systems, migrated knowledge is a rule that is a partial instantiation of a general rule with one or more of its variables replaced by ground terms. In the previous transitive rule example, $\mathbf{r}_s$ is said to be migrated from $\mathbf{r}_g$ in the course of answering `on`(a,c). Experience is represented by relative specificity relationships between the general and the migrated knowledge, and is important to construct rule selection control knowledge that is used in future deductions.

Knowledge shadowing, formally defined in Section 3.4, recognizes redundancy, recalls the experiential information from the expertise base and compares it with the current situation, and blocks those rules that are determined to be unnecessary. Decision-making about which rule must be blocked and which rule must be used depends on the content of the expertise base and the nature of the current query. A general rule is activated for a general query, and a specific rule is activated for a specific query. In the transitive rule example, $\mathbf{r}_g$ would be blocked for the query `on`(b,d), while $\mathbf{r}_g$ would be blocked for the query `r`(b,d).

Learning at the rule activation level considers the issue of fast execution of selected rules by caching and recalling the history of rule activation steps. This also corresponds to the problem of efficient implementation of various rules of inference in natural deduction systems. In real applications, rules are often complicated, containing many antecedents and many shared

[1] By 'rule', we mean the factual-level rule that is stored in the knowledge base and that consists of antecedents and consequents. In contrast, a meta-level rule such as modus ponens will be called an 'inference rule' or 'rule of inference'. So 'rule activation' indicates an attempt to derive the consequent of a factual-level rule by satisfying its antecedents.

variables, and as a result activating rules of inference can be as costly as the selection of proper rules. In most natural deduction systems, activating a rule of inference involves a number of pattern matchings and a number of testings for resolving binding conflicts of shared variables. It is also possible that the same rule could be executed several times when similar deductions are repeated. From this observation, a special data structure named "rule use information" (RUI) has been used to exploit the information about rule activation history [Hull, 1986]. Rule activation steps are saved in the RUI set structure of each rule and recalled the next time to reduce the duplicate rule activation jobs. A RUI set has been implemented in the SNePS knowledge representation and reasoning system by using a single linear list for each rule, but the processing of this linear RUI set is intractable due to the exponential complexity to process the set as the number of antecedents in a rule and the number of instances of the antecedents increase. To achieve real performance enhancements, we propose two algorithms **S-indexing** and **P-tree** that reduce the complexity of processing the RUI set to polynomial time by distributing information over several places and resolving only necessary instances [Choi, 1990, Choi and Shapiro, 1992].

1.7 System Overview

In general, a deductive reasoning system consists of two major components; the knowledge base containing facts and rules, and the inference engine performing reasoning about knowledge. In our system, called SNIP2.2 (SNePS Inference Package version 2.2), the inference engine is augmented with a number of learning components that accumulate and make use of expertise information during a deduction. SNIP2.2 consists of three components: (1) storage, (2) reasoning, and (3) learning as shown in Figure 1.1.

The storage component includes the knowledge base, the expertise base, and the rule activation cache. The knowledge base maintains rules and facts. The expertise base stores deduction expertise generated during reasoning. This experiential information includes the instance set information and the origin set information that are described in Chapter 3. The rule activation cache contains the history of rule activation steps of each rule that is also a

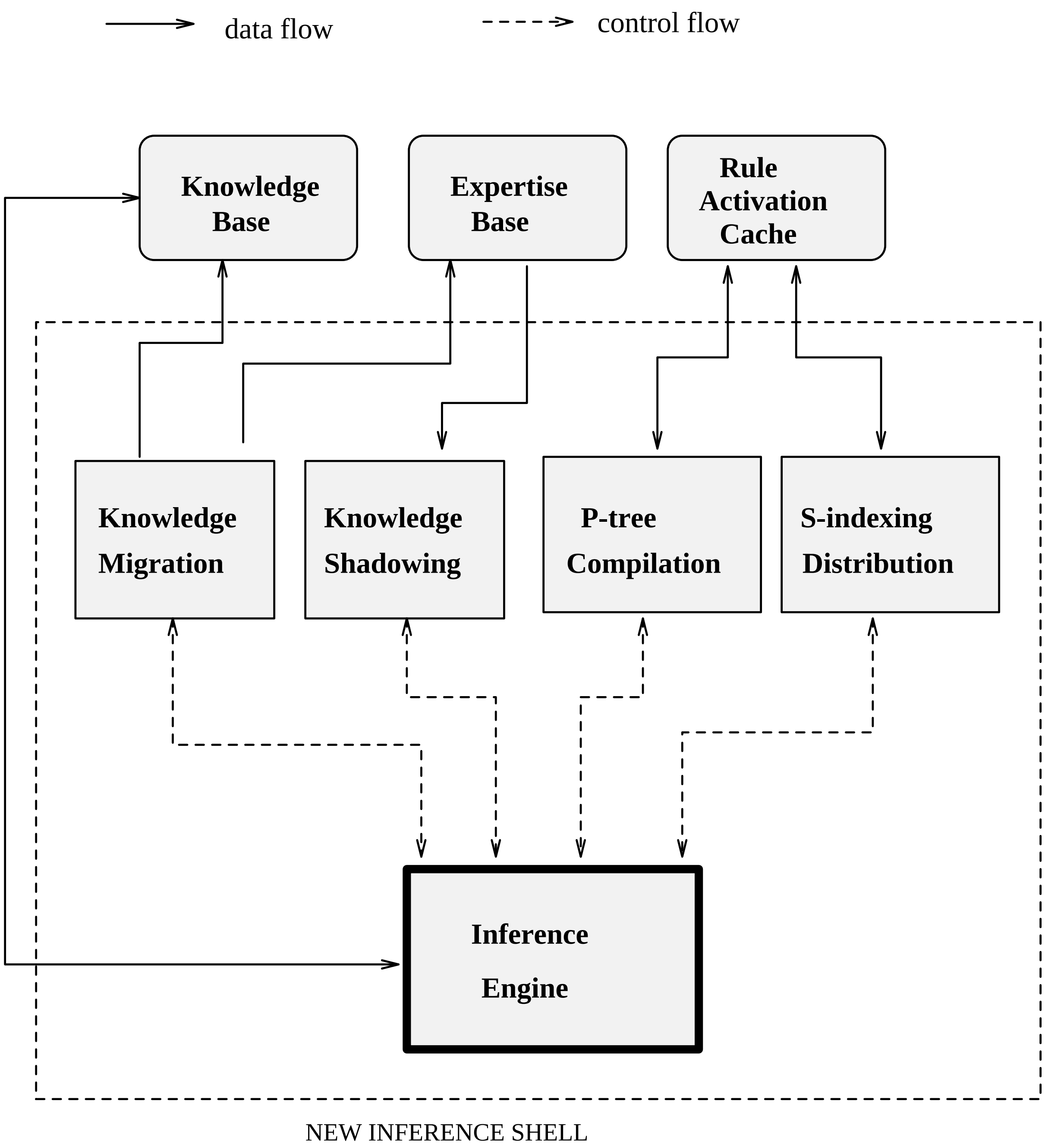


Figure 1.1: Architecture of SNIP2.2

part of expertise information.

The reasoning component is a pure logical inference engine that initiates a deduction, applies a set of rules of inference, and collects all results.

The learning component includes modules of knowledge migration, knowledge shadowing, **S-indexing**, and **P-tree**. Knowledge migration updates both knowledge base and expertise base by asserting derived knowledge to the knowledge base, and at the same time by storing deduction expertise information to the expertise base. Knowledge shadowing retrieves information from the expertise base to choose the best knowledge to be applied. Both **S-indexing** and **P-tree** retrieve and update the content of the rule activation cache.

Several characteristics of SNIP2.2 are worth mentioning.

First, SNIP2.2 is basically a natural deduction system. Rules are represented by not only standard connectives such as negation, conjunction, and disjunction, but also non-standard connectives and quantifiers as described in Chapter 4. These non-standard connectives and quantifiers represent rules more expressively and more compactly than other logic representations. The price that we should pay for the compact representation is efficiency, since in a natural deduction system each rule is associated with rules of inference that are used to derive new knowledge from existing knowledge and the complexity of executing rules of inference is exponential in terms of number of instances of each pattern. The main accomplishment we are claiming is that by using rule activation algorithms we not only express rules in an expressible and more human-like way, but also achieve polynomial complexity for the execution of rules of inference.

Second, there are 3 directions of reasoning in SNIP2.2: (1) backward chaining, (2) forward chaining, and (3) bi-directional chaining. Backward chaining is initiated by asking a question, and requests are directed from consequents of rules to antecedents of rules. When the antecedents are satisfied, the information is directed back to the consequents to draw new conclusions. We call this step *forward triggering.* So backward chaining is the combination of backward requesting and forward triggering. Forward chaining is initiated by adding a new fact, and reports (derived instances) are directed from antecedents of rules to consequents of rules. Bi-directional inference is the combination of backward and forward chaining. Bi-directional inference is made possible by retaining the information about previous backward chaining, and using this information in the future forward chaining.

Third, the system adds derived knowledge to the knowledge base monotonically. This assumption can be contrasted with the method of selective assertion or the method of forgetting. This is discussed in Chapter 3.

Fourth, a query with variables expects all possible answers at once. This is different from most deduction systems that pursue a single answer, or find multiple answers but one by one by a sequence of queries.

1.8 Outline of the Dissertation

In Chapter 2, a number of speedup learning systems are reviewed and compared with SNIP2.2 in terms of how experiential information is retracted, retained, and exploited.

In Chapter 3, techniques of learning at the rule selection level are presented. The algorithms of knowledge migration and knowledge shadowing are described along with the representation of deduction expertise information in the expertise base.

In Chapter 4, techniques of learning at the rule activation level are presented. The RUI data structure is explained, and the algorithms of **S-indexing** and **P-tree** are described.

In Chapter 5, a number of application problems are tested to show the improvement of system performance resulted from the learning methods.

In Chapter 6, we summarize the dissertation and discuss future work.

Chapter 2

Background

2.1 Machine Learning

The issue of developing experience-based problem solving methodologies has been pursued in the area of machine learning. In general, learning denotes changes in a system that are adaptive in the sense that they enable the system to do the same thing or tasks drawn from the same population more efficiently and more effectively the next time [Simon, 1983]. Specifically, learning algorithms attempt to obtain answers more economically, provide more accurate solutions, cover a wider range of problems, or simplify coded knowledge [Forsyth and Rada, 1986]. The area of machine learning is largely classified into *inductive learning* [Michalski, 1983] and *speedup learning* [Laird *et al.*, 1984, Mitchell *et al.*, 1986] depending on whether the system's behavior is changed by acquiring more knowledge from external sources (inductive learning) or by refining and modifying the current knowledge into a better form (speedup learning).

Inductive learning produces general rules or procedures from externally supplied examples, and predicts a result for a new example by using the acquired rules. Inductive learning is also called *empirical learning* or *similarity-based learning*. Inductive learning is further classified into supervised learning and unsupervised learning. In supervised learning, examples of the form (x_i, y_i) are provided, and the system is supposed to learn a function f such that $f(x_i) = y_i$ for all i. In particular, if y_i has two values (e.g. 'arch' or 'no arch' in Winston's arch-learning algorithm), it is called concept learning. Unsupervised learning is to find regularities among a given collection of x_i values. Clustering and discovery are two examples of unsupervised learning which is less developed compared to supervised learning.

Speedup learning, also called *skill acquisition* or *deductive learning*, improves a system's performance by exploiting current knowledge more effectively and by reformulating given knowledge into a better form. Instances of speedup learning include the introduction of macro operators [Fikes *et al.*, 1972] that compose a sequence of primitive operators into a single operator to reduce the search depth required to move from the start state to a goal state, and the introduction of metalevel search control knowledge [Minton, 1988] that enables the selection of proper inference branches. Knowledge compilation [Anderson, 1989, Keller, 1990, Neves and Anderson, 1981], explanation-based learning (EBL) [DeJong and Mooney, 1986, Ellman, 1989, Mitchell *et al.*, 1986], chunking [Laird *et al.*, 1984], and case-based reasoning

[Kolodner *et al.*, 1985] are notable examples of speedup learning.

We are interested in speedup learning methods because our primary concern is to improve system's performance through experience. This chapter reviews some of speedup learning systems and compares them with our approach. The comparison is based on two major issues that have been used to evaluate the effectiveness of learning algorithms [Kolodner and Kolodner, 1987].

The first issue is how to encode or represent experience appropriately. Although each system has its own terminologies and definitions about experience, and hence some details are different across the systems, one thing that is common to all systems is that experience should be represented to make future reasoning more effective. Experience in speedup learning is often obtained by the relative relationship between the learned knowledge and the original knowledge from which it is learned. Thus, a proper representation of experience is related to the concept of knowledge hierarchy and knowledge redundancy. Some of the questions in this issue can be raised as follows:

- Is there a distinction between general and specific knowledge? How are the relationships among different levels of generality represented?
- Is knowledge redundancy maintained as a result of adding learned knowledge to the knowledge base?
- How does experience change the structure of knowledge in memory?

The second issue is how to recognize, recall, and use represented experience in reasoning. In other words, how to integrate experiential information with the problem-solving process. Some of the questions in this category are:

- Is there a systematic way of exploiting redundancy?
- How to distinguish between useful knowledge and unnecessary knowledge in problem solving?
- How to prevent unnecessary knowledge from being used?

Several speedup learning systems including EBL, knowledge compilation, chunking, and case-based reasoning are reviewed according to the aforementioned issues. These systems are compared with SNIP2.2 in Section 2.6.

2.2 Explanation-Based Learning (EBL)

2.2.1 General Description

EBL [Ellman, 1989, Mitchell *et al.*, 1986] is a speedup learning technique that can be applied to learn macro operators and also to learn search control knowledge. EBL starts from a description of a target concept that is too abstract to be operational, and a training example is given along with sufficient domain knowledge to explain why the example is an instance of the target concept. The explanation structure is similar to a normal deduction derivation tree. This structure is traversed to learn a new description that is more operational than the target concept and also more general than the training example.

Inputs of EBL consist of 4 components: 1) a *target concept* that is a concept to be learned, 2) a *training example* that is an example of the target concept, 3) the *domain theory* which is a set of rules and facts to be used in explaining why the training example is an instance of the target concept, and 4) the *operationality criterion* that is a description of predicates over descriptions, specifying the form in which the learned description must be expressed. The output of EBL is a description that is both a generalization of the training example and a specialization of the target concept which satisfies the operationality criterion.

For example, an initial domain theory is represented by the following collection of rules [Mooney, 1990]. Here, a term with a question mark denotes a variable.

knows(?x,?y) $\wedge$ nice-person(?y) $\rightarrow$ likes(?x,?y)
animate(?z) $\rightarrow$ knows(?z,?z)
human(?u) $\rightarrow$ animate(?u)
friendly(?v) $\rightarrow$ nice-person(?v)
happy(?w) $\rightarrow$ nice-person(?w)

Assume the target concept is the rule "knows(?x,?y) $\wedge$ nice-person(?y) $\rightarrow$ likes(?x,?y)", and the operationality criterion is defined by *human* and *happy*. This means that the target

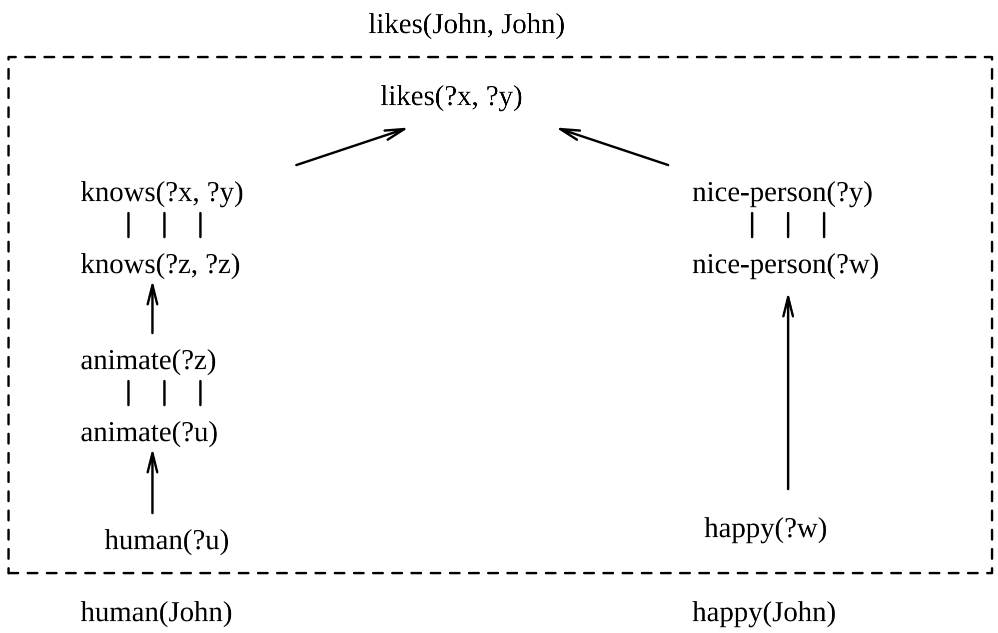


Figure 2.1: The explanation structure for solving *likes(John, John)*

concept should be described by *human* and *happy* predicates. Since the initial description of *likes* does not satisfy the operationality criterion, a specific training example is provided as follows.

Given *human(John)* and *happy(John)*, show that *likes(John, John)*.

Now a deduction derivation of *likes(John, John)* is performed from the above collection of rules and the facts *human(John)* and *happy(John)*. The explanation structure for this problem, as shown in Figure 2.1, is similar to the derivation tree.

One simple rule that might be learned by traversing this explanation structure is

human(John) $\wedge$ happy(John) $\rightarrow$ likes(John,John)

Instead, EBL performs the step of generalization during this traversal and a more general rule than the above rule is actually learned as below.

human(?z) $\wedge$ happy(?z) $\rightarrow$ likes(?z, ?z)

This is a generalization of the training example by generalizing the constant *John* to the variable ?z so that it can be applied to all people. Also, this rule is a specialization of the target concept since the concept of *likes* is defined by more operational predicates *human* and *happy*.

2.2.2 Representing and Using Experience in EBL

Experience in EBL consists of the learned description at the end of learning and the explanation structure that is similar to the deduction tree.

The relationship between the learned rule and the existing rules can also be found in the explanation structure, although this relation is not explicitly exploited in later reasoning.

Rule redundancy is created by adding the learned rule to the knowledge base. In the previous example, the learned rule "human(?z) $\wedge$ happy(?z) $\rightarrow$ likes(?z, ?z)" is redundant because all queries about *likes* that can be solved by using this rule can also be solved without using this rule. The specificity level of the learned rule can be described in two different ways. I.e, the learned rule is more specific than the domain theory rule, but is more general than the training example.

Redundancy in EBL is formally studied by Greiner and his colleagues [Greiner and Likuski, 1989, Greiner, 1991]. This work is basically an extension of Smith's work [Smith, 1989] of finding an optimal strategy for answering a given query to the redundant knowledge base. According to this study, there are two ways of improving the expected cost of a derivation: (1) by determining the best strategy, and (2) by adding redundancies. Generally, using (2) without (1) causes inefficiency, but Greiner's work indicates that efficiency can be achieved by combining (1) and (2).

In fact, Greiner proved that finding an optimal strategy in a general redundant knowledge base is NP-complete. He also claimed that leaving in both the learned rule and the original rule, as most EBL systems do, is never efficient. Instead, his method of finding an optimal strategy removes one of the general rules from the knowledge base to make the knowledge base irredundant. In the example, "knows(?x,?y) $\wedge$ nice-person(?y) $\rightarrow$ likes(?x,?y)" might be deleted from the knowledge base to remove the redundancy. As Greiner pointed out, this method is only applicable to solving the same query, and in order to solve different queries, both rules must be left in the knowledge base. Another limitation is that it can only be applied to disjunctive knowledge bases in which rules can have at most one antecedent. So even the *likes* rule violates this condition.

EBL separates learning from problem solving. Learning is done by traversing the explanation structure to produce a new description. Although the explanation structure implicitly represents the relationship between the learned rule and existing rules, this information is not used in future problem solving.

It seems that the EBL community has focused on generating new descriptions, not having put much emphasis on using these learned descriptions in future problem solving. There is no systematic decision-making of which rule should be applied in the current situation. Rather, since the learned rule is more operational than the initial domain theory, a set of learned rules is tried first, and if they fail to solve the current query, the domain theory is applied.

The EBL method is also used in acquiring effective control rules [Minton, 1988] that pursue the best alternatives at each choice point. Minton is concerned about the genuine efficiency improvement in an EBL system by introducing the notion of *utility* that is a cost/benefit evaluation of acquired knowledge. Utility measures the effectiveness of search control rules, and only those search control knowledge that are determined to be effective will be stored. One disadvantage might be that calculating the utility values is complex.

2.3 Knowledge Compilation

2.3.1 General Description

The main purpose of knowledge compilation is to resolve the problem of *representation mismatch* that has been debated mostly in the knowledge acquisition community. Representation mismatch means that a domain expert provides an initial representation whose form often cannot be interpreted effectively by a target performance system. This is similar to the declarative vs. procedural knowledge distinction.

We discuss knowledge compilation mainly by considering Anderson's ACT system [Anderson, 1989, Neves and Anderson, 1981]. In ACT, knowledge compilation transforms domain-general declarative knowledge to domain-specific procedural knowledge by tracing the activation of a general production to a particular problem task. Domain-general knowledge that is not committed to a particular use is stored in a memory called the *declarative memory* which is separate from the *production memory* that consists of compiled productions that are encoded in a use-specific way.

Knowledge compilation is accomplished by two subprocesses, *proceduralization* and *composition*. Only the proceduralization is discussed here. Proceduralization builds a domain-specific production from a domain-general production, and compiled knowledge by proceduralization is an instance of general knowledge from which it is compiled with some variables replaced by constant elements. For instance, consider the following production described in [Anderson, 1989].

```
If the goal is to achieve =relation on =arg1 and =arg2
   and =operation achieves =relation on =term1 and =term2
Then use =operation
```

Here, terms prefixed by "=' denote variables. This production might apply when the goal is to insert an element into a list, and there is a LISP function CONS that achieves this goal. In this case, the goal is "to achieve insertion of arg1 into arg2", and our knowledge about CONS is "CONS achieves insertion of argument1 into argument2" both of which are stored in the working memory. So we get the following variable bindings.

```
=relation      :    insertion
=operation     :    CONS
=arg1          :    arg1
=arg2          :    arg2
=term1         :    argument1
=term2         :    argument2
```

Notice that the second condition of the production "=operation achieves =relation on =term1 and =term2" is matched with "CONS achieves insertion of argument1 into argument2", and is eliminated in the generation of proceduralized production. In this case, a domain-specific production produced by the proceduralization is as follows.

```
If   the goal is to achieve insertion of =arg1 into =arg2
Then use CONS.
```

One characteristic of proceduralization is that it eliminates reference to the declarative knowledge of the domain, and builds the consequences of the knowledge into domain-specific production rules.

2.3.2 Representing and Using Experience in Knowledge Compilation

In knowledge compilation, two separate knowledge bases are maintained for storing different levels of production rules. Domain-general declarative rules are stored in the *declarative memory*, and domain-specific procedural rules are stored in the *production memory*. Experience consists of those rules in the production memory compiled from the general productions, and the specificity relationship represented by the instantiation. A rule with more variables is regarded as more general than the compiled version with some variables instantiated by constants.

As in EBL, knowledge compilation separates learning from problem solving. Compiled rules are stored separately in the production memory, but the instance relationship obtained during the proceduralization is not explicitly maintained. When solving a problem, the system refers to domain-specific rules in the production memory first, and then uses domain-general rules in the declarative memory if the specific rules fail to solve the current problem. Occasionally, knowledge compilation contributes to system performance by preferring specific rules and by keeping redundant general rules from being used, as indicated in the following quote in [Anderson, 1989], section 6.3.

> "Knowledge compilation in ACT* can actually change the direction of problem solving because of changes in the conflict resolution ... (deleted) .. compilation is *unsafe* in that it is possible that a compiled production will fire in situations when the production(s) from which it was compiled would be *blocked*. This is regarded as a feature, not a bug, in the theory because this allows the system to favor the more efficient rules it has formed."

In fact, this phenomenon is the motivation of our knowledge shadowing scheme which favors more specific rule by blocking more general rules from the inference. However, the decision of which productions to block in ACT is not systematic in that a compiled production is always fired first, and if it fails to solve the problem, the production from which it was compiled will be fired.

A method of using EBL for knowledge compilation is suggested in LSPA (Learning Systems for Pilot Aiding) [Levi *et al.*, 1992]. In LSPA, the pilot's initial representation corresponds to a domain theory of facts and relationships about aircraft tactical maneuvers, and learning instances can come from records of pilot behavior in a flight simulation. Knowledge compilation consists of EBL that creates macro rules that generalize and summarize the explanation of the learning experience, and the macro translator that converts each macro rule into a plan that can be directly executed.

2.4 Chunking

2.4.1 General Description

Chunking [Laird *et al.*, 1984, Laird *et al.*, 1986] is a general learning mechanism developed for the SOAR architecture. SOAR is a general problem solver in which the structure of problem solving supports learning by determining when new knowledge is needed, what to learn, and when new knowledge can be acquired. In goal-based problem solving, chunking creates rules that summarize the processing of a subgoal, so that in the future, it can lead the problem solver directly to the solution without redoing subgoals.

A chunk is basically a production rule consisting of conditions and actions. As each subgoal terminates, successfully or unsuccessfully, a chunk is built that tests the relevant conditions and produces a preference for one of the operators at the choice point.

As an example, consider a macro problem solving in SOAR to an eight puzzle problem. There are eight tiles in a 3x3 frame whose position is named A through I as in Figure 2.2(a). The initial and goal states are represented in Figure 2.2(b) and (c), respectively.

Searches performed for the first three operators are shown in Figure 2.3. The first operator

B	C	D
I	A	E
H	G	F

(a) The positions in the frame

	4	2
3	1	8
5	6	7

(b) Initial State

1	2	3
8		4
7	6	5

(c) Goal State

Figure 2.2: An eight puzzle problem

is to place the blank tile in cell A, the second operator is to place the number 1 tile in Cell B, and the last operator is to place the number 2 tile in Cell C. There are 3 kinds of search tree in each operator. The left column shows the search without learning, the middle column shows the search during learning, and the right column shows search after learning. Here, horizontal arrows indicating no choice is required cause no chunks to be created. A '+' indicates that a chunk is created that preferred a given operator, while a '–' indicates that a chunk is created to avoid an operator. In this example, you can notice that search steps are reduced by applying chunking.

Extensive comparisons between chunking and EBL are made in [Rosenbloom and Laird, 1986]. The similarities of the two methods are discussed in terms of mapping EBL to SOAR. In this mapping, the target concept in EBL corresponds to a goal to be solved in SOAR, the training example corresponds to the situation that exists when a goal is generated, the operationality criterion can be expressed in terms of the predicates existing prior to the creation of the goal, and the domain theory corresponds to a problem space in which the goal can be attempted. Also the proof in EBL can be thought of as the problem solving in SOAR, and the explanation structure can be mapped onto the backtracked production traces in SOAR.

2.4.2 Representing and Using Experience in Chunking

Chunking is a technique of learning search control knowledge by forming new macro operators that take big steps in the search space. By contrast, EBL and knowledge compilation

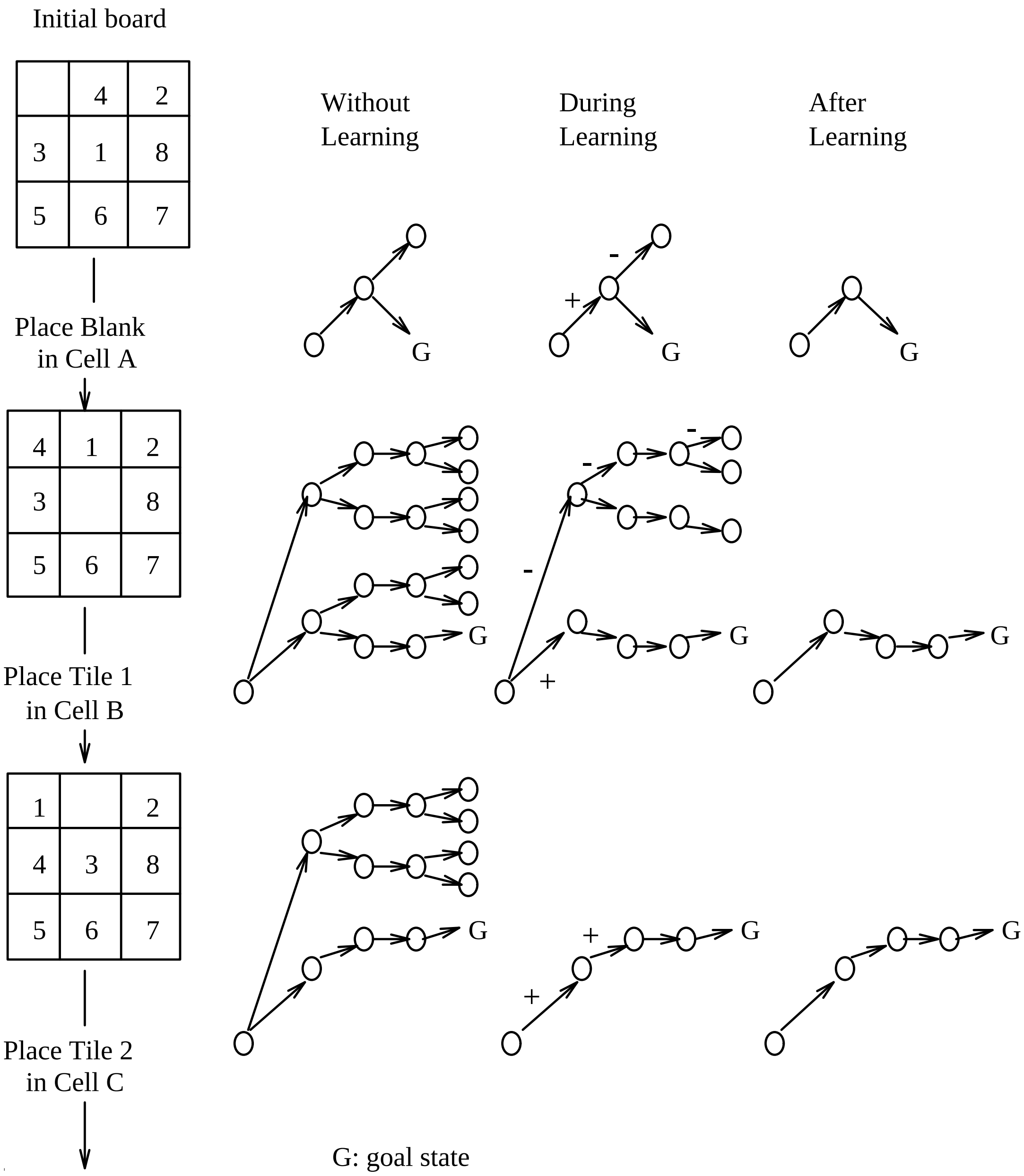


Figure 2.3: Searches performed in an eight puzzle problem

are techniques of learning composite rules that are included in the knowledge base creating redundancy.

Experience in chunking is represented by a number of chunks created during problem solving. A chunk is basically a production rule consisting of conditions and actions. The condition of a chunk consists of those aspect of the situation that existed prior to the goal, and this corresponds to the working-memory elements that were matched by productions that fired in the goal, but that existed before the goal was created. The actions of a chunk are based on the results of the subgoal for which the chunk was created.

The method of chunking is influenced by a plan learning method in the STRIPS robot planning system [Fikes *et al.*, 1972]. A process for generalizing a plan is added to STRIPS, and a *triangle table* is used to store generalized plans that contain a sequence of operator applications. These macro operators could be applied to speed up subsequent plan creation. They also found that problem-solving time and search-tree sizes are all smaller when using learned macro operators.

Performance in chunking is measured by the number of decisions necessary to perform a task, but the cost of matching chunks might be huge when a large number of complex chunks are included. In order to analyze the issue of expensive chunks, a model of matcher is presented in [Tambe and Newell, 1988, Tambe and Rosenbloom, 1989], and the claim is that, although it can be proved that smarter matchers will not eliminate all expensive chunks, learning performance can be improved with some changes in the matcher.

2.5 Case-Based Reasoning (CBR)

2.5.1 General Description

CBR is the problem solving process which solves problems by analogy to previous or hypothetical cases [Hammond, 1986, Kolodner *et al.*, 1985, Simpson, 1985, Sycara, 1987]. CBR can be used to improve the initial understanding of a problem and the generation of solutions, as well as the reinterpretation and selection of alternative lines of reasoning in case of misunderstanding or a failure. CBR is very similar to analogical reasoning. The differences between these two approaches are mentioned in [Burstein, 1989] and [Shavlik and Dietterich, 1990].

Storing new episodes and comparing them with previously-stored episodes are two main tasks in CBR. Storing new episodes is not a simple matter, since unrestricted storing might make it necessary to compare the new problem with all episodes in memory. Therefore, constructing an indexed memory structure is an important issue in this technique [Kolodner and Simpson, 1984, Kolodner, 1984, Schank, 1982]. By comparing the new episodes with previous episodes, the system can infer additional properties of the new episodes, and also generalize the new episode and one or more stored episodes to produce a more abstract concept.

The CBR process involves [Kolodner *et al.*, 1985]: (1) locate and retrieve potentially applicable cases from long term memory; (2) evaluate selected cases to determine the applicable ones; (3) transfer knowledge from the old case(s) to the current one. Problem solving by consulting previous cases and refining them is effective especially for ill-defined domains without a good domain theory including medical decision making and modeling judges.

2.5.2 Representing and Using Experience in CBR

Experience in CBR is denoted by stored episodes with their indexes. We review some developed CBR systems to see how episodes are represented.

The MEDIATOR program [Simpson, 1985] provides advice about which mediation plans might be useful in the resolution of disputes encountered on a daily basis. In this system, experience is represented by *generalized episodes* [Kolodner, 1984, Schank, 1982] which have two components: (1) the *norms* which represent the abstracted content of all the cases organized within that particular episode, and (2) the *indices* which connect the generalized episode with the tree of other generalized episodes and specific cases organized below it. For instance, consider the following two dispute problems:

```
PROBLEM1:  ''DISPUTE'' with
            name: orange-dispute
            disputant1: sister1
            argument1: wants possession of orange1
            disputed-object: orange1
            disputant2: sister2
            argument2: wants possession of orange1
```

```
 norms:    dispute is over possession of object
           object is a kind of physical object
           disputants are people
           disputant's goals are physical control goals
           precedent case is orange dispute
         -----------------------------------------
indices:          |                  |
                goals            disputes object
                  |              /        |
               position     window      orange
                  |       /               |
            window dispute           orange dispute
```

Figure 2.4: Generalized episode for the physical disputes

```
PROBLEM2:  ``DISPUTE'' with
            name: window-dispute
            disputant1: man1
            argument1: wants window1 open
            disputed-object: window1
            disputant2: man2
            argument2: wants window1 closed
```

These are similar cases in the sense that the norms of the two cases are the same since both of these problems are disputes between people about some physical object. Generalized episodes that differentiate these two problems are shown in Figure 2.4.

Here, 'goals' and 'disputes object' play a role of indices, 'position', 'window', and 'orange' are values of those indices. The combination of an index and a value determines a case such as 'window dispute' or 'orange dispute'.

In the CHEF system [Hammond, 1989] that learns about cooking, each episode is called a complete plan. Each plan is indexed by $< S, G, A >$, where S is a starting state, G is a goal state, and A is a set of additional goals to avoid failures. A is obtained before solving a planning problem by investigating any potential failures or risks by applying failure-prediction rules. Each stored plan is also indexed by all known generalizations of S, G, and A. For example, any plan indexed under *beef* is also indexed under *meat* in order to support retrieval of *similar* plans.

In the PROTOS system [Porter *et al.*, 1990] that diagnoses hearing disorders, each case corresponds to a patient information including symptoms, the associated disease diagnosis, and an explanation of why the symptoms indicate the diagnosis. Retrieving similar cases for a new patient is done by comparing the symptoms as well as by examining if the old explanation can be applied to the new case.

In the CASEY system [Koton, 1989] that is applied to medical decision making, each episode contains information about an individual case including the name of the patient, the description of the patient, the causal explanation derived for this patient, the diagnosis, therapy suggestions, and so on. Individual cases are indexed by the features that distinguish them from other cases. Indexing requires two levels, the first level indicates the category of the index and the second level indicates the values that the feature takes on. The set of indices defines a set of paths through the memory structure.

In the JUDGE system [Bain, 1986] that is a model of the behavior of judges who sentence criminals, each episode corresponds to a legal case of murder, assault, or manslaughter. Cases stored in memory are indexed by several salient features including the statute that was violated, who started the fight, the violative actions and results, and the interpretations assigned to those actions and results by the JUDGE program.

Since cases are stored by indexes, the main problems in CBR is the proper design of indexes for organizing case libraries and the development of an efficient retrieval algorithm for finding similar cases. Also general techniques for modifying previous cases must be developed so that they apply to new problems.

It is interesting to note that CBR integrates learning with problem solving, and hence the problem solving component is able to utilize what was learned previously. However, its reasoning process is generally complex with the steps of locating and retrieving potentially applicable cases, evaluating selected cases, and transferring old cases to the current one.

2.6 Discussions

Several speedup learning systems that have been described in previous sections are compared with SNIP2.2.

In SNIP2.2, the first issue of representing experience is done in the knowledge migration phase. Experience is stored in the expertise base that contains the relationship among several redundant rules. Redundancy in knowledge is maintained by adding derived knowledge to the knowledge base. Knowledge migration changes both the knowledge base and the expertise base. That is, derived knowledge is put into the knowledge base, and the experiential knowledge is accumulated in the expertise base.

Knowledge migration is similar to the *proceduralization* in the ACT system, the *operationalization* in EBL, and producing macro search rule in chunking in the sense that some rules are acquired as a result of a deduction, and these rules are used in future deduction to improve efficiency. Also all systems allow knowledge redundancy after problem solving. However, the difference is that knowledge migration not only creates knowledge redundancy, but also extracts and stores the experiential information about how this redundancy occurred. It is different from the proceduralization in that we do not store the acquired rule in a separate memory from the memory containing general rules. It is also different from the operationalization in that the form of a learned rule is not restricted by the operationality criterion that specifies predicates allowed in expressing the rule, but by the form of the general rule from which the acquired rule is generated. Unlike chunking, which produces explicit search control knowledge that is represented as production rules, our learning mechanism employs an implicit search control scheme that uses the information in the expertise base.

The second issue of using experiential knowledge in problem solving is accomplished in SNIP2.2 by knowledge shadowing. A key feature to accomplish this is the integration of learning and problem solving. In most learning systems where learning and reasoning are separate, the experiential information obtained during learning may not be fully utilized by the problem solving component. The integration of deduction and learning components solves this problem by enabling the deduction component to use information obtained by the learning process and eventually to decide which knowledge is most effective at a certain point when several rules at different levels of generality are available. The decision-making proceeds by remembering the relationships among several rules as stored, and comparing them to the current situation to choose the most appropriate ones.

Note that case-based reasoning also integrates learning with problem solving, but its representation of experience consists of entire episode of previous cases, and consequently its reasoning process is generally complex with the steps of locating and retrieving potentially applicable cases, evaluating selected cases, and transferring old cases to the current one. Compared to this, we limit the scope of experience to the relationship among redundant knowledge, which results in a relatively simple process of storing and retrieving expertise information.

Chapter 3

Learning at the Rule Selection Level

It has been claimed that reasoning by using specific knowledge causes less system overload than by using general knowledge [Hart, 1982], and consequently specific knowledge significantly contributes to system efficiency. On the other hand, general knowledge is also useful for unfamiliar situations which known specific knowledge is unable to handle. Therefore, in order to achieve both system efficiency and generality, we want to keep several different levels of knowledge in a multi-level knowledge structure, and select appropriate knowledge according to the current situation.

Rule selection has been a critical problem in AI, since unguided selection of rules may lead to a combinatorial explosion in the number of different inference steps possible [Smith, 1989]. A selection criterion that we focus on is the specificity relationship among knowledge that is obtained from previous reasoning by creating knowledge redundancy.

Two learning schemes, knowledge migration and knowledge shadowing, are presented for experience-based rule selection. Knowledge migration is a process of generating specific rules from general rules. Knowledge migration also accumulates deduction experience represented by the specificity relationship between migrating and migrated knowledge. Knowledge shadowing uses deduction experience to shadow or block unnecessary deduction branches to make subsequent similar deduction faster.

3.1 Transitive Rule Example

The transitive rule example introduced in Chapter 1 is discussed in more detail to get a sketch of the mechanisms of knowledge migration and knowledge shadowing.

A knowledge base for this example is as follows:

(r1) $\forall$r [trans(r) $\rightarrow$ $\forall$x,y,z [r(x,y) $\wedge$ r(y,z) $\rightarrow$ r(x,z)]]
(f1) trans(on)
(f2) on(a,b)
(f3) on(b,c)
(f4) on(c,d)

A query on(a,c) to this knowledge base leads to the following natural deduction derivation [Bibel, 1986]. This derivation is also shown graphically in Figure 3.1.

(p1) trans(on) → ∀x,y,z [on(x,y) ∧ on(y,z) → on(x,z)]

from (r1) *by Universal Instantiation with a substitution* {*on/r*}

(p2) ∀x,y,z [on(x,y) ∧ on(y,z) → on(x,z)]

from (p1) *and* (f1) *by Modus Ponens*

(p3) on(a,b) ∧ on(b,c) → on(a,c)

from (p2) *by Universal Instantiation with a substitution* {*a/x, b/y, c/z*}

(p4) on(a,b) ∧ on(b,c)

from (f2) *and* (f3) *by And Introduction*

(p5) on(a,c)

from (p3) *and* (p4) *by Modus Ponens*

During this inference, a specific rule r2 (p2 above) is generated as an intermediate result. In fact, r2 is an instance of $r1_{cq}$, the consequent of r1.

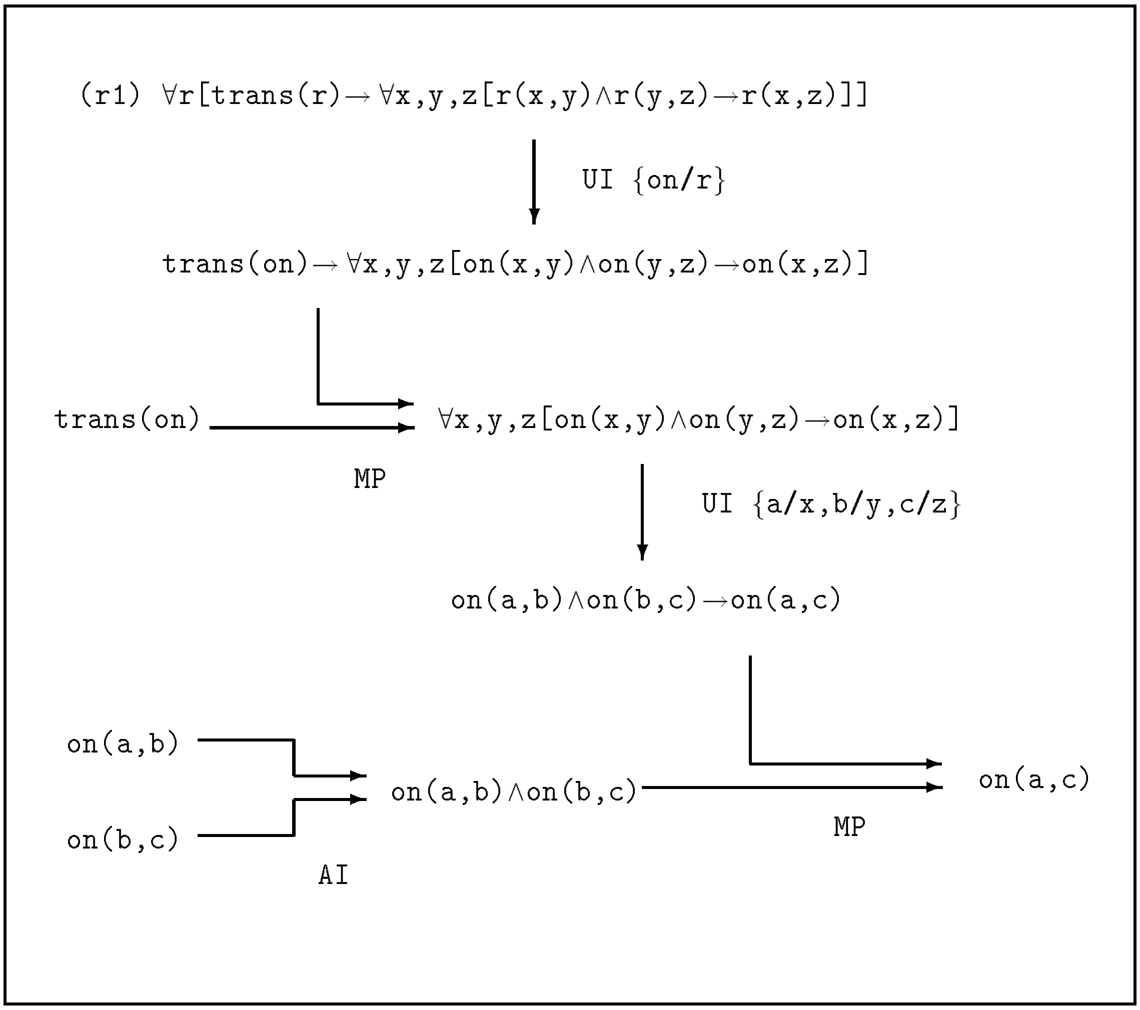
(r2) ∀x,y,z [on(x,y) ∧ on(y,z) → on(x,z)]
($r1_{cq}$) ∀x,y,z [r(x,y) ∧ r(y,z) → r(x,z)]

We call this process of generating a specific rule from a general rule *knowledge migration*. In this example, r2 is said to be *migrated* from r1 by a *migrating substitution* {*on/r*}. By asserting derived knowledge, the knowledge base is changed to

(r1) ∀r [trans(r) → ∀x,y,z [r(x,y) ∧ r(y,z) → r(x,z)]]
(r2) ∀x,y,z [on(x,y) ∧ on(y,z) → on(x,z)]
(f1) trans(on)
(f2) on(a,b)
(f3) on(b,c)
(f4) on(c,d)
(f5) on(a,c)

The knowledge base becomes rule redundant since the next query on(b,d) can be answered in two ways; one by using r1 and the other by using r2.

<u>First branch:</u>



UI: Universal Instantiation
MP: Modus Ponens
AI: And Introduction

Figure 3.1: A natural deduction derivation of on(a,c).

(`p1`) `trans`(on) $\rightarrow \forall$x,y,z [`on`(x,y) $\wedge$ `on`(y,z) $\rightarrow$ `on`(x,z)]

from (`r1`) *by Universal Instantiation with a substitution* $\{on/r\}$

(`p2`) $\forall$x,y,z [`on`(x,y) $\wedge$ `on`(y,z) $\rightarrow$ `on`(x,z)]

from (`p1`) *and* (`f1`) *by Modus Ponens*

(`p6`) `on`(b,c) $\wedge$ `on`(c,d) $\rightarrow$ `on`(b,d)

from (`p2`) *by Universal Instantiation with a substitution* $\{b/x,\ c/y,\ d/z\}$

(`p7`) `on`(b,c) $\wedge$ `on`(c,d)

from (`f3`) *and* (`f4`) *by And Introduction*

(`p8`) `on`(b,d)

from (`p6`) *and* (`p7`) *by Modus Ponens*

<u>Second branch:</u>

(`p6`) `on`(b,c) $\wedge$ `on`(c,d) $\rightarrow$ `on`(b,d)

from (`r2`) *by Universal Instantiation with a substitution* $\{b/x,\ c/y,\ d/z\}$

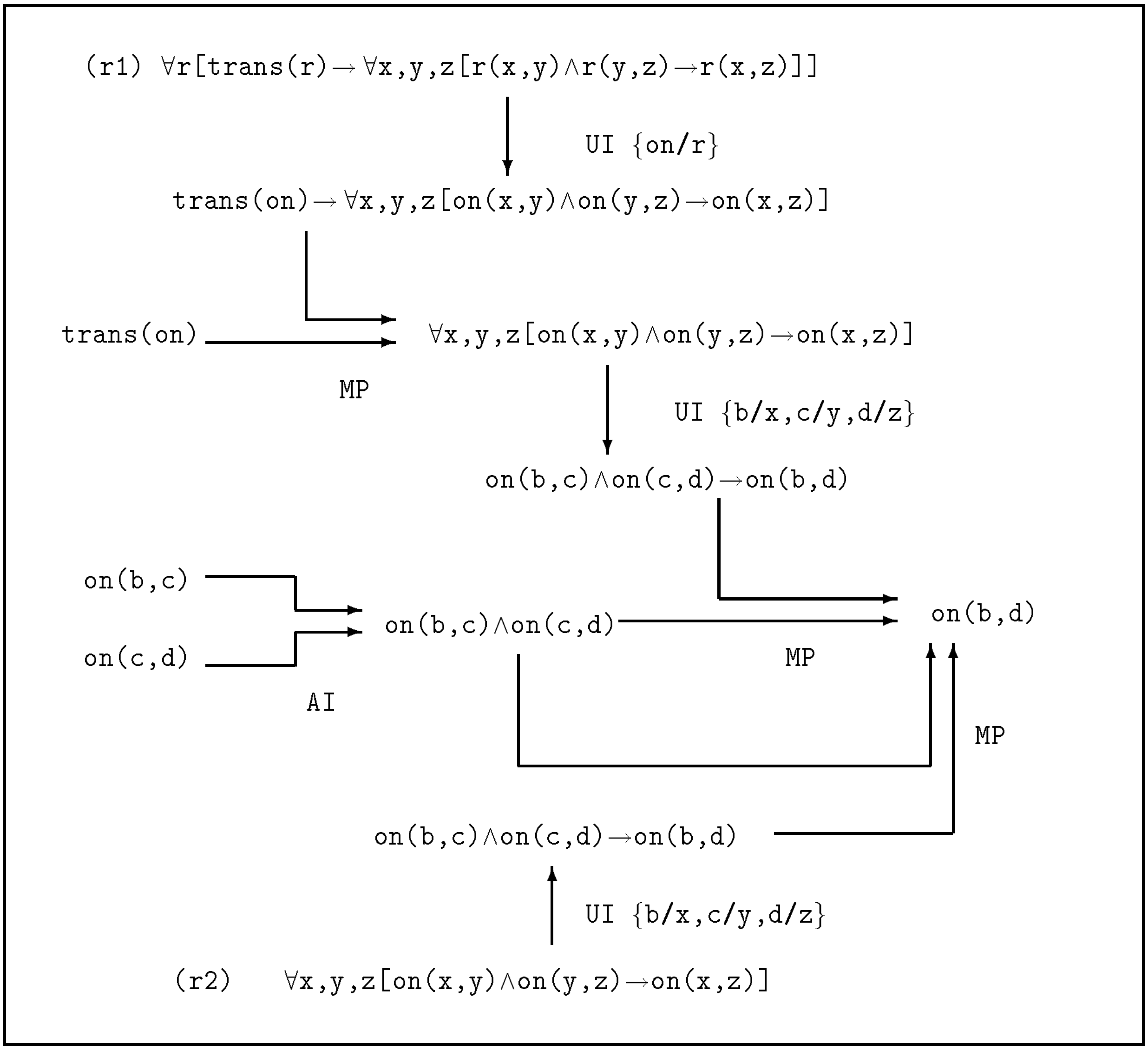
(`p7`) `on`(b,c) $\wedge$ `on`(c,d)

from (`f3`) *and* (`f4`) *by And Introduction*

(`p8`) `on`(b,d)

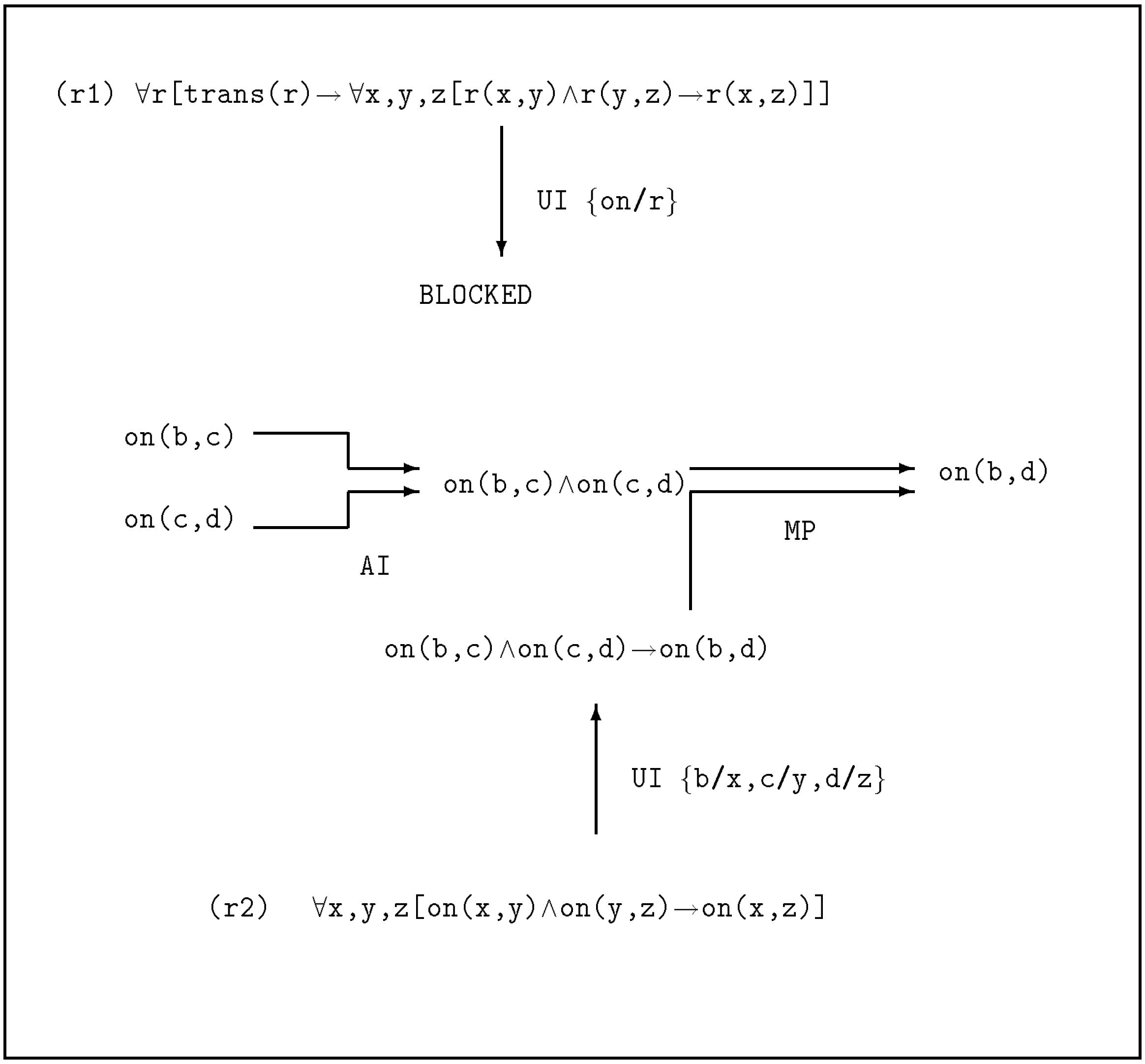
from (`p6`) *and* (`p7`) *Modus Ponens*

This derivation is graphically shown in Figure 3.2. The steps in the first branch are the same as those in the derivation of `on`(a,c) except for substitutions in universal instantiations. Thus, it can be recognized that the first branch is redundant in the sense that `on`(b,d) can be derived solely by the second branch with fewer steps and also the first branch is not producing anything new. Eventually, we want to block the redundant branch as shown in Figure 3.3. This process of blocking unnecessary branches from the inference is called *knowledge shadowing*. In this example, the branch activated from `r1` is said to be *shadowed* by the branch activated from `r2`. The major effect of knowledge shadowing is that the system can solve a problem more quickly than the previous similar problem even though the size of the knowledge base was increased



UI: Universal Instantiation
MP: Modus Ponens
AI: And Introduction

Figure 3.2: A natural deduction derivation of on(b,d)



UI: Universal Instantiation
MP: Modus Ponens
AI: And Introduction

Figure 3.3: A natural deduction derivation of on(b,d) where a redundant deduction branch is blocked

and rule redundancy has caused more possible deduction branches.

3.2 Deduction Expertise

Compared to other rule generating processes including the operationalization in EBL and the proceduralization in knowledge compilation, a peculiar feature of knowledge migration is that the instance relationship between the migrating rule and the migrated rule is saved and exploited in future reasoning. This information is called *deduction expertise*.

Two types of expertise are considered. The first type of expertise is based on substitutions that are applied during the migration process. An instance set is managed for each rule to record the information about which substitutions have contributed to the migration of specific rules. The second type of expertise uses the concept of origin sets that have been defined in a truth maintenance system for propositional dependency.

3.2.1 Instance Set Log

As we have seen in the transitive rule example, the generation of a specific rule from a nested general rule involves a variable substitution called a *migrating substitution* that is informally defined below.

Definition A *migrating substitution* σ is a variable substitution that unifies all antecedents of a nested rule with facts in the knowledge base.

Each binding in this migrating substitution has the form of c/v, where c is a constant and v is a variable. Note that c itself cannot be a variable.

In the transitive rule example, $\sigma = \{on/r\}$ that unifies `trans`(r), which is the antecedent of `r1`, with `trans`(on), which is a fact in the knowledge base. This migrating substitution causes the generation of the consequent of a nested rule. For instance, `r2` is generated from `r1` by applying σ to the consequent of `r1`. A migrating substitution creates an instance relationship between a general (migrating) rule and a specific (migrated) rule. An instance relationship is represented by a 3-tuple as below.

Definition An *instance relationship* is a 3-tuple $< \mathbf{s}, \mathbf{g}, \sigma >$, where σ is a migrating substitution by which a specific rule $\mathbf{s}$ is migrated from a general rule $\mathbf{g}$.

A relationship $< \mathbf{s}, \mathbf{g}, \sigma >$ satisfies that $\mathbf{g}\sigma = \mathbf{s}$. In the transitive rule example, an instance relationship after the migration will be $\{< \mathtt{r2}, \mathtt{r1}_{cq}, \{on/r\} >\}$, where $\mathtt{r1}_{cq}$ is the consequent of `r1`.

In order to memorize and reuse instance relationships obtained during deductions, an *instance set* $\mathcal{I}_{\mathbf{g}}$ is managed for each rule $\mathbf{g}$.

Definition An *instance set* $\mathcal{I}_{\mathbf{g}} = \{< \mathbf{s}_1, \sigma_1 >, < \mathbf{s}_2, \sigma_2 >, \ldots, < \mathbf{s}_l, \sigma_l >\}$, where $\mathbf{s}_i$ is a specific rule that is migrated from a general rule $\mathbf{g}$ by a migrating substitution σ_i $(1 \leq i \leq l)$. Each pair $< \mathbf{s}_i, \sigma_i >$ is called a *migrated instance.*

After the migration from `r1` occurs during the derivation of on(a,c), $\mathcal{I}_{\mathtt{r1}_{cq}}$ becomes $\{< \mathtt{r2}, \{\mathtt{on}/\mathtt{r}\} >\}$. Note that an instance relationship $< \mathbf{s}, \mathbf{g}, \sigma >$ is implemented by adding a migrated instance $< \mathbf{s}, \sigma >$ to $\mathcal{I}_{\mathbf{g}}$.

Definition An *instance set information* is a pair $< \mathbf{r}, \mathcal{I}_{\mathbf{r}} >$, where $\mathbf{r}$ is a rule and $\mathcal{I}_{\mathbf{r}}$ is the instance set of $\mathbf{r}$.

The *instance set log* denoted by $\mathcal{IL}$ is maintained to record all instance set information.

Definition The *instance set log* $\mathcal{IL}$ is defined as a set of instance set information.

$\mathcal{IL} = \{< \mathbf{r}_1, \mathcal{I}_{\mathbf{r}_1} >, < \mathbf{r}_2, \mathcal{I}_{\mathbf{r}_2} >, \ldots, < \mathbf{r}_n, \mathcal{I}_{\mathbf{r}_n} >\}.$

3.2.2 Origin Set Log

The second type of deduction expertise can be discussed in the context of belief revision systems. In these systems, each proposition is associated with a record of where each proposition in the knowledge base came from in order to keep track of and propagate propositional dependencies. This record is called a *support.* In an assumption based truth maintenance system (`ATMS`) which is a kind of belief revision system, the support of each proposition contains hypotheses (nonderived propositions) that produced it. In the SWM system [Martins, 1983, Martins and Shapiro, 1988], this support is named an *origin set*. An origin set of a proposition contains every hypothesis used in its derivation. Using the origin set, when a contradiction is detected, we should be able to identify exactly which assumptions were used in the derivation of the contradictory propositions.

Definition O_p denotes the origin set of a proposition p.

For example, consider the following knowledge base:

$(\mathtt{r}_{mp})$ $\forall$x [man(x) $\rightarrow$ person(x)]
$(\mathtt{r}_{ph})$ $\forall$x [person(x) $\rightarrow$ human(x)]
$(\mathtt{r}_{hp})$ $\forall$x [human(x) $\rightarrow$ person(x)]

Adding man(fred) to the knowledge base causes the derivation of person(fred), and in turn, person(fred) causes the derivation of human(fred). So the origin sets of man(fred), person(fred), and human(fred) are:

$$O_{\mathtt{man(fred)}} = \{\mathtt{man}(\mathrm{fred})\}$$
$$O_{\mathtt{person(fred)}} = \{\mathtt{man}(\mathrm{fred}), \mathtt{r}_{mp}\}$$
$$O_{\mathtt{human(fred)}} = \{\mathtt{man}(\mathrm{fred}), \mathtt{r}_{mp}, \mathtt{r}_{ph}\}$$

If man(fred) is no longer believed afterward, person(fred) and human(fred) should also be no longer believed since the origin sets of person(fred) and human(fred) contain man(fred).

From the viewpoint of deductive learning, propositional dependencies represented by an origin set can be regarded as a type of deduction expertise. Consider the initial knowledge base of the previous transitive rule example that is now represented with origin sets.

(r1) $\forall$r [trans(r) $\rightarrow$ $\forall$x,y,z [r(x,y) $\wedge$ r(y,z) $\rightarrow$ r(x,z)]] : $O_{\mathtt{r1}} = \{\mathtt{r1}\}$
(f1) trans(on) : $O_{\mathtt{f1}} = \{\mathtt{f1}\}$
(f2) on(a,b) : $O_{\mathtt{f2}} = \{\mathtt{f2}\}$
(f3) on(b,c) : $O_{\mathtt{f3}} = \{\mathtt{f3}\}$
(f4) on(c,d) : $O_{\mathtt{f4}} = \{\mathtt{f4}\}$

r2 is derived from r1 and f1, resulting in the following support structures.

(r2) $\forall$x,y,z [on(x,y) $\wedge$ on(y,z) $\rightarrow$ on(x,z)] : $O_{\mathtt{r2}} = \{\mathtt{r1,f1}\}$

(f5) on(a,c) : $O_{\mathtt{f5}} = \{\mathtt{r1,f1,f2,f3}\}$

Definition An *origin set information* is a pair $< \mathbf{k}, O_{\mathbf{k}} >$, where $\mathbf{k}$ is a rule or a fact, and $O_{\mathbf{k}}$ is the origin set of $\mathbf{k}$.

The *origin set log* denoted by $\mathcal{OL}$ is maintained to record all origin set information.

Definition The *origin set log* $\mathcal{OL}$ is a set of origin set information.

$\mathcal{OL} = \{< \mathbf{k}_1, O_{\mathbf{k}_1} >, < \mathbf{k}_2, O_{\mathbf{k}_2} >, \ldots, < \mathbf{k}_m, O_{\mathbf{k}_m} >\}$.

The two types of expertise are combined to build the *expertise base* denoted by $\mathcal{EB}$.

Definition The *expertise base* $\mathcal{EB}$ is the collection of all instance set information and origin set information.

$$\begin{aligned}\mathcal{EB} &= < \mathcal{IL}, \mathcal{OL} > \\ &= < \{< \mathbf{r}_1, \mathcal{I}_{\mathbf{r}_1} >, < \mathbf{r}_2, \mathcal{I}_{\mathbf{r}_2} >, \ldots, < \mathbf{r}_n, \mathcal{I}_{\mathbf{r}_n} >\}, \\ &\quad \{< \mathbf{k}_1, O_{\mathbf{k}_1} >, < \mathbf{k}_2, O_{\mathbf{k}_2} >, \ldots, < \mathbf{k}_m, O_{\mathbf{k}_m} >\} >\end{aligned}$$

3.3 Knowledge Migration

A formal definition of the scheme of knowledge migration is given in this section. Note that knowledge migration generates a specific rule from a general rule, and both the knowledge base and the expertise base are updated as a result of it.

Definition The *knowledge base* $\mathcal{KB} = < \mathcal{F}, \mathcal{R} >$, where $\mathcal{F}$ is a set of facts and $\mathcal{R}$ is a set of rules.

If we define a term as a predicate with arguments, a fact is a term with no variables. A rule consists of antecedents and consequents. Each antecedent is a term which may contain variables. Each consequent is either a term or a rule. If it is a term, we call it non-rule consequent, and if it is a rule, we call it rule consequent. A rule is formally defined below.

Definition A rule $\mathbf{r} = < ANT, NRCQ, RCQ >$, where ANT is a set of antecedents, $NRCQ$ is a set of non-rule consequents, and RCQ is a set of rule consequents

In this abstraction, every variable in a rule is assumed to be universally quantified. Existentially quantified variables are skolemized. The semantics of a rule is that if all antecedents are satisfied, then all consequents are derived. Disjunctive rules can be represented by building a separate rule for each disjunct.

If any consequent of a rule is also a rule, it is called a *nested rule* or an *embedded rule*. In other words, $\mathbf{r} = < ANT, NRCQ, RCQ >$ is a nested rule if RCQ is not empty.

For example, a rule

$$\forall \mathrm{x,y}\ [(\mathtt{male}(\mathrm{x}) \wedge \mathtt{parent}(\mathrm{x,y})) \rightarrow$$
$$(\mathtt{father}(\mathrm{x,y}) \wedge \forall \mathrm{z}\ [\mathtt{parent}(\mathrm{y,z}) \rightarrow \mathtt{grandfather}(\mathrm{x,z})])\ ]$$

is a nested rule that is represented recursively by

$$< \{\mathtt{male}(\mathrm{x}), \mathtt{parent}(\mathrm{x,y})\}, \{\mathtt{father}(\mathrm{x,y})\},$$
$$\{< \{\mathtt{parent}(\mathrm{y,z})\}, \{\mathtt{grandfather}(\mathrm{x,z})\}, \{\}>\}>$$

A system state is defined as the amalgamation of information in the knowledge base and in the expertise base.

Definition A *system state* $\mathcal{S} = < \mathcal{KB}, \mathcal{EB} >$, where $\mathcal{KB}$ is the knowledge base, and $\mathcal{EB}$ is the expertise base of the system.

Knowledge migration updates the knowledge base by adding migrated knowledge to the current knowledge base, and also updates the expertise base by adding instance set information and origin set information. Eventually, knowledge migration transforms a system state into a new system state.

Definition *Knowledge migration* is a function $\mathcal{M}$ which changes a system state into a new system state when (1) a nested rule

$$\mathbf{r}_e =< \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_l\}, \{\mathbf{nr}_1, \mathbf{nr}_2, \ldots, \mathbf{nr}_m\}, \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n\} >$$

is activated during a deduction, and (2) there exists a migrating substitution σ such that, for each antecedent $\mathbf{a}_i$ $(1 \leq i \leq l)$, either $\mathbf{a}_i$ is unified with a fact $\mathbf{f}_i \in \mathcal{F}$ $(\mathbf{a}_i\sigma = \mathbf{f}_i)$, or $\mathbf{a}_i\sigma$ can be derived from rule chaining.

More formally,

$$\mathcal{M}(\mathcal{S}) = \mathcal{S}' \text{ or}$$
$$\mathcal{M}(<< \mathcal{F}, \mathcal{R} >, < \mathcal{IL}, \mathcal{OL} >>) \quad = \quad << \mathcal{F}', \mathcal{R}' >, < \mathcal{IL}', \mathcal{OL}' >>.$$

A system state change caused by knowledge migration is explained for each component of the system state.

$\underline{\mathcal{F} \quad \rightarrow \quad \mathcal{F}'}$

$\mathcal{F}$ is incremented by the facts derived from non-rule consequents of the nested rule. Hence, $\mathcal{F}' = \mathcal{F} \cup \mathcal{F}_s$, where $\mathcal{F}_s = \{\mathbf{nrs}_1, \mathbf{nrs}_2, \ldots, \mathbf{nrs}_m\}$ is a set of derived facts. Here, $\mathbf{nrs}_i = \mathbf{nr}_i\sigma$ $(1 \leq i \leq m)$. $\mathcal{F}$ is also incremented by final answers to non-rule queries.

$\underline{\mathcal{R} \quad \rightarrow \quad \mathcal{R}'}$

$\mathcal{R}$ is incremented by the migrated rules. Hence, $\mathcal{R}' = \mathcal{R} \cup \mathcal{R}_s$, where $\mathcal{R}_s = \{\mathbf{rs}_1, \mathbf{rs}_2, \ldots, \mathbf{rs}_n\}$ is a set of migrated rules. Here, $\mathbf{rs}_i = \mathbf{r}_i\sigma$ $(1 \leq i \leq n)$.

$\underline{\mathcal{IL} \quad \rightarrow \quad \mathcal{IL}'}$

$\mathcal{IL}$ is incremented by new instance set information. More specifically, the instance set of $\mathbf{r}_i$ is incremented by $< \mathbf{rs}_i, \sigma >$ when a specific rule $\mathbf{rs}_i$ is migrated from $\mathbf{r}_i$ $(1 \leq i \leq n)$. Hence,

$$\mathcal{IL}' = \mathcal{IL} \oplus \mathcal{IL}_s$$
$$\mathcal{IL}_s = \{< \mathbf{r}_1, \{< \mathbf{rs}_1, \sigma >\} >, < \mathbf{r}_2, \{< \mathbf{rs}_2, \sigma >\} >, \ldots, < \mathbf{r}_n, \{< \mathbf{rs}_n, \sigma >\} >\}.$$

Here, the operator $\oplus$ makes a union of two sets except collapsing any two pairs $< r, A_1 >$ and $< r, A_2 >$ into $< r, A_1 \cup A_2 >$.

$$\begin{aligned} S_1 \oplus S_2 = & (S_1 - \{< r, A_1 > \mid < r, A_1 > \in S_1 \wedge < r, A_2 > \in S_2\}) \cup \\ & (S_2 - \{< r, A_2 > \mid < r, A_1 > \in S_1 \wedge < r, A_2 > \in S_2\}) \cup \\ & \{< r, A_1 \cup A_2 > \mid < r, A_1 > \in S_1 \wedge < r, A_2 > \in S_2\} \end{aligned}$$

For instance,

$$S_1 = \{< a, \{A, B\} >, < b, \{C, D\} >, < c, \{E\} >\}$$
$$S_2 = \{< a, \{F\} >, < c, \{G\} >, < d, \{H, I\} >\}$$
$$S_1 \oplus S_2 = \{< a, \{A, B, F\} >, < b, \{C, D\} >, < c, \{E, G\} >, < d, \{H, I\} >\}$$

$\underline{\mathcal{OL} \;\rightarrow\; \mathcal{OL}'}$

$\mathcal{OL}$ is incremented by new origin set information. Since the migration of specific rules $\mathbf{rs}_i$ $(1 \le i \le n)$ involves facts that are unified with antecedent patterns of the nested rule, the origin set of each migrated rule should contain the origin set of each fact that contributed to the successful antecedent pattern matchings. So

$$\mathcal{OL}' = \mathcal{OL} \oplus \mathcal{OL}_s$$
$$\mathcal{OL}_s = \{< \mathbf{rs}_1, O^* >, \ldots, < \mathbf{rs}_n, O^* >, < \mathbf{nrs}_1, O^* >, \ldots, < \mathbf{nrs}_m, O^* >, \}$$
$$O^* = O_{\mathbf{r}_e} \cup O_{\mathbf{a}_1\sigma} \cup O_{\mathbf{a}_2\sigma} \cup \cdots \cup O_{\mathbf{a}_l\sigma}.$$

As an example, consider the transitive rule knowledge base. The initial system state can be described as

$$\mathcal{S} \;=\; < \mathcal{KB}, \mathcal{EB} > \;=\; << \mathcal{F}, \mathcal{R} >, < \mathcal{IL}, \mathcal{OL} >>, \qquad \text{where}$$

$$\mathcal{F} = \{\texttt{f1,f2,f3,f4}\}$$
$$\mathcal{R} = \{\texttt{r1}\}$$
$$\mathcal{IL} = \emptyset$$
$$\mathcal{OL} = \{< \texttt{r1}, \{\texttt{r1}\} >, < \texttt{f1}, \{\texttt{f1}\} >, < \texttt{f2}, \{\texttt{f2}\} >, < \texttt{f3}, \{\texttt{f3}\} >, < \texttt{f4}, \{\texttt{f4}\} >\}$$

The condition of knowledge migration is satisfied for a nested rule `r1` during the processing of `on`(a,c), since a migrating substitution $\sigma = \{on/r\}$ unifies the antecedent `trans`(r) with the fact `trans`(on). As a result of this migration, `r2` is migrated and added to $\mathcal{R}$. After the derivation of `on`(a,c), a new system state becomes as below.

$$\mathcal{S}' =< \mathcal{KB}', \mathcal{EB}' >=<< \mathcal{F}', \mathcal{R}' >, < \mathcal{IL}', \mathcal{OL}' >>, \qquad \text{where}$$

$$\mathcal{F}' = \{\texttt{f1,f2,f3,f4,f5}\}$$
$$\mathcal{R}' = \{\texttt{r1,r2}\}$$
$$\mathcal{IL}' = \{< \texttt{r1}_{cq}, \{< \texttt{r2}, \{on/r\} >\} >\}$$
$$\mathcal{OL}' = \{< \texttt{r1}, \{\texttt{r1}\} >, < \texttt{r2}, \{\texttt{r1}, \texttt{f1}\} >, < \texttt{f1}, \{\texttt{f1}\} >, < \texttt{f2}, \{\texttt{f2}\} >,$$
$$< \texttt{f3}, \{\texttt{f3}\} >, < \texttt{f4}, \{\texttt{f4}\} >, < \texttt{f5}, \{\texttt{r1}, \texttt{f1}, \texttt{f2}, \texttt{f3}\} >\} >$$

3.3.1 Nested Rule Representation

Notice that nested rule representation is emphasized in the knowledge migration process for two reasons.

First, we want to take advantage of the semantics of natural nesting in a rule. Natural nesting appears in many occasions of representing generic types of rules. For example, `r1` is a generic rule for all transitive relations. We might represent this rule by simple conjunctions without using rule nesting as follows:

$$(\mathtt{r1}') \qquad \forall \mathrm{r,x,y,z}\ [\mathtt{trans}(\mathrm{r}) \wedge \mathtt{r}(\mathrm{x,y}) \wedge \mathtt{r}(\mathrm{y,z}) \rightarrow \mathtt{r}(\mathrm{x,z})\ ]$$

In fact, `r1` and $\mathtt{r1}'$ are functionally equivalent, meaning that any knowledge that can be derived from `r1` can also be derived from $\mathtt{r1}'$, and vice versa. However, the "flat" representation of $\mathtt{r1}'$ achieved by placing the first conjunct $\mathtt{trans}(\mathrm{r})$ in the same level with $\mathtt{r}(\mathrm{x,y})$ and $\mathtt{r}(\mathrm{y,z})$ makes it hard to capture its real meaning, since it is difficult to separate the transitive relation which is $[\mathtt{r}(\mathrm{x,y}) \wedge \mathtt{r}(\mathrm{y,z}) \rightarrow \mathtt{r}(\mathrm{x,z})]$ from the main body of the rule. We believe that the first conjunct should be interpreted as one level higher than the second and third conjuncts, and consequently the representation of `r1` seems to be more natural than $\mathtt{r1}'$ as far as semantic interpretation is concerned.

Second, nested rule representation enables the rule builder to specify which of the possible specializations of the rule will be most useful. The level of specificity of a migrated rule depends on the way the nested rule was represented with quantifiers. The nested rule definition delivers the intention of a rule builder about which of the possible migrated forms will be most useful. It is clear that different specific rules would have been generated if `r1` had been expressed differently with different quantifier declarations such as $\mathtt{r1}_a$ or $\mathtt{r1}_b$:

$$(\mathtt{r1}_a) \qquad \forall \mathrm{r,x}\ [\mathtt{trans}(\mathrm{r}) \rightarrow \forall \mathrm{y,z}\ [\mathtt{r}(\mathrm{x,y}) \wedge \mathtt{r}(\mathrm{y,z}) \rightarrow \mathtt{r}(\mathrm{x,z})]]$$

$$(\mathtt{r1}_b) \qquad \forall \mathrm{r,x,y}\ [\mathtt{trans}(\mathrm{r}) \wedge \mathtt{r}(\mathrm{x,y}) \rightarrow \forall \mathrm{z}\ [\mathtt{r}(\mathrm{y,z}) \rightarrow \mathtt{r}(\mathrm{x,z})]]$$

No matter which definition is applied, the system can derive $\mathtt{on}(\mathrm{a,c})$, but different specific rules would be migrated from different rules. I.e., $\mathtt{r2}_a$ and $\mathtt{r2}_b$ might be migrated from $\mathtt{r1}_a$ and $\mathtt{r1}_b$, respectively.

($\mathtt{r2}_a$) $\forall$y,z [$\mathtt{on}$(a,y) $\wedge$ $\mathtt{on}$(y,z) $\rightarrow$ $\mathtt{on}$(a,z)]

($\mathtt{r2}_b$) $\forall$z [$\mathtt{on}$(b,z) $\rightarrow$ $\mathtt{on}$(a,z)]

While the intention of `r2` is to find the `on` relationship between two arbitrary objects, $\mathtt{r2}_a$ is interested in finding the `on` relationship between a particular object `a` and some arbitrary objects. $\mathtt{r2}_b$ has two particular objects in mind, since `on`(a,b) is already satisfied during the migration.

Our point here is that any of these rules may be useful for particular situations in which the rule builder has some particular objects or relations in mind. $\mathtt{r2}_a$ focuses on the object `a`, and $\mathtt{r2}_b$ on the objects `a` and `b` and also the relation `on`(a,b). Note that `r2`, $\mathtt{r2}_a$, and $\mathtt{r2}_b$ have different levels of generality. Different rules at different levels of generality can be migrated from different nestings.

Nested rule representations have not been emphasized in most rule-based reasoning systems, especially in resolution-based systems [Robinson, 1965]. Although the definition of a well-formed formula in the resolution-based system allows nested representations, its effect disappears after those rules are translated into clause form before applying resolution. For instance, in a resolution-based system, `r1` is translated into

$\neg$ `trans`(r) $\vee$ $\neg$ `r`(x,y) $\vee$ $\neg$ `r`(y,z) $\vee$ `r`(x,z)

In this mechanism, there will be no difference among `r1`, $\mathtt{r1}_a$, and $\mathtt{r1}_b$, since all three rules are uniformly transformed to the same clause form. It may be possible to migrate some specific rules here, but which one is useful for a particular domain of application cannot be determined.

3.4 Knowledge Shadowing

Knowledge shadowing is a systematic decision-making process that recognizes unnecessary deduction branches and blocks them from activation when several rules at different specificity levels are available and applicable. Knowledge shadowing takes advantage of knowledge redundancy created in the knowledge migration process, and also consults the information stored in the expertise base to speed up deduction. A sketch of knowledge shadowing was shown in Section 3.1 by using the transitive rule example.

Two main criteria for deciding which rule to choose are (1) the specificity relation among rules represented in the expertise base, and (2) the nature of the query, that is, whether the query is general or specific. More general knowledge is selected for a general query, and more specific knowledge is selected for a specific query.

Knowledge shadowing consults both instance set information and origin set information stored in the expertise base that is updated by the knowledge migration process.

3.4.1 Using Instance Sets and Origin Sets

The instance set log $\mathcal{IL}$ provides instance relationship among rules. For instance, $< \mathbf{r}_s, \sigma > \in \mathcal{I}_{\mathbf{r}_g}$ implies that $\mathbf{r}_s$ is an instance of (or more specific than) $\mathbf{r}_g$. In a case when both $\mathbf{r}_g$ and $\mathbf{r}_s$ are applicable during a deduction, the more general rule $\mathbf{r}_g$ is prevented from being activated, since the more specific rule $\mathbf{r}_s$ produces fewer inference steps than $\mathbf{r}_g$ does. However, there might be some cases where $\mathbf{r}_g$ could generate more answers than $\mathbf{r}_s$ could, and in this situation, we cannot shadow $\mathbf{r}_g$. In fact, whether the more general rule can be shadowed depends on the nature of the query. The generality level of a query is determined by a variable substitution that unifies the query and a consequent of a rule[1]

Our approach is different from previous work such as EBL and knowledge compilation where a system always tries a specific rule first, and backtracks to try general rule if the specific rule fails to solve the problem. We provide systematic decision algorithms about when to use specific rules and when to use general rules so that no information is lost as a result of shadowing.

The information in the origin set log $\mathcal{OL}$ provides propositional dependencies of derived rules and facts. In the transitive rule example, the origin set of `r2` consists of `r1` and `trans`(on), since its derivation was dependent upon them. However, if `trans`(on) is no longer believed in subsequent reasoning, which causes `r2` to be inapplicable, knowledge shadowing should not

[1]This is applicable to backward chaining. Note that in forward chaining where the inference is driven by an asserted fact, the shadowing is straightforward, since you can always shadow the more general rule. So, we only focus on backward chaining.

block `r1` even for a query about *on* relationship. Using only the instance set information would not work since the instance set of `r1` (more precisely $\texttt{r1}_{cq}$) still contains `r2`. Therefore, the main use of origin set information in knowledge shadowing is to examine if an instance in the instance set of a more general rule is still believed in the current belief space.

Knowledge shadowing by using the instance set information and the origin set information can be illustrated by the following example.

Example : Consider the transitive rule example. Assume that the derivation of on(a,c) is complete and `r2` is migrated from `r1`.

$$(\texttt{r1}_{cq}) \qquad \forall x,y,z\ [\texttt{r}(x,y) \wedge \texttt{r}(y,z) \rightarrow \texttt{r}(x,z)]$$
$$(\texttt{r2}) \qquad \forall x,y,z\ [\texttt{on}(x,y) \wedge \texttt{on}(y,z) \rightarrow \texttt{on}(x,z)]$$

As a result of migration, $< \texttt{r2}, \sigma > \in \mathcal{I}_{\texttt{r1}_{cq}}$ with $\sigma = \{on/r\}$, and $O_{\texttt{r2}} = \{\texttt{r1}, \texttt{trans}(\text{on})\}$. We want to find out shadowing conditions under which it is safe to prevent `r1` from being activated to solve a query. The shadowing condition in this transitive rule example is whether or not the query involves the on relation. For example, queries like on(x,y), on(a,y), or on(b,d) satisfy this condition, but queries like ancestor(x,y), supports(a,y), or r(b,d) do not. (Here, x,y, and r are variables.) The most general unifier (*mgu*) that unifies the consequent of the rule and the query can be used to establish this condition. In other words, we can informally say that `r1` can be shadowed when the *mgu* produced by the pattern matching between the consequent of `r1` and the query contains the migrating substitution $\{on/r\}$.

We now compare two situations when the query is on(b,d) or r(b,d), for variable r. The query on(b,d) can be interpreted as "check if the on relation is satisfied between b and d". The query r(b,d) can be interpreted as "find all relations holding between b and d". These two queries make both `r1` (or $\texttt{r1}_{cq}$) and `r2` applicable.

Consider **q** = on(b,d) first. Matching of r(x,z), which is the consequent of $\texttt{r1}_{cq}$, and **q** produces the *mgu* $\phi_1 = \{on/r,\ b/x,\ d/z\}$. Matching of on(x,z), which is the consequent of `r2`, and **q** produces the *mgu* $\phi_2 = \{b/x,\ d/z\}$. Since ϕ_1 contains σ, we can conclude that r(x,z)ϕ_1 = on(x,z)ϕ_2. This implies that everything that can be inferred from `r1` can also be inferred from `r2` for the current query **q**. In this case, `r1` can be shadowed, and on(b,d) can be derived by using `r2` only.

In case of $\mathbf{q}$ = `r`(b,d), the *mgu* between `r`(x,z) and $\mathbf{q}$ is $\phi_1 = \{r/r,\ b/x,\ d/z\}$, and the *mgu* between `on`(x,z) and $\mathbf{q}$ is $\phi_2 = \{on/r,\ b/x,\ d/z\}$. Since ϕ_1 does not include the migrating substitution σ, it can be inferred that `r`(x,z)$\phi_1 \neq$ `on`(x,z)ϕ_2. This implies that a solution that is deducible from `r1` is not always deducible from `r2`. For example, `ancestor`(b,d) can be derived from `r1` if the knowledge base contains `trans`(ancestor), `ancestor`(b,c), and `ancestor`(c,d), but not from `r2`. In this case, `r1` is not shadowed. An interesting question in this situation is whether or not `r2` should be shadowed. There are pros and cons. Certainly, `r1` will produce everything `r2` will, so in this case `r2` is redundant. On the other hand, `r2` will produce its results more quickly. In our system, `r2` is not shadowed. When `r2` is not shadowed, `r1` will not reproduce `r2` since its instance set already contains information about `r2`.

From this observation, a principle of knowledge shadowing that uses the instance set information and the origin set information can be given as follows.

Shadowing Principle 1 Let $\mathbf{r}$ be a rule that is applicable to a query $\mathbf{q}$ at some point during a deduction. Also let ϕ be a most general unifier (*mgu*) between a consequent of $\mathbf{r}$ and $\mathbf{q}$. Then, $\mathbf{r}$ is shadowed from the inference if there exists $< \mathbf{r}_s, \sigma > \in \mathcal{I}_{\mathbf{r}}$ such that $\phi = \sigma\rho$ for some substitution ρ (i.e., ϕ_i contains σ), and $\mathbf{r}_s$ is asserted. □

Satisfying the condition that ϕ_i contains σ indicates that the query is specific since it already contains the information about the migrating substitution σ. In this situation, the more general rule $\mathbf{r}$ does not contribute to a new solution, so it can be blocked. If the condition is not satisfied, the query is meant to solve more general problems, so the more general rule cannot be shadowed.

The following theorem proves that there is no loss of information as a result of shadowing deduction branches according to Shadowing Principle 1.

Theorem 1 For a particular query asked of the knowledge base, any knowledge that can be derived from the rules that are shadowed by Shadowing Principle 1 can also be derived from the rules that are not shadowed.

Proof Suppose a rule $\mathbf{r}_i$ is applicable to a query $\mathbf{q}$, and $\mathbf{r}_j$ is an instance of $\mathbf{r}_i$ satisfying $<\mathbf{r}_j, \sigma> \in \mathcal{I}_{\mathbf{r}_i}$.

Let $\mathbf{r}_i = < ANT_i, NRCQ_i, RCQ_i >$ and $\mathbf{r}_j = < ANT_j, NRCQ_j, RCQ_j >$. Here, a new operator $\odot$ applies a variable substitution to all members in a set.

$$A \odot \sigma \stackrel{\text{def}}{=} \{a\sigma | \forall a \in A\}$$

Then, $ANT_i \odot \sigma = ANT_j$, $NRCQ_i \odot \sigma = NRCQ_j$, and $RCQ_i \odot \sigma = RCQ_j$.

Since $\mathbf{r}_i$ and $\mathbf{r}_j$ are applicable to $\mathbf{q}$, there exist consequents cq_i of $\mathbf{r}_i$ and cq_j of $\mathbf{r}_j$ with $cq_i\sigma = cq_j$. Let ϕ_i be the *mgu* between cq_i and $\mathbf{q}$, and ϕ_j be the *mgu* between cq_j and $\mathbf{q}$. The key step of this proof is to check if any answer A that is deducible from $\mathbf{r}_i$ is also deducible from $\mathbf{r}_j$.

If ϕ_i contains σ, $\phi_i = \sigma\phi_j$, since the *mgu*s are obtained from the same query. Therefore, $cq_i\phi_i = cq_i\sigma\phi_j$, and eventually $cq_i\phi_i = cq_j\phi_j$. From $ANT_i \odot \sigma = ANT_j$, we know that for any $ant_i \in ANT_i$, there exists a unique $ant_j \in ANT_j$ such that $ant_i\sigma = ant_j$, which satisfies $ant_i\sigma\phi_j = ant_j\phi_j$, which in turn yields to $ant_i\phi_i = ant_j\phi_j$. Therefore, for any answer A that can be derived from $\mathbf{r}_i$, there exists a substitution ψ that satisfies $ant_i\phi_i\psi \in \mathcal{F}$ and $cq_i\phi_i\psi = A$, for any $ant_i \in ANT_i$. But, the same substitution ψ also satisfies $ant_j\phi_j\psi \in \mathcal{F}$ and $cq_j\phi_j\psi = A$, for any $ant_j \in ANT_j$. This implies that A is also deducible from $\mathbf{r}_j$. Since every answer deducible from $\mathbf{r}_i$ is also deducible from $\mathbf{r}_j$, $\mathbf{r}_i$ can be shadowed without losing any information.

If ϕ_i does not contain σ, $\phi_i\sigma = \phi_j$ and $cq_i\phi_i\sigma = cq_j\phi_j$. In this case, an answer A that is deducible from $\mathbf{r}_i$ may not be deducible from $\mathbf{r}_j$. This situation happens when A is derived from $\mathbf{r}_i$ by a substitution ψ', i.e., $cq_i\phi_i\psi' = A$, where there exists a binding $c'/v \in \psi'$ and a binding $c/v \in \sigma$ such that $c' \neq c$. Certainly, $cq_j\phi_j\psi' \neq A$, and A is not deducible from $\mathbf{r}_j$. Therefore, $\mathbf{r}_i$ may not be blocked from the fact that this rule might produce a new result that cannot be produced from $\mathbf{r}_j$. $\square$

Traces in SNIP

We present and compare the traces of the transitive rule example in the old SNIP (SNIP 2.1) and the new SNIP (SNIP 2.2) to see the effect of the first shadowing principle.

A SNePSLOG representation is given as below. (Details about the SNePSLOG interface in described in Chapter 5.)

```
all(r) (transitive(r) => {all(x,y,z) ({r(x,y), r(y,z)} &=> {r(x,z)})})
transitive(on)
on(a,b)
on(b,c)
on(c,d)
on(a,c)?
on(b,d)?
```

In SNIP2.1, the following inference trace is obtained. In this trace, '`I wonder if`' represents a request for satisfying a goal, '`I know`' indicates the system found an asserted fact that is matched with a goal, and '`I infer`' indicates the system derives and asserts a new fact.

```
: on(a,c)?

I wonder if  ON(A,C)
I wonder if  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
I wonder if  TRANSITIVE(ON)

I know  TRANSITIVE(ON)
Since  all(R)(TRANSITIVE(R) => (all(X,Y,Z)({R(X,Y),R(Y,Z)} &=> {R(X,Z)})))
and  TRANSITIVE(ON)
I infer  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})

I wonder if  ON(A,Y)
I wonder if  ON(Y,C)
I know  ON(A,B)
I know  ON(B,C)

Since  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
and  ON(A,B)
and  ON(B,C)
I infer  ON(A,C)
```

```
: on(b,d)?

I wonder if  ON(B,D)
I wonder if  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
I wonder if  ON(B,Y)
I wonder if  ON(Y,D)
I know  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
I wonder if  ON(Y,D)
I wonder if  ON(B,Y)

I know  ON(B,C)
I know  ON(C,D)
Since  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
and  ON(B,C)
and  ON(C,D)
I infer  ON(B,D)

I wonder if  TRANSITIVE(ON)
I wonder if  ON(Y,Z)
I wonder if  ON(X,Y)

I know  TRANSITIVE(ON)
Since  all(R)(TRANSITIVE(R) => (all(X,Y,Z)({R(X,Y),R(Y,Z)} &=> {R(X,Z)})))
and  TRANSITIVE(ON)
I infer  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})

Since  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
and  ON(B,C)
and  ON(C,D)
I infer  ON(B,D)
```

In SNIP2.2, the following inference trace is obtained. In fact, the trace for `on(a,c)` is the same as in SNIP 2.1, so we only show the trace for `on(b,d)` here.

```
: on(b,d)?

I wonder if  ON(B,D)
I wonder if  ON(B,Y)
I wonder if  ON(Y,D)

I know  ON(B,C)
I know  ON(C,D)
```

```
Since  all(X,Y,Z)({ON(X,Y),ON(Y,Z)} &=> {ON(X,Z)})
and  ON(B,C)
and  ON(C,D)
I infer  ON(B,D)

    ON(B,D)
```

The inference steps for `on(a,c)?` in both SNIP2.1 and SNIP2.2 are the same, but the inference steps for `on(b,d)?` in SNIP2.2 are much shorter than those in SNIP2.1 by shadowing the generic transitive rule from the inference.

3.4.2 Using Common Instances

Shadowing Principle 1, which we have discussed in the previous section focuses on the relative specificity relationship among several applicable rules. This section discusses a different kind of shadowing by using the *most general common instance* (`mgci`) of two patterns defined as below.

Definition Let ϕ be the *mgu* between two patterns S and T, and $\texttt{mgci}_{ST}$ be the most general common instance of S and T. Then, $\texttt{mgci}_{ST} \stackrel{\text{def}}{=} S\phi$. (Note that $S\phi = T\phi$.)

For example, matching *p(x,b)* and *p(a,y)*, where x and y are variables, produces $\phi = \{a/x, b/y\}$, and the `mgci` of these two is *p(a,b)*. In general, the `mgci` of two patterns is a ground fact when ϕ contains all variables of S and T, and contains no binding "t/v" such that both t and v are variables. We want a shadowing rule that uses the `mgci` of a query and a consequent of the rule that are matched during a deduction. An interesting phenomenon is observed when the `mgci` of two patterns is ground and already asserted in the knowledge base. An example of shadowing using the `mgci` is as follows.

Example : Consider the transitive rule example when the derivation of on(a,c) is complete and `r1` and `r2` coexist.

(r1) $\forall$r [trans(r) $\rightarrow$ $\forall$x,y,z [r(x,y) $\wedge$ r(y,z) $\rightarrow$ r(x,z)]]
(r2) $\forall$x,y,z [on(x,y) $\wedge$ on(y,z) $\rightarrow$ on(x,z)]
(f1) trans(on)
(f2) on(a,b)

(f3) on(b,c)
(f4) on(c,d)
(f5) on(a,c)

Suppose the next query is "find all relations between objects a and c" represented by r(a,c), for variable r. The mgci of r(a,c) and on(x,z) is on(a,c) with the *mgu* $\phi = \{on/r,\ a/x,\ c/z\}$. Since on(a,c) is a ground instance and already is asserted in the knowledge base, r2 will never produce any new answers for the question of r(a,c). In this case, we can block the rule r2 and apply only r1 to find more relations between a and c. Note that the mgci of r(a,c) and r(x,z) is r(a,c) with $\phi = \{r/r,\ a/x,\ c/z\}$. Since r(a,c) is not ground, the branch from r1 is not blocked.

Example : Consider a knowledge base with a rule ∀x [man(x) → mortal(x)] and a fact man(socrates). A query p(socrates), for a variable p, asked of this knowledge base generates two answers man(socrates) and mortal(socrates), and a new fact mortal(socrates) is asserted to the knowledge base. Suppose the same query p(socrates) is asked in a subsequent deduction. Since the mgci of mortal(x) and p(socrates) is mortal(socrates), and it is asserted, the rule is not activated again. Rather, the query is answered by retrieving two facts man(socrates) and mortal(socrates). The same query can be answered more quickly even though the search space has more knowledge.

Note that no deduction is needed if the query itself is ground and asserted. An example is when we ask mortal(socrates) of the above knowledge base. In this case, the mgci of mortal(socrates) and mortal(x) is mortal(socrates), and in the same fashion, the rule ∀x [man(x) → mortal(x)] is shadowed from the inference. This special case can also be found in the transitive rule example when a query on(a,c) is asked again of the knowledge base where both r1 and r2 are applicable. In this case, the mgci of on(a,c) and r(x,z) or the mgci of on(a,c) and on(x,z) is on(a,c). As a result, both rules r1 and r2 are shadowed, and no chaining occurs.

From this observation, a shadowing principle using the mgci is as follows.

Shadowing Principle 2 Let **r** $=< ANT, NRCQ, RCQ >$ be a rule that is applicable to a query **q** at some point during a deduction. (This implies that there is a consequent $cq \in NRCQ$ or $cq \in RCQ$ that is matched with the query **q**.) Then, **r** is shadowed from the inference if the `mgci` of **q** and cq is ground and asserted in the knowledge base. □

Interestingly, if the `mgci` of a rule consequent and a query is P, and P is asserted, the rule will be shadowed even if the rule would produce $\sim P$. Thus, this shadowing prevents inconsistent knowledge bases from making their inconsistencies explicit—they exhibit a form of *cognitive dissonance*. For example, suppose a knowledge base contains two rules ∀x [`man`(x) → `mortal`(x)] and ∀x [`philosopher`(x) → ∼`mortal`(x)], and a fact `man`(socrates). A query `mortal`(socrates) asked of this knowledge base will produce the answer `mortal`(socrates) by activating the rule ∀x [`man`(x) → `mortal`(x)]. Then, suppose we assert a fact `philosopher`(socrates), and ask the same query again. In this case, both rules are shadowed, although the second rule might have produced ∼`mortal`(socrates) if it had been activated.

The following theorem proves that in a consistent knowledge base there is no loss of information as a result of shadowing deduction branches according to Shadowing Principle 2.

Theorem 2 For a particular query asked of a consistent knowledge base, any knowledge that can be derived from the rules that are shadowed by Shadowing Principle 2 can also be derived from the rules that are not shadowed.

Proof Let ϕ be the *mgu* between **q** and cq. Also let the `mgci` of **q** and cq be A which is equal to $\mathbf{q}\phi$ or $cq\phi$. By the assumption of Shadowing Principle 2, A is asserted, so A is unique, and, since the knowledge base is consistent, $\sim A$ is not derivable. This implies that the only answer we can derive from **r** for the query **q** is A. Since A is already in the knowledge base, **r** does not produce any new result, so it can be shadowed without losing any information. □

In an inconsistent knowledge base, the only information that can be lost is the explicit derivation of $\sim A$.

These theorems have proved the completeness of the shadowing principles by showing that any deduction problem that could be solved without the shadowing methods can also be solved with them except, possibly, for the explicit production of contradictions.

Traces in SNIP

This section presents two simple problems to see the effect of Shadowing Principle 2.

Problem 1:

The knowledge base for the first problem represented in SNePSLOG is :

```
all(x) (man(x)=> mortal(x))
all(x) (greek(x) => philosopher(x))
man(Socrates)
```

Then, the following trace is obtained for the query `?P(socrates)` (`?P` denotes a variable). Note that the same trace is obtained from SNIP 2.1 and SNIP 2.2.

```
: ?P(socrates)?

I wonder if  ?P(SOCRATES)
I know  MAN(SOCRATES)

I wonder if  GREEK(SOCRATES)
I wonder if  MAN(SOCRATES)

I know  MAN(SOCRATES)
Since  all(X)(MAN(X) => MORTAL(X))
and  MAN(SOCRATES)
I infer  MORTAL(SOCRATES)

  MAN(SOCRATES)
  MORTAL(SOCRATES)
```

Now we add a new fact `greek(socrates)` to the knowledge base and ask the query `?P(socrates)` again. In SNIP 2.2, the rule `all(x) (man(x)=> mortal(x))` is shadowed, since the `mgci` of `?P(socrates)` and `mortal(x)` is `mortal(socrates)`, and it is ground and asserted. As a result, SNIP 2.1 and SNIP 2.2 produce different traces. In SNIP 2.1, the following trace is obtained.

```
: ?P(socrates)?

I wonder if  ?P(SOCRATES)
```

```
I know  MORTAL(SOCRATES)
I know  GREEK(SOCRATES)
I know  MAN(SOCRATES)

I wonder if  GREEK(SOCRATES)
I wonder if  MAN(SOCRATES)

I know  GREEK(SOCRATES)
Since  all(X)(GREEK(X) => PHILOSOPHER(X))
and  GREEK(SOCRATES)
I infer  PHILOSOPHER(SOCRATES)

I know  MAN(SOCRATES)
Since  all(X)(MAN(X) => MORTAL(X))
and  MAN(SOCRATES)
I infer  MORTAL(SOCRATES)

  MAN(SOCRATES)
  GREEK(SOCRATES)
  MORTAL(SOCRATES)
  PHILOSOPHER(SOCRATES)
```

In SNIP 2.2, the following trace is obtained that has shorter inference steps by shadowing the rule `all(x) (man(x)=> mortal(x))`.

```
: ?P(socrates)?

I wonder if  ?P(SOCRATES)
I know  MORTAL(SOCRATES)
I know  GREEK(SOCRATES)
I know  MAN(SOCRATES)

I wonder if  GREEK(SOCRATES)

I know  GREEK(SOCRATES)
Since  all(X)(GREEK(X) => PHILOSOPHER(X))
and  GREEK(SOCRATES)
I infer  PHILOSOPHER(SOCRATES)

    MAN(SOCRATES)
    PHILOSOPHER(SOCRATES)
    GREEK(SOCRATES)
    MORTAL(SOCRATES)
```

Problem 2

The knowledge base of the second problem is:

```
all(x) (cousin(x,john) => cousin(x,pete))
cousin(chris,john)
cousin(chris,pete)
```

In SNIP 2.2, the rule `all(x) (cousin(x,john) => cousin(x,pete))` is not activated at all if the query is `cousin(chris,?y)`, `y` being a variable. The reason is that the `mgci` of `cousin(chris,?y)` and `cousin(x,pete)` is `cousin(chris,pete)`, but it is already asserted in the knowledge base.

A trace in SNIP 2.1 is as follows.

```
: cousin(chris,?y)?

I wonder if  COUSIN(CHRIS,?Y)
I know  COUSIN(CHRIS,PETE)
I know  COUSIN(CHRIS,JOHN)

I wonder if  COUSIN(CHRIS,JOHN)
I know  COUSIN(CHRIS,JOHN)
Since  all(X)(COUSIN(X,JOHN) => COUSIN(X,PETE))
and  COUSIN(CHRIS,JOHN)
I infer  COUSIN(CHRIS,PETE)

  COUSIN(CHRIS,JOHN)
  COUSIN(CHRIS,PETE)
```

In contrast, the trace in SNIP 2.2 is as follows.

```
: cousin(chris,?y)?

I wonder if  COUSIN(CHRIS,?Y)
I know  COUSIN(CHRIS,PETE)
I know  COUSIN(CHRIS,JOHN)

  COUSIN(CHRIS,JOHN)
  COUSIN(CHRIS,PETE)
```

3.5 Analysis of Knowledge Migration and Knowledge Shadowing

In this section, we analyze the space and time complexity of the knowledge migration and the knowledge shadowing schemes, and also discuss the performance enhancement resulting from applying the shadowing principles. From these analyses, we will show how efficient these learning methods are and under what circumstances they will be effective.

3.5.1 Space Complexity

System space is mainly affected by the knowledge migration process that expands both the knowledge base and the expertise base.

One of the assumptions in our system regarding knowledge base management was that derived knowledge is monotonically and *unconditionally* asserted to the knowledge base. This might bring up the issue of *unconditional* versus *selective* assertion of derived knowledge.

The main advantage of unconditional assertion is the simplicity and the speed of the assertion mechanism because it does not involve any testing for the usefulness of newly derived knowledge. However, this unconditional assertion causes not only a space problem by monotonically expanding the knowledge base, but also an increased search time due to additional branching factors in the search space that slows down the system.

Alternatively, *selective assertion* asserts derived knowledge only when it is determined to be useful. Although it might be effective in reducing the data space and also the search space, it is generally very difficult to build a decision procedure for selective assertion, and consequently its application is limited. In most cases, predicting the usefulness of new knowledge in future reasoning is impossible. Therefore, instead of spending time to determine which knowledge is useful, we decided to compensate for the deficiency of unconditional assertion by providing systematic ways of selecting appropriate knowledge in the search process. Knowledge shadowing was developed for this purpose.

Another factor that affects space complexity is the storage of deduction expertise in the expertise base. Knowledge migration enlarges the expertise base by expanding instance sets and origin sets with migration information. However, we expect the inclusion of the expertise base does not cause a serious space problem since the size of each unit in the instance set or the origin set is relatively small and the cardinality of each set is proportional to the number of rules in the knowledge base.

3.5.2 Time Complexity

In knowledge shadowing, a major portion of time is spent retrieving information from the expertise base and comparing it with the current situation. The cost of retrieving and comparing the expertise information is denoted by $C_{\mathcal{EB}}$.

Conceptually, an instance set is a (mathematical) set, and the cost of checking if an instance is in the instance set of a rule has a linear time complexity in terms of average number of instances in an instance set. This linear set might cause a speed problem as the size of an instance set increases. Note that each instance in a set has a unique migrating substitution, and from this fact, we can implement the process of storing and retrieving instance information by using an indexing mechanism that has constant time complexity regardless of the size of the instance set. (See the **S-indexing** algorithm in Chapter 4.)

3.5.3 Analysis of Performance Enhancement

This section discusses the performance enhancement that results from the combination of knowledge migration and the knowledge shadowing methods. We also try to predict some conditions under which knowledge shadowing will be effective.

It is generally difficult to calculate the cost of a rule execution in a natural deduction system, since it depends on several factors including the content of the knowledge base and the specific implementation of inference rules, especially whether it is sequential or parallel. We try to estimate the cost of a rule execution approximately by two metrics: (1) the cost of satisfying each antecedent in a rule and (2) the cost of resolving binding conflicts of shared variables among all antecedents. Satisfying an antecedent is normally done by pattern matching to see if there is any asserted fact that matches the antecedent, and if there is, a substitution that unifies the pattern with the fact will be returned. If no fact is matched with the current antecedent pattern, it initiates a rule chaining by checking if the current antecedent is matched with a consequent of a rule. Otherwise, the antecedent is not satisfied. So the average cost of satisfying an antecedent pattern A denoted by $C_{PAT}(A)$ will be

$$C_{PAT}(A) \approx C_{MATCH}(A) + P_{fact}(A) \cdot C_{RET}(A) + P_{pat}(A) \cdot C_{RC}(A)$$

where $C_{MATCH}(A)$ is the cost of pattern matching for A, $P_{fact}(A)$ is the probability that there exists a fact that is matched with A, $C_{RET}(A)$ is the cost of retrieving all facts matched with A, $P_{pat}(A)$ is the probability that there exists a rule whose consequent is matched with A, and $C_{RC}(A)$ is the cost of rule chaining to satisfy A. Here, $1 - P_{fact}(A) - P_{pat}(A)$ is the probability that A is not satisfied by the current knowledge base.

After satisfying all antecedent patterns, we then have to find a substitution, if any, that satisfies all antecedents in the rule. This process is called binding conflict resolution, and the cost of this process is denoted by $C_{BCR}(\mathbf{r})$ for a rule $\mathbf{r}$. In a naive method, $C_{BCR}(\mathbf{r})$ has the complexity of $O(m^2 \cdot 2^n)$ on average, where m is the average number of instances for each antecedent pattern, and n is the average number of antecedent patterns in a rule. (Details are in Chapter 4.)

For example, consider the following nested rule $\mathbf{r}_g$.

$$(\mathbf{r}_g) \quad A_1 \wedge A_2 \wedge \cdots \wedge A_m \rightarrow (B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow C)$$

So the total cost to execute $\mathbf{r}_g$ denoted by $C_{RULE}(\mathbf{r}_g)$ can be approximately estimated as below:

$$\begin{aligned} C_{RULE}(\mathbf{r}_g) \approx\ & C_{PAT}(A_1) + C_{PAT}(A_2) + \cdots + C_{PAT}(A_m) + C_{BCR}(\mathbf{r}_g) \\ & + C_{PAT}(B_1) + C_{PAT}(B_2) + \cdots + C_{PAT}(B_n) + C_{BCR}((\mathbf{r}_g)_{cq}) \end{aligned}$$

Here, $(\mathbf{r}_g)_{cq}$ is the consequent of $\mathbf{r}_g$. Satisfying all antecedents of $\mathbf{r}_g$ results in the generation of a specific rule $\mathbf{r}_s$ by the knowledge migration process.

$$(\mathbf{r}_s) \quad B_1\sigma \wedge B_2\sigma \wedge \cdots \wedge B_n\sigma \rightarrow C\sigma$$

Here, σ is a migrating substitution such that $A_i\sigma$ $(1 \leq i \leq m)$ is satisfiable. The cost to execute $\mathbf{r}_s$ is

$$C_{RULE}(\mathbf{r}_s) \approx C_{PAT}(B_1\sigma) + C_{PAT}(B_2\sigma) + \cdots + C_{PAT}(B_n\sigma) + C_{BCR}(\mathbf{r}_s)$$

Now consider a situation where both $\mathbf{r}_g$ and $\mathbf{r}_s$ are applicable.

In a sequential environment, the total cost is approximately $C_{RULE}(\mathbf{r}_g) + C_{RULE}(\mathbf{r}_s)$ without shadowing because both rules are executed in sequence. With the shadowing method, this cost is reduced to $C_{RULE}(\mathbf{r}_s) + C_{\mathcal{EB}}$ when $\mathbf{r}_g$ is shadowed. $C_{\mathcal{EB}}$ refers to the cost of retrieving the expertise information and comparing it with the current situation, as mentioned before. Therefore, shadowing will be effective when $C_{\mathcal{EB}}$ is much smaller than $C_{RULE}(\mathbf{r}_g)$.

In a parallel environment where $\mathbf{r}_g$ and $\mathbf{r}_s$ are executed concurrently, the total cost without shadowing is approximately $C_{RULE}(\mathbf{r}_g)$, since $C_{RULE}(\mathbf{r}_g)$ is generally greater than $C_{RULE}(\mathbf{r}_s)$. With shadowing, if $\mathbf{r}_g$ is shadowed, the total cost becomes $C_{RULE}(\mathbf{r}_s) + C_{\mathcal{EB}}$ and you can save approximately $C_{PAT}(A_1) + C_{PAT}(A_2) + \cdots + C_{PAT}(A_m) + C_{BCR}(\mathbf{r}_g)$.

In general, we can predict from the above discussion that the shadowing method will be effective in situations where a nested rule has a large number of antecedent patterns, or satisfying antecedents in the rule is complex with long rule chainings and a large number of pattern matchings.

Chapter 4

Learning at the Rule Activation Level

Besides proper rule selection in a redundant knowledge base discussed in the previous chapter, another issue in improving deduction performance will be how to efficiently execute each selected rule itself. In rule-based systems, a rule is activated when newly added data matches an antecedent of a rule (forward chaining), or when the query matches a consequent of a rule (backward chaining). When all antecedents of the rule are satisfied, the rule triggers and may fire. In natural deduction systems, this rule firing corresponds to the execution of a rule of inference. Efficient rule execution is particularly important in a situation where a rule contains a large number of antecedent patterns (with possibly many shared variables among antecedents), and each antecedent is matched with a large number of instances. In these situations, the execution of the rule of inference on a rule might nullify the efficiency obtained by the rule selection procedure due to a combinatorial number of rule activation steps including binding conflict resolution of shared variables.

A solution to this problem is to cache previous rule activation steps and reuse them when the same rule is reactivated in subsequent deductions. For this purpose, a data structure named "rule use information" (RUI) has been defined and used in the SNePS knowledge representation and reasoning system [Hull, 1986]. A set of RUIs is maintained for each rule not only in order to store instances of antecedents, but also to resolve binding conflicts of shared variables among antecedents and combine those RUIs that have consistent substitutions. The design of RUIs was originally motivated by several reasoning purposes in SNePS including the implementation of non-standard connectives and quantifiers [Shapiro, 1979c], the prevention of infinite loops with recursive rules [McKay and Shapiro, 1981], and the efficient manipulation of bi-directional inference [Shapiro *et al.*, 1982]. In this dissertation, the information stored in the RUI structure is regarded as a type of experience, and will be used in subsequent reasoning for performance enhancement.

One of the problems in the implementation of the RUI set in SNIP2.1 is the lack of efficiency caused by the linear set implementation of accumulating rule activation steps. This linear implementation has an exponential complexity with a combinatorial number of substitution compatibility checks between RUIs, which may cause a bottleneck in reasoning as the problem size increases. In SNIP2.2, new algorithms are designed and implemented that distribute the information of rule activation steps to reduce the processing complexity to polynomial.

This chapter is organized as follows. First, we review the concept of non-standard connectives and quantifiers that has motivated the use of the RUI structure. Second, we review the RUI structure in detail and explain how it is used in saving instances of antecedents and resolving binding conflicts of shared variables. Third, we present two new algorithms, **S-indexing** and **P-tree**, for efficient processing of the RUI structure. Formal complexity analyses are done for both mechanisms.

4.1 Non-Standard Connectives and Quantifiers

Non-standard connectives and quantifiers are generalizations of the common connectives and quantifiers such as conjunction, disjunction, negation, implication, universal quantifier, and existential quantifier [Shapiro, 1979c]. They are designed to provide closeness to human reasoning, structural simplicity, and expressibility in the areas of natural language understanding, knowledge representation, and reasoning [Martins and Shapiro, 1988, Shapiro, 1979a, Shapiro, 1979b]. Some of the connectives and quantifiers are briefly introduced below.

And-or is symbolized as ${}_n\bowtie_i^j$, and the formula ${}_n\bowtie_i^j\ (A_1, A_2, \ldots, A_n)$ is true when at least i and at most j of the n arguments are true. *And-or* generalizes conjunction, disjunction, negation, exclusive-or, nor, nand, etc. For instance, ${}_2\bowtie_2^2$(A,B) denotes *A AND B*, ${}_2\bowtie_1^2$(A,B) denotes *A OR B*, ${}_1\bowtie_0^0$(A) denotes *NOT A*, and ${}_2\bowtie_1^1$(A,B) denotes *A XOR B*. Two possible inference rules for the *and-or* connective are, (1) if it is known that exactly j of the arguments are true, then the remaining $n-j$ arguments must be false, and (2) if it is known that exactly $n-i$ of the arguments are false, then the remaining i arguments must be true.

Thresh denoted by ${}_n\Theta_i^j$ is the dual of *and-or*, and the formula ${}_n\Theta_i^j\ (A_1, A_2, \ldots, A_n)$ is true when either fewer than i or more than j of the n arguments are true. If j is omitted, j is automatically set to $n-1$. *Thresh* is mainly used to represent the equivalence relation denoted by ${}_n\Theta_1\ (A_1, A_2, \ldots, A_n)$ which indicates that the arguments are either all true or all false.

Or-entailment ($\vee\!\!\rightarrow$) and *and-entailment* ($\&\!\!\rightarrow$) are the generalizations of implication ($\rightarrow$). The formula $(A_1, A_2, \ldots, A_n)\ \vee\!\!\rightarrow(C_1, C_2, \ldots, C_m)$ means that the disjunction of the antecedents implies the conjunction of the consequents, and the formula $(A_1, A_2, \ldots, A_n)$ $\&\!\!\rightarrow(C_1, C_2, \ldots, C_m)$ means that the conjunction of the antecedents implies the conjunction of the consequents. These two entailments may be combined into one by using *numerical entailment* ($\xrightarrow{i}$). The formula $(A_1, A_2, \ldots, A_n) \xrightarrow{i}(C_1, C_2, \ldots, C_m)$ means that if any i of the antecedents are true, so are all of the consequents.

Numerical quantifier represented by ${}_n\exists_i^j$ generalizes the universal and existential quantifiers. The formula ${}_n\exists_i^j(\bar{x})(P_1(\bar{x}), \ldots, P_k(\bar{x}) : Q(\bar{x}))$, where $\bar{x}$ is a sequence of variables, means that of the n individuals that satisfy $\mathrm{P}_1(\bar{x}) \wedge \ldots \wedge \mathrm{P}_k(\bar{x})$, at least i and at most j of these will also satisfy $\mathrm{Q}(\bar{x})$. For instance, ${}_5\exists_2^2$*(w) (Woman(w): Isa(w, Teacher))* says that there are five women, and exactly two of them are teachers. This kind of rule is especially used for reasoning by exclusion, i.e. if it is already known that two women are teachers, it is inferred that the remaining three women are not teachers.

Introduction and elimination inference rules for these connectives and quantifiers are explained in [Martins and Shapiro, 1988].

As mentioned above, the main advantages of using non-standard connectives and quantifiers for the representation of rules are expressibility and closeness to human reasoning. As an example, consider the representation of the following two statements in the Freeman puzzle described in [Summers, 1972].

> There are five women: Ada, Bea, Cyd, Deb, and Eve.
> The women are in two age brackets: three women are under 30 and two women are over 30.

With standard connectives such as negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$), the phrase *three women are under 30* would have to be expressed in the following way:

$$
\begin{aligned}
&(\neg\mathrm{age(Ada,u30)} \wedge \neg\mathrm{age(Bea,u30)} \wedge \mathrm{age(Cyd,u30)} \wedge \mathrm{age(Deb,u30)} \wedge \mathrm{age(Eve,u30)})\\
\vee\ &(\neg\mathrm{age(Ada,u30)} \wedge \mathrm{age(Bea,u30)} \wedge \neg\mathrm{age(Cyd,u30)} \wedge \mathrm{age(Deb,u30)} \wedge \mathrm{age(Eve,u30)})\\
\vee\ &(\neg\mathrm{age(Ada,u30)} \wedge \mathrm{age(Bea,u30)} \wedge \mathrm{age(Cyd,u30)} \wedge \neg\mathrm{age(Deb,u30)} \wedge \mathrm{age(Eve,u30)})\\
\vee\ &(\neg\mathrm{age(Ada,u30)} \wedge \mathrm{age(Bea,u30)} \wedge \mathrm{age(Cyd,u30)} \wedge \mathrm{age(Deb,u30)} \wedge \neg\mathrm{age(Eve,u30)})
\end{aligned}
$$

∨ (age(Ada,u30) ∧ ¬age(Bea,u30) ∧ ¬age(Cyd,u30) ∧ age(Deb,u30) ∧ age(Eve,u30))
∨ (age(Ada,u30) ∧ ¬age(Bea,u30) ∧ age(Cyd,u30) ∧ ¬age(Deb,u30) ∧ age(Eve,u30))
∨ (age(Ada,u30) ∧ ¬age(Bea,u30) ∧ age(Cyd,u30) ∧ age(Deb,u30) ∧ ¬age(Eve,u30))
∨ (age(Ada,u30) ∧ age(Bea,u30) ∧ ¬age(Cyd,u30) ∧ ¬age(Deb,u30) ∧ age(Eve,u30))
∨ (age(Ada,u30) ∧ age(Bea,u30) ∧ ¬age(Cyd,u30) ∧ age(Deb,u30) ∧ ¬age(Eve,u30))
∨ (age(Ada,u30) ∧ age(Bea,u30) ∧ age(Cyd,u30) ∧ ¬age(Deb,u30) ∧ ¬age(Eve,u30))

Compared to this representation, we can obtain a simpler and more readable expression by using a non-standard connective *and-or* as follows:

$${}_{5}{\bigvee\!\!\!\!\!\bigwedge}_{3}^{3}\ (\mathrm{age(Ada,u30),\ age(Bea,u30),\ age(Cyd,u30),\ age(Deb,u30),\ age(Eve,u30)})$$

With *numerical quantifier*, this is represented more compactly as follows:

$${}_{5}\exists_{3}^{3}\ (\mathrm{x})\ (\mathrm{woman(x){:}\ age(x,u30)})$$

These expressions with non-standard connectives and quantifiers are certainly readable and naturally expressible compared to those with standard connectives. However, the system may be slowed down in order to directly activate these non-standard rules, since the manipulation of their rules of inference is complicated. Our general objective is to develop a method that can efficiently process the rules of inference of non-standard connectives and quantifiers, and consequently to achieve both expressibility and performance.

In fact, each of the non-standard connectives and quantifiers has different rules of inference depending on how many antecedents must be consistently instantiated (by positive or negative instances) to draw conclusions. For instance, *and-entailment* requires all the antecedents of a rule be instantiated to deduce its consequents, but *or-entailment* requires at least one of them be instantiated. Also, the *and-or* connective ${}_{n}{\bigvee\!\!\!\!\!\bigwedge}_{i}^{j}$ has many different rules of inference depending on the values of the parameters n, i, and j. A uniform way of manipulating these various kinds of rules of inference can be achieved by using the RUI set structure that represents each rule activation step and maintains the instance information including variable substitutions and the number of antecedents satisfied by positive or negative instances.

4.2 Rule Use Information (RUI)

The RUI structure has been proposed and implemented in SNePS [Hull, 1986, McKay and Shapiro, 1981, Shapiro, 1977] to save instance information of the antecedents of a rule including variable substitutions and the number of positive and negative instantiations, and also to combine those instances that have consistent bindings for shared variables. A rule is associated with a set of RUIs called a "RUI set" that traces the history of instance handling. The current implementation in SNIP2.1 maintains a linear RUI set for each rule.

4.2.1 Data Structure of RUI

A RUI consists of 4 elements as below,

```
<RUI> ::= ( <sbst> <pcount> <ncount> <fns> )
```

where `<sbst>` denotes variable substitutions from instances of antecedents, `<pcount>` denotes the number of antecedents known to be true, `<ncount>` denotes the number of antecedents known to be false, and a flagged node set `<fns>` indicates which antecedents are known to be true or false.

As an example, consider a knowledge base for reasoning about kinship facts. It might have an *and-entailment* deduction rule such as $\mathbf{r}_{husband}$ for recognizing the husband relationship.

($\mathbf{r}_{husband}$) ∀x,y [`man`(x), `woman`(y), `married`(x,y) &→`husband`(x,y)]

$\mathbf{r}_{husband}$ says that if there is a man x and a woman y, and they are married, x is the husband of y. Suppose we want to derive all of the husband relationships from the following set of facts through backward chaining.

`man`(john)
`man`(fred)
`man`(bob)
`man`(steve)
`woman`(mary)
`woman`(jane)
`woman`(deb)
`woman`(ada)

woman(sue)
married(steve, sue)

A query husband(x,y) asked to this knowledge base invokes pattern matching procedures to find instances of antecedent patterns. In this case, the pattern man(x) has 4 instances, woman(y) has 5 instances, and married(x,y) has 1 instance. Initially, the RUI set of a rule is empty, and is dynamically augmented as new instances of antecedents are processed. Figure 4.1 shows a resulting RUI set of the rule $\mathbf{r}_{husband}$ after processing all the above instances. (P1, P2, and P3 denote man(x), woman(y), and married(x,y), respectively). Here, r1 is created from the instance man(john), r2 is created from the instance man(fred), and r5 is created from the instance woman(mary), and so on. Note that the same RUI set, except for the order of RUIs, will result no matter which antecedent is processed first.

4.2.2 Resolving Binding Conflicts by Using RUI

One peculiar characteristic of the RUI set is that it not only accumulates the instances of each antecedent, but it also maintains combined RUIs by merging any two RUIs that have no binding conflicts. Resolving binding conflicts of shared variables is done by checking substitution compatibility between RUIs. Two RUIs are said to be *compatible* when the substitutions of the two RUIs are consistent, meaning that a shared variable in both substitutions is bound to the same value. For example, r4 in Figure 4.1 is compatible with r30, but not compatible with r1. Note that those RUIs with no shared variables are always compatible, for instance, r1 and r5 are compatible.

Any two RUIs in a RUI set that have compatible substitutions and disjoint <fns> are combined to create a merged RUI. Suppose <fns> of a RUI r_a is f_a, and <fns> of a RUI r_b is f_b. These two RUIs can be combined to create a new merged RUI r_m when (1) r_a and r_b are compatible, and (2) the node set of f_a is disjoint with the node set of f_b. As a result, <pcount> of r_m is the sum of <pcount> of r_a and <pcount> of r_b, and <ncount> of r_m is the sum of <ncount> of r_a and <ncount> of r_b. Also, <sbst> of r_m is a union of <sbst> of r_a and <sbst> of r_b, and <fns> of r_m is the union of f_a and f_b. For instance, r1 and r5 are combined to produce r6, and r29 and r30 are merged to get r33. Eventually, husband(steve,sue) is derived

```
(
r1   ( {john/x} 1 0 {P1:true} )
r2   ( {fred/x} 1 0 {P1:true} )
r3   ( {bob/x} 1 0 {P1:true} )
r4   ( {steve/x} 1 0 {P1:true} )
r5   ( {mary/y} 1 0 {P2:true} )
r6   ( {john/x, mary/y} 2 0 {P1:true, P2:true} )
r7   ( {fred/x, mary/y} 2 0 {P1:true, P2:true} )
r8   ( {bob/x, mary/y} 2 0 {P1:true, P2:true} )
r9   ( {steve/x, mary/y} 2 0 {P1:true, P2:true} )
r10  ( {jane/y} 1 0 {P2:true} )
r11  ( {john/x, jane/y} 2 0 {P1:true, P2:true} )
r12  ( {fred/x, jane/y} 2 0 {P1:true, P2:true} )
r13  ( {bob/x, jane/y} 2 0 {P1:true, P2:true} )
r14  ( {steve/x, jane/y} 2 0 {P1:true, P2:true} )
r15  ( {deb/y} 1 0 {P2:true} )
r16  ( {john/x, deb/y} 2 0 {P1:true, P2:true} )
r17  ( {fred/x, deb/y} 2 0 {P1:true, P2:true} )
r18  ( {bob/x, deb/y} 2 0 {P1:true, P2:true} )
r19  ( {steve/x, deb/y} 2 0 {P1:true, P2:true} )
r20  ( {ada/y} 1 0 {P2:true} )
r21  ( {john/x, ada/y} 2 0 {P1:true, P2:true} )
r22  ( {fred/x, ada/y} 2 0 {P1:true, P2:true} )
r23  ( {bob/x, ada/y} 2 0 {P1:true, P2:true} )
r24  ( {steve/x, ada/y} 2 0 {P1:true, P2:true} )
r25  ( {sue/y} 1 0 {P2:true} )
r26  ( {john/x, sue/y} 2 0 {P1:true, P2:true} )
r27  ( {fred/x, sue/y} 2 0 {P1:true, P2:true} )
r28  ( {bob/x, sue/y} 2 0 {P1:true, P2:true} )
r29  ( {steve/x, sue/y} 2 0 {P1:true, P2:true} )
r30  ( {steve/x, sue/y} 1 0 {P3:true} )
r31  ( {steve/x, sue/y} 2 0 {P1:true, P3:true} )
r32  ( {steve/x, sue/y} 2 0 {P2:true, P3:true} )
r33  ( {steve/x, sue/y} 3 0 {P1:true,P2:true,P3:true})
)
```

Figure 4.1: A RUI set of the rule $\mathbf{r}_{husband}$

from r33 whose <pcount> is the same as the number of antecedents of the rule, according to the rule of inference for *and-entailment*.

4.2.3 Advantages of the RUI Structure

The advantages of employing the RUI set structure in deductive problem solving can be described in 4 ways.

First, non-standard rules of inference can be uniformly implemented. Uniformity in manipulating various kinds of rules of inference can be achieved by exploiting <pcount>, <ncount>, and <fns> fields of the RUI structure. For example, an *and-entailment* rule can deduce a consequent if there is a RUI whose <pcount> value equals the number of antecedents in the rule. Also, two rules of inference for the *and-or* rule ${}_n\bowtie_i^j\ (A_1, A_2, \ldots, A_n)$ can be stated as, (1) if there is a RUI whose <pcount> value is equal to j, the <sbst> instances of those arguments that are not in its <fns> field are proved to be false, and (2) if there is a RUI whose <ncount> value is equal to $n - i$, the <sbst> instances of those arguments not in its <fns> fields are proved to be true. In the same fashion, expressions like *exactly two of n arguments*, *not all of n arguments*, or *at least 3 of n arguments* can also be expressed easily.

Second, we can reuse the previous history of rule activation steps saved in the RUI set for subsequent deductions on the same rule. Duplicate rule activation steps, including pattern matchings and binding conflict resolutions, are avoided. This reusability can contribute to better performance for future deductions, which is the main purpose of speedup learning. For example, suppose a new fact married(john,mary) is added to the knowledge base to find more husband relationships. If the system does not maintain any mechanism for remembering instances, all the rule execution steps must be repeated. This includes pattern matchings for man(x), woman(y), and married(x,y), even though we have already obtained most of instances in the previous inference. Also the same steps of binding conflicts resolution must be done again with little additional information, and most steps are repetitions of previous steps. Duplicate pattern matching can be avoided if the instances of each antecedent pattern are saved separately, but substitution compatibility tests must still be done to get consistent combinations of instances. Using the RUI set, all these duplicate rule deduction steps are avoided. In the above case, all we have to do now is to create a new RUI `r34` for the instance `married`(john,mary), and check substitution compatibility between `r34` and each of the RUIs in the `RUI` set of $\mathbf{r}_{husband}$ to produce the following merged RUIs.

`r34`	({john/x, mary/y} 1 0 {P3:true})
`r35`	({john/x, mary/y} 2 0 {P1:true,P3:true})
`r36`	({john/x, mary/y} 2 0 {P2:true,P3:true})
`r37`	({john/x, mary/y} 3 0 {P1:true,P2:true,P3:true})

`r34` is combined with `r1`, `r5`, and `r6` to generate `r35`, `r36`, and `r37`, respectively. `r37` contributes to the derivation of `husband`(john,mary).

Third, the RUI set structure facilitates complete AND-parallelism. A major issue in the AND-parallel computation of a conjunctive rule is the overhead of resolving binding conflicts of shared variables, since it is almost impossible to execute two patterns with shared variables concurrently. This problem mainly results from trying to resolve the conflicts at the pattern level. By maintaining a separate process for a rule that maintains a RUI set structure, the only thing that each antecedent pattern process should do is save instances that are matched and send them to the rule process without worrying about the compatibility of shared variables. This step of checking the substitution compatibility is performed by the rule process that receives instance information from all antecedent processes. There is no need to order antecedent patterns according to variable specifications [Conery and Kibler, 1983], and no need to have complex communication mechanisms among antecedents.

Fourth, the RUI set structure is used to facilitate the implementation of bi-directional inference which combines backward and forward chaining [Shapiro *et al.*, 1982]. In fact, we have already witnessed a case of bi-directional inference in the husband rule example. In other words, the first query `husband`(x,y) is processed by backward chaining, and the second deduction is carried out by adding a new fact `married`(john,mary) that initiates forward chaining to find more husband relationships. Also the RUI set is used to implement recursive rule inference without causing infinite loops [McKay and Shapiro, 1981].

4.2.4 Efficiency Issues

One major problem in maintaining the RUI structure is efficiency. In general, reasoning by maintaining a linear RUI set for each rule is intractable due to the combinatorial number of substitution compatibility checks between RUIs. The number of RUIs in a set also become large as the application problem size increases. Furthermore, many unnecessary RUIs are merged between patterns that share no common variables, such as `man`(x) and `woman`(y), because the compatibility checks between RUIs of these patterns always succeed. It is recognized that updating a linear RUI set has an exponential complexity, in average, in terms of the number of antecedents in a rule. (Details are in Section 4.5).

In this dissertation, techniques for distributing RUIs are presented for fast reasoning with deduction rules, especially with non-standard connectives and quantifiers. Two algorithms are designed for efficient RUI handling in this distributed RUI set scheme.

The first algorithm, called **S-indexing** (**S** stands for *substitution*), is designed for rules in which all antecedent patterns or all arguments have the same set of variables, for example, ${}_{n}\bigvee\!\!\!\bigwedge_{i}^{j}\ (A(x,y), B(x,y), C(x,y))$. In **S-indexing**, RUIs are indexed and distributed by bound values for variables. A single RUI is built for each variable substitution.

The second algorithm, called **P-tree** (**P** stands for *pattern*), is designed for conjunctive rules whose antecedent patterns have different sets of variables. A conjunctive rule has two or more antecedents which must be satisfied simultaneously with consistent variable bindings to derive the consequents. Rules with the *and-entailment* connective often belong to this category. In this method, a **P-tree** is built from the set of antecedent patterns of a conjunctive rule so that those patterns with common variables can be arranged to be adjacent in the tree. RUIs are distributed over the nodes of the **P-tree**.

One exception is that rules with the *numerical quantifier* are handled by the **P-tree** method even if all antecedents have the same set of variables. The main reason for this is that *numerical quantifier* expressions have the characteristics of both conjunctiveness and non-conjunctiveness, and therefore **S-indexing** may not handle the rule of inference for *numerical quantifier* expressions correctly.

In summary, the **S-indexing** method is applied to those rules, except *numerical quantifier* rules, in which the antecedents (or arguments) have the same set of variables, whereas the **P-tree** method is applied to all *numerical quantifier* rules and those *and-entailment* rules whose antecedents have different sets of variables. The linear method is applied to those rules that do not belong to the above categories.

Complexity analyses presented in Section 4.5 show that these schemes of RUI set distribution keep the complexity of the RUI set handling to polynomial in terms of the number of RUIs and the number of substitution compatibility tests.

4.3 S-indexing Algorithm

S-indexing is designed for rules in which all antecedent patterns or all arguments have the same set of variables. For instance, an expression *a person is either a man or a woman* can be represented by $\forall$x [person(x) $\lor\!\!\rightarrow {}_2\mathbb{X}_1^1$ (man(x), woman(x))].

S-indexing distributes RUIs by bound values for a fixed set of variables in a rule. An index key is represented by $< b_1, \ldots, b_m >$, generated from a substitution $\{b_1/v_1, \ldots, b_m/v_m\}$. Here, m is the number of variables in each antecedent pattern.

For each rule to which **S-indexing** is applied, an index key table is managed so that a RUI can be accessed and modified by the corresponding index key in the table. Since every argument has the same set of variables, two instances with identical variable bindings will have the same index key, and eventually will access the same RUI. When a new instance is processed, the system retrieves the corresponding RUI via its index key, changes the values of <pcount>, <ncount>, and <fns> properly, and replaces the old RUI by the new one. The main benefit we get from indexing is that there is no need for checking binding conflicts between RUIs, since different bindings lead to different RUIs. As a result, it is sufficient to maintain a single RUI for each different variable substitution. A schematic of the **S-indexing** mechanism is given in Figure 4.2.

As an example, consider an *and-or* rule

$${}_4\mathbb{X}_2^3\ \{\ P1(x,y),\ P2(x,y),\ P3(x,y),\ P4(x,y)\}$$

and facts

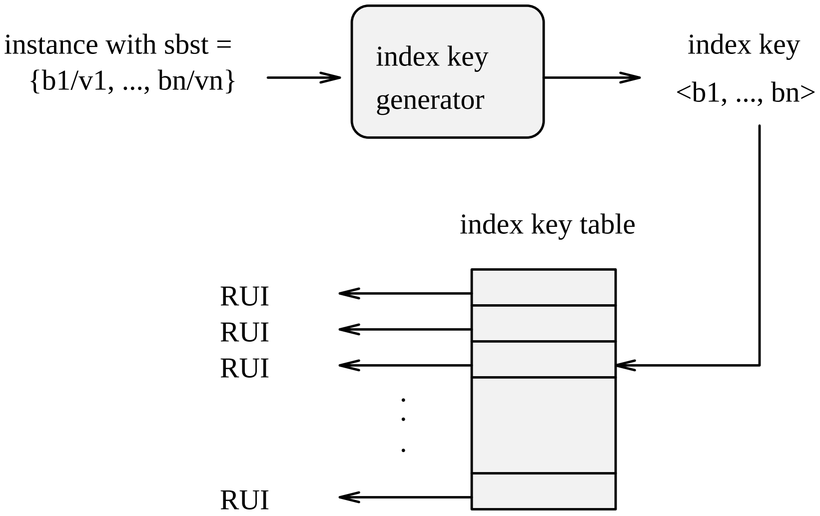


Figure 4.2: Overview of **S-indexing** mechanism

$$P1(a,b) \quad P2(a,b) \quad \neg P3(b,c) \quad P4(b,c)$$
$$\neg P1(c,d) \quad \neg P2(b,c) \quad \neg P3(a,d) \quad P4(a,d)$$

Here, we use $\neg$ as the negation symbol abbreviating $_{1}\!\Join_{0}^{0}$. There are two variables in the rule, so an index key will have the form of $< b_1, b_2 >$, where b_1 is a bound value for x and b_2 is a bound value for y. When this *and-or* rule is activated, facts that are matched with its arguments provide information about variable substitutions and whether they are positive or negative instances. A successful matching of $P1(x,y)$ and $P1(a,b)$ builds a new RUI and creates a new index key $< a, b >$ in the table as below.

index table RUIs

$\boxed{< a, b >}$ $\longrightarrow$ (1 0 {`P1:true`})

Note that we removed the `<sbst>` field in the RUI representation because it is no longer needed. The next instance $P2(a,b)$ that is matched with $P2(x,y)$ produces the same index key, so the previous RUI is updated as shown.

$\boxed{< a, b >}$ $\longrightarrow$ (2 0 {`P1:true, P2:true`})

The next instance $\neg P3(b,c)$ produces a new index key $< b,c >$ that is added to the index table, and creates a new RUI.

$\boxed{< a,b >} \longrightarrow$ (2 0 {`P1:true, P2:true`})
$\boxed{< b,c >} \longrightarrow$ (0 1 {`P3:false`})

The next instance $P4(b,c)$ that is matched with $P4(x,y)$ produces an index key $< b,c >$ that is already built in the table, so the corresponding RUI is replaced by an updated RUI.

$\boxed{< a,b >} \longrightarrow$ (2 0 {`P1:true, P2:true`})
$\boxed{< b,c >} \longrightarrow$ (1 1 {`P3:false, P4:true`})

The other instances are processed in the same fashion, and the following RUI distribution is made after $\neg P1(c,d)$, $\neg P2(b,c)$, $\neg P3(a,d)$, and $P4(a,d)$ are processed.

$\boxed{< a,b >} \longrightarrow$ (2 0 {`P1:true,P2:true`})
$\boxed{< b,c >} \longrightarrow$ (1 2 {`P2:false, P3:false, P4:true`})
$\boxed{< c,d >} \longrightarrow$ (0 1 {`P1:false`})
$\boxed{< a,d >} \longrightarrow$ (1 1 {`P3:false,P4:true`})

As soon as $\neg P2(b,c)$ is processed, the rule of inference of *and-or* is applied to infer $P1(b,c)$ for the index of $< b,c >$, since the *and-or* rule says at least 2 of 4 arguments should be true, but 2 arguments are already known to be false (`<ncount>` is 2). So the final RUI distribution looks like

$\boxed{< a,b >} \longrightarrow$ (2 0 {`P1:true,P2:true`})
$\boxed{< b,c >} \longrightarrow$ (2 2 {`P1:true, P2:false,P3:false, P4:true`})
$\boxed{< c,d >} \longrightarrow$ (0 1 {`P1:false`})
$\boxed{< a,d >} \longrightarrow$ (1 1 {`P3:false,P4:true`})

In the **S-indexing** method, there is no need for substitution compatibility tests because a different variable substitution refers to a different RUI. Instead, this method requires some extra time to find the corresponding index key in the table. The complexity of this table lookup depends on the particular implementation, for instance, we can achieve constant time complexity by using a hash table.

The number of RUIs in this method is equal to the number of different variable substitutions obtained from instances, and consequently is also equal to the size of the index table. Therefore, the number of RUIs depends on how many different variable substitutions exist.

4.4 P-tree Algorithm

The **P-tree** algorithm is designed for deduction rules using *and-entailment* or *numerical quantifier*. More precisely, this algorithms handles all *numerical quantifier* rules and those *and-entailment* rules whose antecedents have different sets of variables.

A binary pattern tree called a **P-tree** is compiled from the set of antecedents of a rule, and RUIs are distributed over the nodes of the **P-tree**. A **P-tree** of a rule is defined as a binary tree in which, (1) a leaf node corresponds to an antecedent pattern, (2) a parent node is a conjunction of its children, and (3) the root node represents the whole conjunction of the rule premise.

4.4.1 P-tree Compilation

A **P-tree** is compiled from a collection of antecedent patterns by considering the variables in each pattern. The algorithm takes a list of pattern-variable specifications as its input and produces a tree of patterns as its output.

As an example, consider an *and-entailment* rule

$$\forall \mathrm{v1,v2,v3,v4,v5} \quad [\mathtt{A}(\mathrm{v1,v2}), \mathtt{B}(\mathrm{v3,v4}), \mathtt{C}(\mathrm{v3,v5}), \mathtt{D}(\mathrm{v1,v3}), \mathtt{E}(\mathrm{v2,v5}), \mathtt{F}(\mathrm{v2,v3}), \mathtt{G}(\mathrm{v1,v4}) \\ \&\!\!\rightarrow \; \mathtt{H}(\mathrm{v1,v2,v3,v4,v5})]$$

The pattern-variable specification of a pattern $\mathrm{P}(\mathrm{x}_1, \ldots, \mathrm{x}_n)$ is defined as a list $(\mathrm{P}\ \mathrm{x}_1 \cdots \mathrm{x}_n)$. So, the pattern-variable list of the above rule will be

((A v1 v2) (B v3 v4) (C v3 v5) (D v1 v3) (E v2 v5) (F v2 v3) (G v1 v4))

The compilation algorithm is divided into 3 procedures.

`PatVar-to-VarPat` converts a pattern-variable list into a variable-pattern list.

A variable-pattern specification has the form of (`v` $\mathrm{P}_1 \cdots \mathrm{P}_m$), where `v` is a variable and each P_i is an antecedent pattern that contains `v`. `PatVar-to-VarPat` generates the following variable-pattern list from the above pattern-variable list.

((v1 `A D G`) (v2 `A E F`) (v3 `B C D F`) (v4 `B G`) (v5 `C E`))

The second procedure `VarPat-to-PatSeq` builds a linear sequence of patterns from a variable-pattern list.

The basic idea in this procedure is to arrange those patterns that have shared variables to be close in the sequence. From the above variable-pattern list, `VarPat-to-PatSeq` works as follows. Initially, the pattern sequence is set to (`A D G`) since the first variable-pattern pair is (v1 `A D G`), and then v1 is marked as processed. The union of variables of these patterns is (v1 v2 v3 v4). Now v2 is the first unprocessed variable in this union list, so (`A E F`) is the next candidate to be included in the sequence, but only `E` and `F` are inserted since `A` is already in the sequence. The resulting sequence is (`A D G E F`) with (v1 v2 v3 v4 v5) as its union of variables. Now v3 is the next unprocessed variable, so (`B C D F`) is the candidate to be included in the sequence, and `B` and `C` are inserted to make the final pattern sequence (`A D G E F B C`). Note that the order of the sequence is important.

The third procedure `PatSeq-to-PTree` builds a **P-tree** from a pattern sequence.

A main step of this procedure is to extract the first two patterns from the sequence to make them adjacent in the tree if they share a common variable. Otherwise the first pattern is appended to the intermediate tree built so far. For example, from the pattern sequence (`A D G E F B C`) obtained in the previous procedure, `A` and `D` are combined since they share a variable v1. Next two patterns `G` and `E` have no common variable, so `G` is appended to (`A D`) to make ((`A D`) `G`). Since `E` and `F` share a variable v2, and `B` and `C` share a variable v3, we can obtain the first intermediate tree ((`A D`) `G` (`E F`) (`B C`)). Now the procedure is called recursively. The first sequence (`A D`) has variables (v1 v2 v3), so it is combined with `G`. Also, the third sequence (`E F`) which has variables (v2 v3 v5) is combined with the fourth sequence (`B C`) which has variables (v3 v4 v5). This results in the second intermediate tree (((`A D`) `G`) ((`E F`) (`B C`))). Eventually, the final **P-tree** becomes ((((`A D`) `G`) ((`E F`) (`B C`)))). Figure 4.3 depicts a resulting

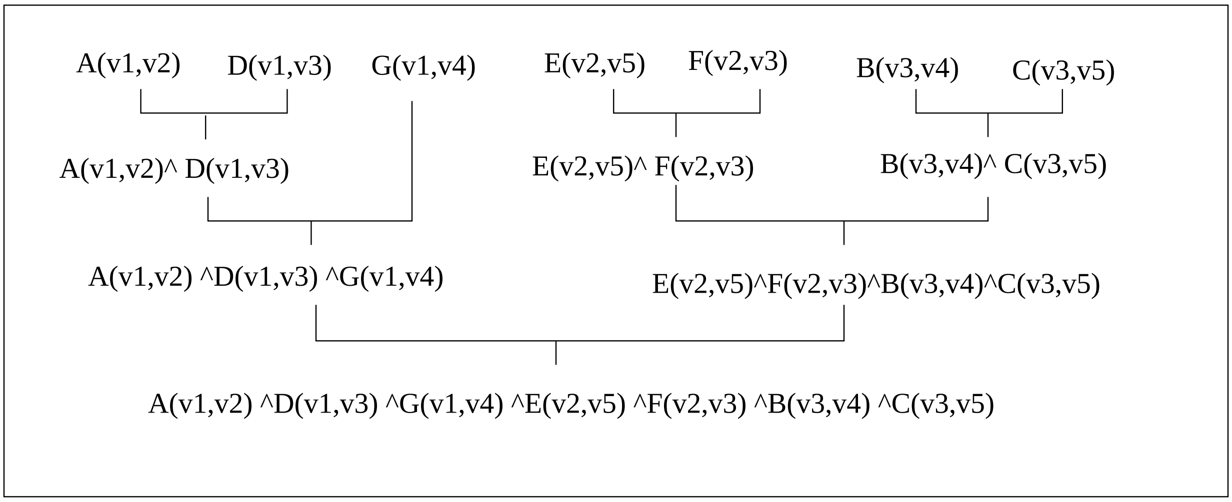


Figure 4.3: A **P-tree** for an example rule

P-tree for the example rule graphically.

Note that the **P-tree** algorithm must deal with the situation in which there are several groups of antecedent patterns in a rule that have disjoint variable unions. For example, consider the following rule.

$$\forall \text{v1,v2,v3,v4,v5,v6} \quad [\texttt{A1}(\text{v1,v2}), \texttt{A2}(\text{v2,v3}), \texttt{A3}(\text{v3,v1}), \texttt{A4}(\text{v4,v5}), \texttt{A5}(\text{v5,v6}), \texttt{A6}(\text{v6,v4})$$
$$\&\!\rightarrow \ \texttt{B}(\text{v1,v2,v3,v4,v5,v6})]$$

The union of variables for a group (`A1 A2 A3`) is (v1 v2 v3), and it is disjoint from the union of variables for (`A4 A5 A6`) which is (v4 v5 v6). In this case, the procedure `VarPat-to-PatSeq` builds a pattern sequence ((`A1 A2 A3`) (`A4 A5 A6`)). Also, the procedure `PatSeq-to-PTree` builds a **P-tree** for each pattern group and combines them to make a final `P-tree` ((((`A1 A2`) `A3`) ((`A4 A5`) `A6`))).

The **P-tree** algorithm has reasonable complexity. Suppose p is the number of antecedent patterns in a rule and v is the average number of variables in a pattern. Then, `PatVar-to-VarPat` has the complexity of $O(p \cdot v)$, `VarPat-to-PatSeq` has $O(v)$ complexity, and `PatSeq-to-PTree` has $O(p)$ complexity. The overall **P-tree** algorithm has $O(p \cdot v)$ complexity.

It is possible that several **P-tree**s can be built from the same set of antecedent patterns. For example, we can obtain at least three **P-tree**s from the above example rule as shown below, including `t1` that is compiled by our algorithm.

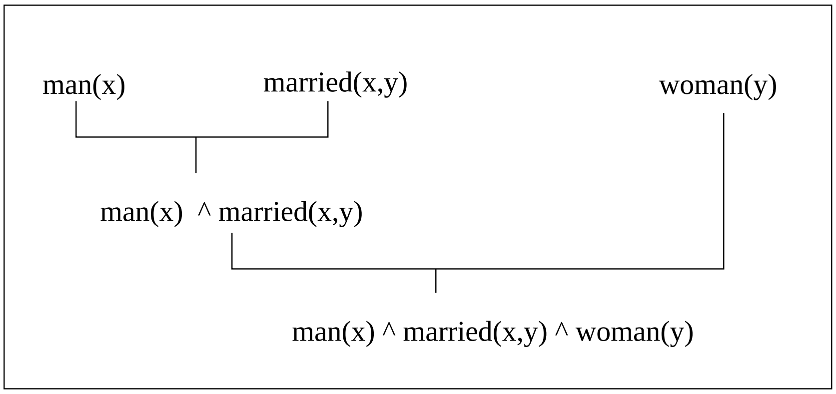
```
(t1) ((((A D) G) ((E F) (B C))))
(t2) ((((A D) (G B)) ((E F) C)))
(t3) (((((A D) G) (E F)) (B C)))
```

In fact, it is difficult to decide which **P-tree** is better, since the complexity of processing **P-tree** RUI sets depends on several factors including the shape of the **P-tree**, the number of instances for each pattern, and the number of successfully combined instances by substitution compatibility tests. If we assume each pattern has the same number of instances and combined instances are produced in the same rate for each non-leaf node, the optimality of the **P-tree** can be determined by the average length of a path from a leaf node to the root node that is the same as the average number of propagations of combined instances to the parent node. In the above **P-tree**s, the average path length of `t1` or `t2` is $(3+3+2+3+3+3+3)/7 = 2.85$, and the average path length of `t3` is $(4+4+3+3+3+2+2)/7 = 3$. In this sense, we can claim that our **P-tree** algorithm produces near optimal **P-tree**s, since it always tries to make a tree balanced.

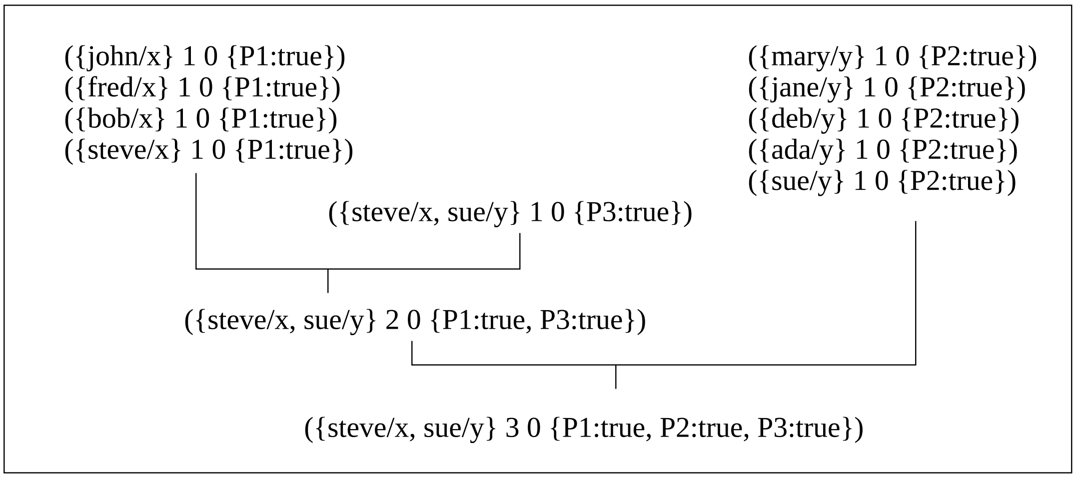
4.4.2 Distributing RUIs over P-tree Nodes

RUIs are distributed by assigning a RUI set to each node in the **P-tree**. The RUI set of a leaf node consists of those RUIs that are directly built from the instances of the corresponding antecedent pattern. The RUI set of a non-leaf node consists of those RUIs that are successfully combined from RUIs of its children nodes through binding conflict resolution. The RUIs in the RUI set of the root node contribute to drawing new conclusions when the value of `<pcount>` is equal to the number of antecedents. For example, a **P-tree** for the rule $\mathbf{r}_{husband}$ is drawn in Figure 4.4(a), and the distribution of RUIs over the nodes of this **P-tree** is shown in Figure 4.4(b).

The distributed RUI set scheme using **P-tree**s significantly reduces the overall number



(a) A **P-tree** for the husband rule



(b) Distribution of RUIs over **P-tree** nodes

Figure 4.4: A **P-tree** distribution for the rule $\mathbf{r}_{husband}$

of substitution compatibility tests as well as the total number of RUIs, since substitution compatibility is only tested between RUIs of adjacent nodes. In the husband relation example, 33 RUIs and 49 compatibility tests are made in the linear RUI set method as shown in Figure 4.1, whereas the **P-tree** RUI set method produces only 12 RUIs and 9 compatibility tests. The differences of these measures will be significant as the problem size increases.

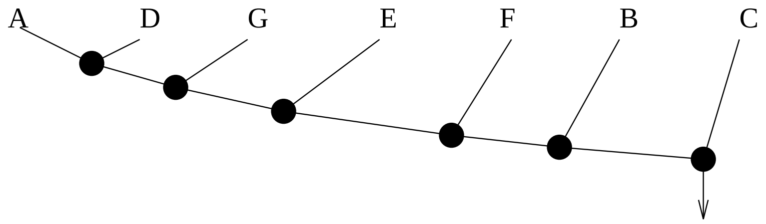
4.4.3 Comparisons with the RETE Algorithm

The **P-tree** algorithm can be compared with the RETE algorithm [Forgy, 1982, Gupta *et al.*, 1989, Miranker, 1987, Nayak *et al.*, 1988] developed to reduce the overhead of pattern matching when many patterns and objects are in the system.

Similarities include that both methods build a tree (or network) structure from the antecedents of a rule (or a set of productions), and each node of the structure saves its instances as a way of avoiding duplicate processing. Also, after a network is built, instances (or working memory elements) are fed to the structure to produce combined instance information that has consistent variable bindings. In production systems, this combined instance information results in a 'conflict set'. Substitution compatibility tests are performed for shared variables between two adjacent nodes (in RETE, this process is called 'join').

The main difference between the two methods is in how a tree or a network is built from patterns. The pattern compiler in the RETE algorithm builds a network by linking together nodes which test elements for intra-element features and inter-element features. Intra-element features are the ones that involve only one working memory element, and inter-element features results from having shared variables in two patterns. One problem in this network compilation is that several antecedent patterns in a production rule with shared variables may cause a long chain of join nodes. As a result, the cost of propagating combined instances may increase. Compared to this, the **P-tree** compilation algorithm tries to make a tree balanced so that the average number of propagations of combined instances can be minimized. For example, the following RETE network may be built from the example rule discussed in the previous section. In this network, the average path length from the root to a terminal node is $(6+6+5+4+3+2+1)/7 = 3.85$, which is much larger than the average path length in our

P-tree which is 2.85.



The other difference is in how the test is performed in a join node. In the RETE algorithm, the length of a join node depends on the number of value pairs tested by the node. In other words, if two patterns with many shared variables are joined, the compatibility test in this node will be complicated since a test is made on each shared variable. In the processing of **P-tree** RUI sets, the compatibility test in a merge node is performed more efficiently by comparing the `<sbst>` fields of two RUIs.

We now provide a set of rules to see how the RETE algorithm and the **P-tree** algorithm produce different outputs.

∀w,x,y,z [`male`(x), `parent`(z,x), `grandp`(z,y), `parent`(w,y) &→ `uncle`(x,y)]

∀w,x,y,z [`female`(x), `parent`(z,x), `grandp`(z,y), `parent`(w,y) &→ `aunt`(x,y)]

∀x,y,z [`male`(x), `female`(y), `married`(x,y), `father`(z,y) &→ `father-in-law`(z,x)]

∀x,y,z [`male`(x), `female`(y), `married`(x,y), `mother`(z,y) &→ `mother-in-law`(z,x)]

Figure 4.4.3 shows the RETE network compiled from this set of rules, and Figure 4.4.3 shows a set of **P-tree**s compiled from the same set of rules. The average length of a path from the root node to a terminal node in the RETE network (2.25) is slightly higher than the average length of a path from a leaf node to the root node in the **P-tree** (2.0). The differences of these measures will be more significant as the number of patterns in a rule increases.

4.5 Complexity of Processing RUI Sets

This section compares the complexity of processing RUI sets for the linear set method, the **S-indexing** method, and the **P-tree** method. Two main metrics in these performance measures are; (1) the number of total RUIs in the rule (space complexity), and (2) the number of

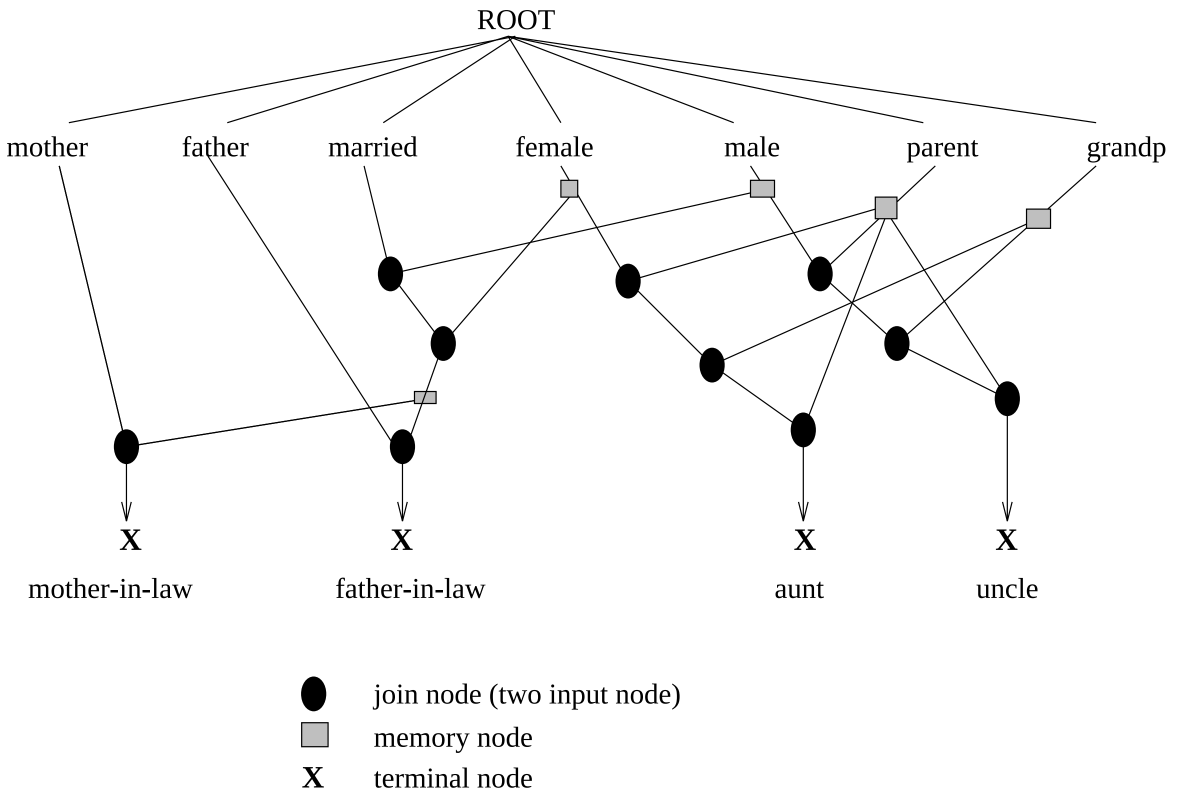


Figure 4.5: A RETE network for the kinship example

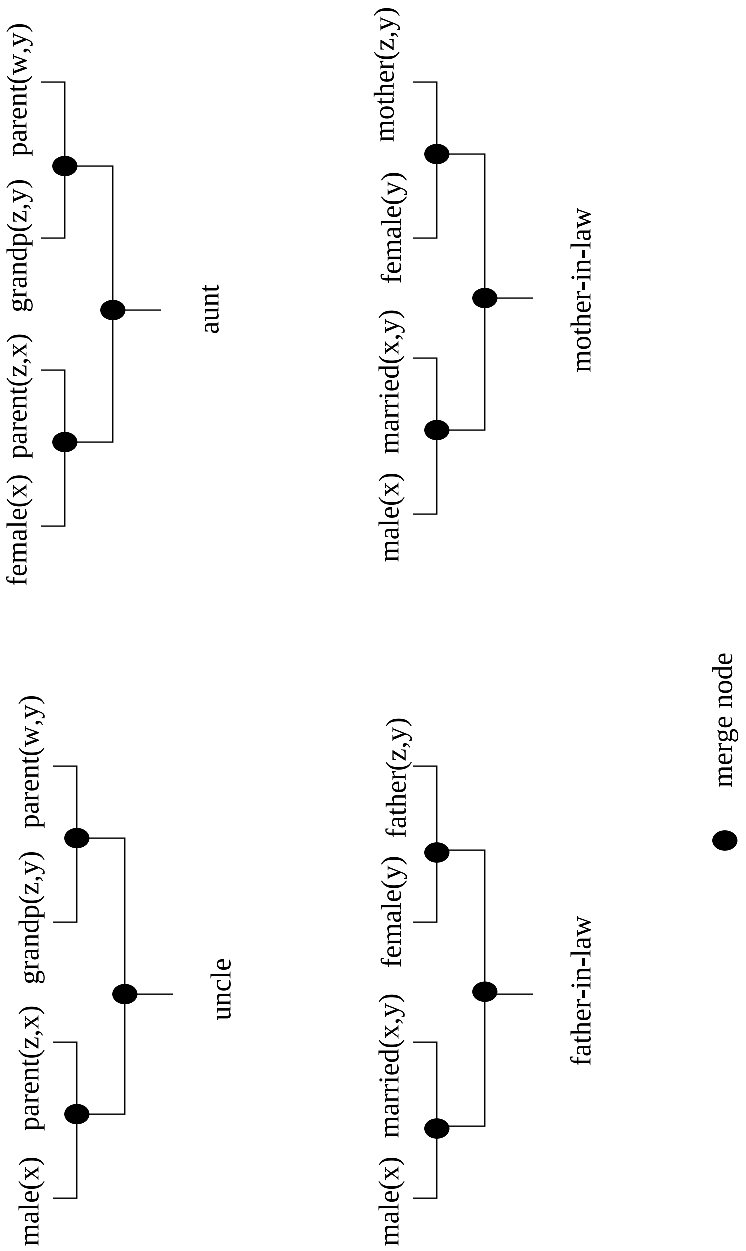


Figure 4.6: A set of **P-tree**s for the kinship example

substitution compatibility tests between RUIs during rule activation (time complexity). We assume that there are, on average, n antecedents in a rule and that each antecedent has m instances. In this analysis, a best case occurs when the two metrics have the minimum values, and a worst case occurs when the two metrics have the maximum values.

4.5.1 Complexity of Processing Linear RUI Sets

A best case in processing linear RUI sets happens when all the compatibility tests fail because of the contents of the knowledge base. For instance, a rule saying *a person who is an actor, a singer, and also very famous is rich*

∀x [`actor`(x), `singer`(x), `famous`(x) &→`rich`(x)]

with the following facts in the knowledge base

`actor`(wayne)
`actor`(taylor)
`singer`(queen)
`singer`(simon)
`famous`(jim)
`famous`(john)

would not produce any merged RUIs since all the compatibility tests fail. In this case, the number of RUIs is $m \cdot n$ which has the complexity of $O(m \cdot n)$, and the number of compatibility tests is

$$m^2 \cdot \sum_{i=1}^{n-1} i = m^2 \cdot (n^2 - n)/2$$

which has the complexity of $O(m^2 \cdot n^2)$.

A worst case happens when all compatibility tests succeed, especially when there is no shared variables among all antecedents. An example rule saying *all americans and canadians are friendly* is defined by

∀x,y [`american`(x), `canadian`(y) &→`friendly`(x,y)]

In this case, the number of RUIs is

$$m \cdot \sum_{i=1}^{n}(m+1)^{i-1} = (m+1)^n - 1$$

and the number of compatibility tests is

$$m \cdot \sum_{i=1}^{n}(m+1)^{i-1} - m \cdot n = (m+1)^n - m \cdot n - 1$$

Note that both have the complexity of $O(m^n)$.

In order to estimate the complexity of an average case, we assume that compatibility tests between two RUI sets of the size m are successful in m out of m^2 cases. In fact, this assumption is based on empirical statistics. Then, the number of RUIs is computed as

$$m \cdot \sum_{i=1}^{n+1} 2^{i-1} = m \cdot (2^{n+1} - 1)$$

which is $O(m \cdot 2^n)$, and the number of compatibility tests is

$$m^2 \cdot \sum_{i=1}^{n-1}(2^i - 1) = m^2 \cdot (2^n - n - 1)$$

which is $O(m^2 \cdot 2^n)$.

4.5.2 Complexity of Processing S-indexing RUI Sets

In the **S-indexing** method, the number of substitution compatibility tests is constant because a different variable substitution refers to a different RUI. Instead, this method requires some extra time to find the corresponding index key in the table for a variable substitution. The complexity of this table lookup depends on a particular implementation of the index key table. More specifically, it has a linear time complexity for a sequential table, and a constant time complexity for a hash table.

The number of RUIs in this method is equal to the number of different variable substitutions obtained from instances, and consequently is also equal to the size of the index table. Therefore, the number of RUIs relies on how many different variable substitutions exist.

Since each antecedent has m instances on average, a best case happens when there are only m different variable substitutions. The example rule we have seen in the previous section

∀x [`actor`(x), `singer`(x), `famous`(x) &→`rich`(x)]

with the following facts in the knowledge base

```
actor(wayne)
actor(taylor)
singer(wayne)
singer(taylor)
famous(wayne)
famous(taylor)
```

has 2 different substitutions $\{wayne/x\}$ and $\{taylor/x\}$. In this case, the number of RUIs has $O(m)$ complexity.

A worst case is when all instances have different substitutions. Since there are n antecedents, the complexity will be $O(m \cdot n)$. For example, the following facts result in 6 different substitutions.

```
actor(wayne)
actor(taylor)
singer(queen)
singer(simon)
famous(jim)
famous(john)
```

An average case happens when half of $m \cdot n$ instances have the same variable substitutions, so the complexity will be $O(m \cdot n)$. For example, the following facts result in 3 different substitutions.

```
actor(wayne)
actor(taylor)
singer(queen)
singer(wayne)
famous(taylor)
famous(queen)
```

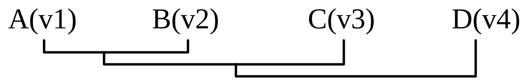
4.5.3 Complexity of Processing P-tree RUI Sets

A best case in processing **P-tree** RUI sets is the same as a best case in the linear set method where all compatibility tests fail regardless of the shape of the **P-tree**. The number of RUIs is $m \cdot n$ which is the same as that of the linear set method. Compatibility tests are only performed for adjacent nodes, so the number of compatibility tests is $m^2 \cdot n/2$ whose complexity is $O(m^2 \cdot n)$.

A worst case is the same as a worst case in the linear set method when all substitution compatibility tests succeed. In particular, for a rule in which there is no shared variables among all antecedents, the **P-tree** compilation algorithm produces a skewed binary tree. For example, a **P-tree** for a rule

∀v1,v2,v3,v4 [A(v1), B(v2), C(v3), D(v4) &→E(v1,v2,v3,v4)]

looks like



In this case, the number of RUIs is

$$n \cdot m + \sum_{i=1}^{n-1} m^{i+1} = n \cdot m + m^2 \cdot (m^{n-1} - 1)/(m-1)$$

and the number of compatibility tests is

$$\sum_{i=1}^{n-1} m^{i+1} = m^2 \cdot (m^{n-1} - 1)/(m-1)$$

both of which have the complexity of $O(m^n)$.

Average case analysis is done with three assumptions; (1) there are n antecedents with $n = 2^k$ for some non-negative integer k, (2) a **P-tree** is a full binary tree, and (3) the compatibility tests between two RUI sets with the size of m are successful in m out of m^2 cases. The depth of a full binary **P-tree** with 2^k patterns is k, and the number of nodes in the tree is $2^{k+1} - 1$. In this case, the number of RUIs is

$$m \cdot \sum_{i=1}^{k+1} 2^{i-1} = m \cdot (2^{k+1} - 1) = m \cdot (2 \cdot n - 1)$$

which is $O(m \cdot n)$. The number of compatibility tests is

$$m^2 \cdot \sum_{i=1}^{k} 2^{i-1} = m^2 \cdot (2^k - 1) = m^2 \cdot (n-1)$$

which is $O(m^2 \cdot n)$.

Complexity comparisons are summarized in the following table. Here, n denotes the average number of antecedents in a rule, and m denotes the average number of instances for each antecedent.

	linear *RUI set handling*		**P-tree** *RUI set handling*		**S-indexing** *RUI set handling*	
Cases	# of RUIs	# of tests	# of RUIs	# of tests	# of RUIs	# of tests
Best	$O(m \cdot n)$	$O(m^2 \cdot n^2)$	$O(m \cdot n)$	$O(m^2 \cdot n)$	$O(m)$	$O(1)$
Worst	$O(m^n)$	$O(m^n)$	$O(m^n)$	$O(m^n)$	$O(m \cdot n)$	$O(1)$
Average	$O(m \cdot 2^n)$	$O(m^2 \cdot 2^n)$	$O(m \cdot n)$	$O(m^2 \cdot n)$	$O(m \cdot n)$	$O(1)$

Notice in this table that the **S-indexing** method is more efficient than the **P-tree** method in both metrics, but there is a restriction in applying **S-indexing** that all patterns must have the same set of variables. For this reason, **S-indexing** is preferred to **P-tree** for an *and-entailment* rule whose antecedents have the same set of variables.

Chapter 5

Test Results

The schemes of knowledge migration, knowledge shadowing, **P-tree**, and **S-indexing** have been implemented in the SNePS Inference Package (SNIP) [Hull, 1986, Shapiro and Rapaport, 1992a, Shapiro and Rapaport, 1992b], which is a semantic-network based natural deduction system. SNIP treats inference as an activation of the network, and employs an object-oriented execution model with a number of processes and message passing among processes [Hull, 1986]. A process is allocated to each node in the network when the node is involved in inference, and each process has a set of registers that are maintained to keep the information used in communications between processes. The expertise base and the RUI set are implemented by these registers.

We show the effectiveness of our speedup learning methods by presenting several application problems. We ran the original SNIP (SNIP2.1) and the modified SNIP (SNIP2.2) on each application problem to compare their results. Test runs are performed on a SUN SPARC station with a COMMON LISP environment. Performance measures are based on several metrics including the CPU execution time, the number of matches, the number of RUIs, the number of substitution compatibility tests for binding conflict resolution, and the number of activations of inference rules.

5.1 SNePSLOG Interface

In these test runs, knowledge bases are represented by SNePSLOG (a logical interface to SNePS) [Shapiro *et al.*, 1981]. The following is a summary of SNePSLOG expressions for non-standard connectives and quantifiers.

(1) *And-or* denoted by ${}_n\vee\!\!\!\wedge_i^j$ is represented by `andor(i,j)`, and the negation ${}_1\vee\!\!\!\wedge_0^0$ is abbreviated by $\sim$. For example,

${}_2\vee\!\!\!\wedge_1^1$ (`isa`(Cyd,teacher), `isa`(Deb,teacher))

is represented by

`andor(1,1) (isa(Cyd,teacher), isa(Deb,teacher))`

(2) *Thresh* denoted by ${}_n\Theta_i^j$ is represented by `thresh(i,j)`, ${}_n\Theta_i$ is represented by `thresh(i)`,

and the equivalence relation ${}_n\Theta_1$ is abbreviated by `<=>`. For example,

$_2\Theta_1$ (`isa`(Bea,teacher), `isa`(Eve,teacher))

can be represented by any one of the following:

```
thresh(1) (isa(Bea,teacher), isa(Eve,teacher))
isa(Bea,teacher) <=> isa(eve,teacher)
```

(3) There are three kinds of connectives for entailments: `=>` is used when there is only one antecedent, `v=>` is used for *or-entailment* with more than one antecedent, and `&=>` is used for *and-entailment*. For example,

∀x (`man`(x) $\vee\!\!\rightarrow$`mortal`(x))

∀x,y,z (`on`(x,y), `on`(y,z) $\&\!\!\rightarrow$`on`(x,z))

are represented, respectively, by

```
all(x) (man(x) => mortal(x))
all(x,y,z) ({on(x,y), on(y,z)} &=> {on(x,z)})
```

(4) *Numerical quantifier* denoted by ${}_n\exists_i^j$ is represented by `nexists(i,j,n)`. For example, v

${}_5\exists_2^2$(w) (`woman`(w): `isa`(w,Teacher))

is represented by

```
nexists(2,2,5) (w) ({woman(w)} : {isa(w,Teacher)})
```

5.2 The Jobs Puzzle

The jobs puzzle problem is described in [Wos *et al.*, 1984, pp 44–83] as follows.

" There are four people: Roberta, Thelma, Steve, and Pete. Among them, they hold eight different jobs. Each holds exactly two jobs. The jobs are: chef, guard, nurse, telephone operator, police officer (gender not implied), teacher, actor, and boxer. The job of nurse is held by a male. The husband of the chef is the telephone operator. Roberta is not a boxer. Pete has no education past the ninth grade. Roberta, the chef, and the police officer went golfing together. Now the question: Who holds which jobs?"

As you might have noticed, there is some hidden information in this puzzle description as mentioned below that should be represented explicitly in the knowledge base to do reasoning correctly. Details of this hidden information are mentioned in [Wos *et al.*, 1984, pp 44–46].

(1) The sex of the people may be determined by the name, so Roberta and Thelma are female, and Steve and Pete are male.

(2) General linguistic knowledge recognizes an actor as a male from its suffix `-or`.

(3) From two sentences 'among them [the four people], they hold eight different jobs' and 'each holds exactly two jobs', we infer that 'no job is held by more than one person'.

(4) From the sentence 'the husband of the chef is the telephone operator' and the implicit fact that husbands are male, the chef must be female, and the telephone operator must be male.

(5) Pete can be neither the teacher, the police officer, nor the nurse, since those jobs require more than a ninth-grade education.

(6) From the phrase 'Roberta, the chef, and the police officer', we know that Roberta is neither the chef nor the police officer. Since they went golfing together, we also know that the chef and the police officer are not the same person.

5.2.1 Knowledge Base

A knowledge base for the jobs puzzle is represented by SNePSLOG. (Rules and facts are described in typewriter style. Statements in italics denote natural language descriptions. All hidden information is included.)

There are four people: Roberta, Thelma, Steve, and Pete

The jobs are: chef, guard, nurse, telephone operator, police officer, teacher, actor, and boxer

```
Person({Roberta,Thelma,Steve,Pete})
Female({Roberta,Thelma})
```

```
Male({Steve,Pete})
Job({chef,guard,nurse,operator,police,teacher,actor,boxer})
```

Among them, they hold eight different jobs, each holds exactly two jobs

```
all(p)(Person(p) => nexists(2,2,8)(j)({Job(j)}: {Is(p,j)}))
all(j)(Job(j) => nexists(1,1,4)(p)({Person(p)}: {Is(p,j)}))
```

The job of nurse is held by a male

The husband of the chef is the telephone operator

```
all(w)(Female(w)=>andor(0,0){Is(w,nurse),Is(w,actor),Is(w,operator)})
all(m)(Male(m) => ~Is(m,chef))
```

Roberta is not a boxer

Pete has no education past the ninth grade

```
~Is(Roberta,boxer)
andor(0,0){Is(Pete,nurse),Is(Pete,teacher),Is(Pete,police)}
```

Roberta, the chef, and the police officer went golfing together

```
andor(0,0) {Is(Roberta,chef), Is(Roberta,police)}
all(p)(Person(p) => andor(0,1){Is(p,chef), Is(p,police)})
```

Now the question: Who holds which jobs?

```
Is(?p,?j)?
```

In this representation, we have used the concept of set argument [Shapiro, 1986] for the representation of `Person`, `Job`, `Female`, and `Male` propositions. Note that we can infer `Male(Steve)` and `Male(Pete)` from `Male({Steve, Pete})`, and so forth. The answer to this puzzle for the query *who holds which jobs* is given below.

```
Is(Roberta, teacher)
Is(Roberta, guard)
Is(Thelma, chef)
```

queries	CPU time (seconds)	No. of Matches	No. of RUIs	Subst. Compat. Tests	Inference Rule Firings
SNIP 2.1					
Is(?p,?j)	38	40	508	10898	130
SNIP 2.2					
Is(?p,?j)	20	28	228	1984	52

Table 5.1: Statistics comparisons for the jobs puzzle

```
Is(Thelma, boxer)
Is(Steve, nurse)
Is(Steve, police)
Is(Pete, actor)
Is(Pete, operator)
```

Statistics comparisons for the query `Is(?p,?j)` are given in Table 5.1.

5.2.2 Effect of Knowledge Shadowing

In order to see the effectiveness of knowledge shadowing, instead of asking *who holds which jobs?*, we ask a sequence of 4 questions that are more specific than the above question: *which job does Roberta hold?* (`Is(Roberta,?j)`), *which job does Thelma hold?* (`Is(Thelma,?j)`), *which job does Steve hold?* (`Is(Steve,?j)`), and *which job does Pete hold?* (`Is(Pete,?j)`).

We focus on the following two rules both of which are nested rules with *numerical quantifier* rule consequents.

```
all(p)(Person(p) => nexists(2,2,8)(j)({Job(j)}: {Is(p,j)}))
all(j)(Job(j) => nexists(1,1,4)(p)({Person(p)}: {Is(p,j)}))
```

We name the consequent of the first rule $\mathbf{r}_p$, and the consequent of the second rule $\mathbf{r}_j$.

($\mathbf{r}_p$) `nexists(2,2,8)(j)(Job(j):Is(p,j))`

($\mathbf{r}_j$) `nexists(1,1,4)(p)(Person(p):Is(p,j))`

These rules are activated by the first query `Is(Roberta,?j)`, since the pattern `Is(p,j)` is matched with the query. Backward chaining with inference rules of *or-entailment* and *numerical quantifiers* initiates the knowledge migration process to eventually generate the following specific rules.

($\mathbf{r}_{p1}$) `nexists(2,2,8)(j)(Job(j):Is(Roberta,j))`
($\mathbf{r}_{p2}$) `nexists(2,2,8)(j)(Job(j):Is(Thelma,j))`
($\mathbf{r}_{p3}$) `nexists(2,2,8)(j)(Job(j):Is(Steve,j))`
($\mathbf{r}_{p4}$) `nexists(2,2,8)(j)(Job(j):Is(Pete,j))`

($\mathbf{r}_{j1}$) `nexists(1,1,4)(p)(Person(p):Is(p,chef))`
($\mathbf{r}_{j2}$) `nexists(1,1,4)(p)(Person(p):Is(p,guard))`
($\mathbf{r}_{j3}$) `nexists(1,1,4)(p)(Person(p):Is(p,nurse))`
($\mathbf{r}_{j4}$) `nexists(1,1,4)(p)(Person(p):Is(p,operator))`
($\mathbf{r}_{j5}$) `nexists(1,1,4)(p)(Person(p):Is(p,police))`
($\mathbf{r}_{j6}$) `nexists(1,1,4)(p)(Person(p):Is(p,teacher))`
($\mathbf{r}_{j7}$) `nexists(1,1,4)(p)(Person(p):Is(p,boxer))`
($\mathbf{r}_{j8}$) `nexists(1,1,4)(p)(Person(p):Is(p,actor))`

As a result, the instance set of $\mathbf{r}_p$ denoted by $\mathcal{I}_{\mathbf{r}_p}$, and the instance set of $\mathbf{r}_j$ denoted by $\mathcal{I}_{\mathbf{r}_j}$ become as below.

$$\begin{aligned}\mathcal{I}_{\mathbf{r}_p} = \{ &< \mathbf{r}_{p1}, \{\texttt{Roberta/p}\} >, < \mathbf{r}_{p2}, \{\texttt{Thelma/p}\} >,\\ &< \mathbf{r}_{p3}, \{\texttt{Steve/p}\} >, < \mathbf{r}_{p4}, \{\texttt{Pete/p}\} >\}\\ \mathcal{I}_{\mathbf{r}_j} = \{ &< \mathbf{r}_{j1}, \{\texttt{chef/j}\} >, < \mathbf{r}_{j2}, \{\texttt{guard/j}\} >,\\ &< \mathbf{r}_{j3}, \{\texttt{nurse/j}\} >, < \mathbf{r}_{j4}, \{\texttt{operator/j}\} >,\\ &< \mathbf{r}_{j5}, \{\texttt{police/j}\} >, < \mathbf{r}_{j6}, \{\texttt{teacher/j}\} >,\\ &< \mathbf{r}_{j7}, \{\texttt{boxer/j}\} >, < \mathbf{r}_{j8}, \{\texttt{actor/j}\} >\}\end{aligned}$$

This information is used for Shadowing Principle 1 when processing the next queries `Is(Thelma,?j)`, `Is(Steve,?j)`, and `Is(Pete,?j)`. The general rules $\mathbf{r}_p$ and $\mathbf{r}_j$ are shadowed from the inference for a request that contains one of migrating substitutions in the rule's instance set, which satisfies Shadowing Principle 1. Statistics comparisons for the sequence of four queries to the jobs puzzle given in Table 5.2 show the effect of knowledge shadowing.

queries	CPU time (seconds)	No. of Matches	No. of RUIs	Subst. Compat. Tests	Inference Rule Firings
SNIP 2.1					
Is(Roberta,?j)	18	29	259	6621	123
Is(Thelma,?j)	31	34	301	8068	99
Is(Steve,?j)	66	55	552	20548	259
Is(Pete,?j)	101	65	652	21951	337
SNIP 2.2					
Is(Roberta,?j)	17	34	325	1444	63
Is(Thelma,?j)	11	20	158	440	42
Is(Steve,?j)	15	21	215	548	64
Is(Pete,?j)	6	5	67	160	21

Table 5.2: Statistics comparisons for the jobs puzzle with a sequence of four queries

5.3 "The Woman Freeman Will Marry" Puzzle

The Freeman puzzle problem is described in [Summers, 1972] as follows:

" Freeman knows five women: Ada, Bea, Cyd, Deb and Eve. The women are in two age brackets: three women are under 30 and two women are over 30. Two women are teachers and the other three women are secretaries. Ada and Cyd are in the same age bracket. Deb and Eve are in different age brackets. Bea and Eve have the same occupation. Cyd and Deb have different occupations. Of the five women, Freeman will marry the teacher over 30. Who will Freeman marry?"

We present a SNePS representation of this puzzle using *and-or*, *thresh*, and *numerical quantifier*. (The SNePS representation in [Martins and Shapiro, 1988] only uses the *and-or* and *thresh* connectives.)

5.3.1 Knowledge Base

The SNePSLOG representation of the knowledge base is as follows.

Freeman knows five women: Ada, Bea, Cyd, Deb and Eve.

```
Woman({Ada, Bea, Cyd, Deb, Eve})
```

The women are in two age brackets: three women are under 30 and two women are over 30.

```
AgeBracket({Under30, Over30})
all(w)(Woman(w) => nexists(1,1,2)(a)({AgeBracket(a)}: {Age(w,a)}))
nexists(3,3,5)(w)({Woman(w)}: {Age(w, Under30)})
```

Two women are teachers and the other three women are secretaries.

```
Job({Teacher, Secretary})
all(w)(Woman(w) => nexists(1,1,2)(j)({Job(j)}: {Isa(w,j)}))
nexists(2,2,5)(w)({Woman(w)}: {Isa(w, Teacher)})
```

Ada and Cyd are in the same age bracket.

```
all(a)(AgeBracket(a) => thresh(1){Age(Ada,a), Age(Cyd,a)})
```

Deb and Eve are in different age brackets.

```
all(a)(AgeBracket(a) => andor(1,1){Age(Deb,a), Age(Eve,a)})
```

Bea and Eve have the same occupation.

```
all(j)(Job(j) => thresh(1){Isa(Bea,j), Isa(Eve,j)})
```

Cyd and Deb have different occupations.

```
all(j)(Job(j) => andor(1,1){Isa(Cyd,j), Isa(Deb,j)})
```

Of the five women, Freeman will marry the teacher over 30.

```
nexists(1,1,5)(w)({Woman(w)}: {andor(2,2){Isa(w, Teacher), Age(w,Over30)}})
all(w)({Isa(w, Teacher), Age(w,Over30)} &=> {FreemanWillMarry(w)})
```

5.3.2 A Problem Solving Strategy

Note that the initial knowledge base does not include any information about the age or the occupation of a particular woman, so in order to figure out which category (either over 30 or under 30, and either a teacher or a secretary) each woman belongs to, we have to raise some hypotheses, reason from them, and if a contradiction is detected, mark the current hypothesis invalid and resume the reasoning with a different hypothesis.

The following is a sequence of hypotheses we made to solve this puzzle. For each hypothesis, we describe what kinds of rules and facts are derived, and if contradiction occurs, why it happened.

(1) Assume "Ada is over 30" : `Age(Ada,Over30)`

Since Ada and Cyd are in the same age bracket, Cyd is also over 30. From the rule that only two women are over 30, it is now inferred that Bea, Deb, and Eve should be under 30. But Deb and Eve should be in different age brackets. Therefore, this hypothesis leads to a contradiction. Any derived knowledge that was dependent upon the initial hypothesis will be removed from the current belief space. Now we resume reasoning by an alternative.

(2) Assume "Ada is under 30" : `Age(Ada,Under30)`

Cyd is also under 30, and no contradiction occurs. Still, the ages of Bea, Deb, and Eve are not yet determined.

(3) Assume "Bea is over 30" : `Age(Bea,Over30)`

No contradiction.

(4) Assume "Deb is over 30" : `Age(Deb,Over30)`

Since Deb and Eve are in different age brackets, Eve is under 30, and no contradiction occurs. At this point, the ages of all five women are determined without any contradiction as follows.

```
Age(Ada,Under30)
Age(Bea,Over30)
Age(Cyd,Under30)
Age(Deb,Over30)
```

```
Age(Eve,Under30)
```

We now want to find out the occupations of five women.

(5) Assume "Ada is a teacher" : `Isa(Ada,Teacher)`

No contradiction occurs.

(6) Assume "Bea is a teacher" : `Isa(Bea,Teacher)`

Since Bea and Eve have the same occupation, Eve is also a teacher. Now we know that Ada, Bea, and Eve are teachers. But this leads to a contradiction since only two women can be teachers. Any knowledge derived from this hypothesis is removed, and we resume reasoning by an alternative.

(7) Assume "Bea is a secretary" : `Isa(Bea,Secretary)`

Bea and Eve have the same occupation, so Eve is also a secretary. Note that there should exist exactly one woman who is a teacher over 30, and we know that Ada, Bea, Cyd, and Eve cannot satisfy both properties. (Ada is under 30, Bea is a secretary, Cyd is under 30, and Eve is under 30.) This implies that Deb should be the woman who is a teacher over 30. Now Ada and Deb are teachers, and since there should be exactly two teachers, Cyd and Eve are secretaries.

At this point, the occupations of all five women are determined without contradiction as follows.

```
Isa(Ada,Teacher)
Isa(Bea,Secretary)
Isa(Cyd,Secretary)
Isa(Deb,Teacher)
Isa(Eve,Secretary)
```

As a result, Freeman will marry Deb.

5.3.3 Effect of Knowledge Shadowing

During the first chaining initiated by the hypothesis "Ada is over 30", a number of specific rules are generated from general rules as follows. Here, general rules are presented with a mark (*) followed by their instances.

```
(*) all(w)(Woman(w) => (nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(w,a)}]))
  nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(Ada,a)}]
  nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(Bea,a)}]
  nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(Cyd,a)}]
  nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(Deb,a)}]
  nexists(1,1,2)(a)[{AgeBracket(a)}:{Age(Eve,a)}]

(*) all(w)(Woman(w) => (nexists(1,1,2)(j)[{Job(j)}:{Isa(w,j)}]))
  nexists(1,1,2)(j)[{Job(j)}:{Isa(Ada,j)}]
  nexists(1,1,2)(j)[{Job(j)}:{Isa(Bea,j)}]
  nexists(1,1,2)(j)[{Job(j)}:{Isa(Cyd,j)}]
  nexists(1,1,2)(j)[{Job(j)}:{Isa(Deb,j)}]
  nexists(1,1,2)(j)[{Job(j)}:{Isa(Eve,j)}]

(*) all(j)(Job(j) => (Isa(Bea,j) <=> Isa(Eve,j)))
  Isa(Eve,Teacher) <=> Isa(Bea,Teacher)
  Isa(Bea,Secretary) <=> Isa(Eve,Secretary)

(*) all(j)(JOB(j) => (andor(1,1){Isa(Cyd,j),Isa(Deb,j)}))
  andor(1,1){Isa(Deb,Teacher),Isa(Cyd,Teacher)}
  andor(1,1){Isa(Cyd,Secretary),Isa(Deb,Secretary)}

(*) all(a)(AgeBracket(a) => (Age(Ada,a) <=> Age(Cyd,a)))
  Age(Cyd,Under30) <=> Age(Ada,Under30)
  Age(Ada,Over30) <=> Age(Cyd,Over30)

(*) all(a)(AgeBracket(a) => (andor(1,1){Age(Deb,a),Age(Eve,a)}))
  andor(1,1){Age(Eve,Under30),Age(Deb,Under30)}
  andor(1,1){Age(Eve,Over30),Age(Deb,Over30)}
```

These specific rules are used to prevent general rules from being activated in subsequent reasoning. Statistics comparisons for the Freeeman puzzle are shown in Table 5.3.

5.4 Streamroller Problem

In 1978, Schubert posed the following challenge problem, called the *streamroller problem*, for automated theorem provers.

" Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Every animal either likes to eat all plants

queries	CPU time (seconds)	No. of Matches	Subst. Compat. Tests	Inference Rule Firings
SNIP 2.1				
Age(Ada,Over30)!	26.87	71	339	39
Age(Ada,Under30)!	39.22	89	598	36
Age(Bea,Over30)!	0.58	1	61	2
Age(Deb,Over30)!	3.66	1	584	9
Isa(Ada,Teacher)!	0.59	1	56	5
Isa(Bea,Teacher)!	2.05	1	59	9
Isa(Bea,Secretary)!	68.14	83	4997	200
FreemanWillMarry(?w)?	4.40	3	708	1
SNIP 2.2				
Age(Ada,Over30)!	14.74	39	85	33
Age(Ada,Under30)!	18.49	53	130	27
Age(Bea,Over30)!	0.48	1	9	1
Age(Deb,Over30)!	1.77	1	123	9
Isa(Ada,Teacher)!	0.31	1	9	3
Isa(Bea,Teacher)!	2.02	1	19	5
Isa(Bea,Secretary)!	15.76	25	477	47
FreemanWillMarry(?w)?	2.46	3	0	1

Table 5.3: Statistics comparisons for the Freeman puzzle

or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore there is an animal that likes to eat a grain-eating animal. "

This problem is known to be very hard to solve using resolution because the search space is simply too big. Some advanced resolution techniques have been developed to overcome the difficulty, such as many-sorted resolution [Cohn, 1985, Walther, 1984] and theory resolution [Stickel, 1985]. In SNIP, which is based on natural deduction, a first-order axiomatization of the puzzle can be directly executed without being converted into clausal form, which is required for the resolution method. Consequently, SNIP does not cause as serious a search space problem for this puzzle as resolution-based methods do.

5.4.1 Knowledge Base

We present a natural deduction representation for the streamroller problem with SNePS, and discuss how some of the learning algorithms are used to improve the efficiency. (The SNePS representation of the streamroller problem was first formalized by Carlos Pinto-Ferreira of Instituto Superior Tecnico at Lisbon, Portugal, and revised by the author.)

A knowledge base for this problem is given in SNePSLOG. For each statement of the puzzle, we discuss some representation issues with respect to claims made in [Stickel, 1986], and provide SNePSLOG expressions.

First of all, the meaning of each predicate is defined here.

`A(x)` : x is an animal
`W(x)` : x is a wolf
`F(x)` : x is a fox
`B(x)` : x is a bird
`C(x)` : x is a caterpillar
`S(x)` : x is a snail
`G(x)` : x is a grain
`P(x)` : x is a plant
`E(x,y)` : x eats y

```
M(x,y) : x is much smaller than y
GE(x) : x is an grain-eating animal
```

Wolves, foxes, birds, caterpillars, and snails are animals

Using *or-entailment*, this statement can be represented by one SNePSLOG expression.

```
all(x) ({W(x), F(x), B(x), C(x), S(x)} v=> {A(x)})
```

There are some of each of them

There are some grains

Existential quantifiers are implemented in SNePS using skolem constants and skolem functions.

```
W(w)
F(f)
B(b)
C(c)
S(s)
G(g)
```

Grains are plants

This statement is expressed by an *or-entailment* rule. Since there is only one antecedent, we use `=>` instead of `v=>`.

```
all(x) (G(x) => P(x))
```

Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants

Recognizing this as an ordinary disjunctive expression, we can obtain 3 separate statements:

1. if an animal x does not like to eat some plants, it must be the case that x likes to eat all animals much smaller than itself that like to eat some plants

2. if an animal x does not like to eat some animals much smaller than itself that like to each some plants, it must be the case that x does not like to each all plants

3. if an animal x does not like to eat some plants, and x does not likes to eat an animal y which is much smaller than itself, it must be the case that y does not like to eat some plants

In fact, (3) is obtained from (1). SNePSLOG expressions for the above three statements are given.

```
all(x,y) ({A(x), P(y), ~E(x,y)} &=>
     {all(z,v) ({A(z), P(v), M(z,x), E(z,v)} &=> {E(x,z)})})

all(x,y,z) ({A(x), P(y), A(z), M(z,x), E(z,y), ~E(x,z)} &=>
     {all(v) (P(v) => E(x,v))})

all (x,y) ({A(x), P(y), ~E(x,y)} &=>
     {all (z,v) ({A(z), P(v), M(z,x), ~E(x,z)} &=> {~E(z,v)})})
```

Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves

```
all(x) ({C(x), S(x)} v=> {all(y) (B(y) => M(x,y))})
all(x,y) ({B(x), F(y)} &=> {M(x,y)})
all(x,y) ({F(x), W(y)} &=> {M(x,y)})
```

Wolves do not like to eat foxes or grains

```
all(x) ({F(x), G(x)} v=> {all(y) (W(y) => ~E(y,x))})
```

Birds like to eat caterpillars but not snails

```
all(x,y) ({B(x), C(y)} &=> {E(x,y)})
all(x,y) ({B(x), S(y)} &=> {~E(x,y)})
```

Caterpillars and snails like to eat some plants

Since the existential quantifiers are in the consequent position, we use a skolemized function.

```
all(x) ({C(x), S(x)} v=> {P(f(x)), E(x,f(x))})
```

There is an animal that likes to eat a grain eating animal

Since we are doing backward chaining, we introduce a new predicate `GE` in the consequent position to initiate chaining by the query `GE(?x,?y)`.

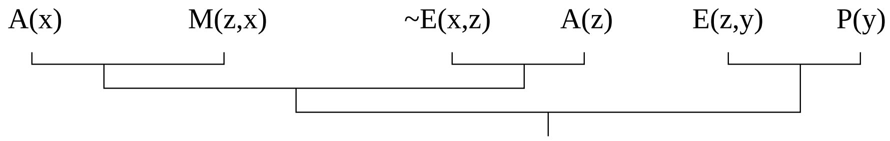
```
all(x,y,z) ({A(x), A(y), G(z), E(x,y), E(y,z)} &=> {GE(x,y)})
```

5.4.2 Effect of P-tree

The **P-tree** algorithm is effective for the following *and-entailment* rule with 6 antecedent patterns.

```
all(x,y,z) ({A(x), P(y), A(z), M(z,x), E(z,y), ~E(x,z)} &=>
     {all(v) (P(v) => E(x,v))})
```

A **P-tree** for this rule is



Statistics comparisons for the query `GE(?x,?y)` are shown in Table 5.4.

There have been a number of attempts to solve this puzzle, and the approaches can largely be divided into two categories: unsorted logic and sorted logic [Cohn, 1985, Walther, 1984]. In [Stickel, 1986], statistics comparisons for various solutions are described. Table 5.5 summarizes the statistics for unsorted logic including our system, and Table 5.6 for sorted logic. Note that there are several different interpretations for the phrase 'grain-eating animal', and this is reflected by the column named 'Problem' in the tables. In (*), the phrase means an animal that eats *some* grains, and in (**), the phrase means an animal that eats *every* grain. The approach (***) is a slight variation of (**). Details of these interpretations are discussed in [Stickel, 1986].

According to these tables, our solution produces fewer unifications, and is faster than most unsorted logic solutions. On the other hand, all sorted logic solutions outperform our

queries	CPU time (seconds)	Derived Knowledge	No. of Matches	Successful Unifications	No. of RUIs	No. of Tests	Inference Rule Firings
SNIP 2.1							
GE(?x,?y)	79	33	83	536	1552	20599	121
SNIP 2.1							
GE(?x,?y)	47	33	59	427	366	769	74

Table 5.4: Statistics comparisons for the streamroller problem

solution. Nevertheless, our contribution is that we could represent the streamroller problem in first-order rules, and do reasoning directly from them by natural deduction without having the step of converting rules into clausal form. As a result, the search space problem in SNIP is minimal.

5.5 Digital Circuit Analysis/Validation

Digital circuit validation is a process of testing whether a digital circuit functions according to its specification. Validation is complicated for most digital circuits, since it is not easy to extract a simple input-output function from a circuit specification. Model-based reasoning has been proposed to deal with this situation by constructing a model that uses structural or functional characteristics of components in the circuit [Chandrasekaran and Milne, 1985, Davis, 1984].

There are several ways of representing a digital circuit. In [Clocksin, 1987], methods of applying techniques of logic programming to problems in digital circuit analysis are presented. In this system, each digital circuit is represented by the *definitional method* in which each

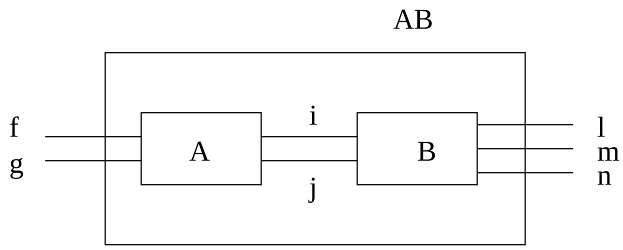
Researcher/Strategy	Problem	CPU time (seconds)	Derived clauses	Successful unifications
Stickel/CG-SOS	(*)	5694	4518	245820
Stickel/CG-SOS-TR	(*)	54	115	2494
Stickel/CG	(*)	6	42	479
Stickel/CG-TR	(*)	10	25	477
Stickel/CG-SOS-TR	(**)	212	228	10139
Stickel/CG	(**)	41	156	1905
Stickel/CG-TR	(**)	34	104	1998
Stickel/CG-SOS-TR	(***)	124	151	5840
Stickel/CG	(***)	25	135	1340
Stickel/CG-TR	(***)	13	38	764
McCune/UR	(*)	158	38	6050
McCune/LUR	(*)	44	19	1106
McCune/UR	(**)	288	49	9456
McCune/LUR	(**)	96	18	2501
Lusk & Overbeek/UR	(***)	340	127	6371
Lusk & Overbeek/QUR	(***)	155	131	26190
McCune/UR	(***)	660	131	26190
McCune/LUR	(***)	120	92	2663
Plaisted/LOCK	(***)	6	199	?
Walther/MKPR	(***)	262	60	?
Choi & Shapiro/natural deduction	?	47	33	427

Table 5.5: Comparisons of unsorted logic solutions of the streamroller problem in [Stickel, 1986, pp 93–94] with a natural deduction solution in SNIP 2.2

Researcher/Strategy	Problem	CPU time (seconds)	Derived clauses	Successful unifications
McCune/UR	(*)	6	8	67
Stickel/CG-SOS	(*)	6	13	137
Stickel/CG	(*)	4	16	76
Stickel/PTTP	(*)	0.2	35	44
McCune/UR	(**)	6	8	59
Stickel/CG-SOS	(**)	6	13	137
Stickel/CG	(**)	4	16	76
Stickel/PTTP	(**)	0.2	35	44
Walther/MKPR	(**)	3	10	48

Table 5.6: Sorted logic solutions of the streamroller problem [Stickel, 1986, page 98]

module with n ports is represented as n-ary predicate symbol, and shared ports among several modules are represented by shared variables. For example, the following digital circuit



is represented by

$A(f,g,i,j) \wedge B(i,j,l,m,n) \rightarrow AB(f,g,l,m,n)$

Hierarchical definition is possible in the definitional method, which results in a concise representation for a complex circuit by grouping a number of logically-related components into a higher-level component.

We now want to represent an `M3A2` circuit using the definitional method. This circuit has three inputs and two outputs, and consists of three multipliers and two adders satisfying the following input-output equations.

$out1 = in1 * in2 + in1 * in3$

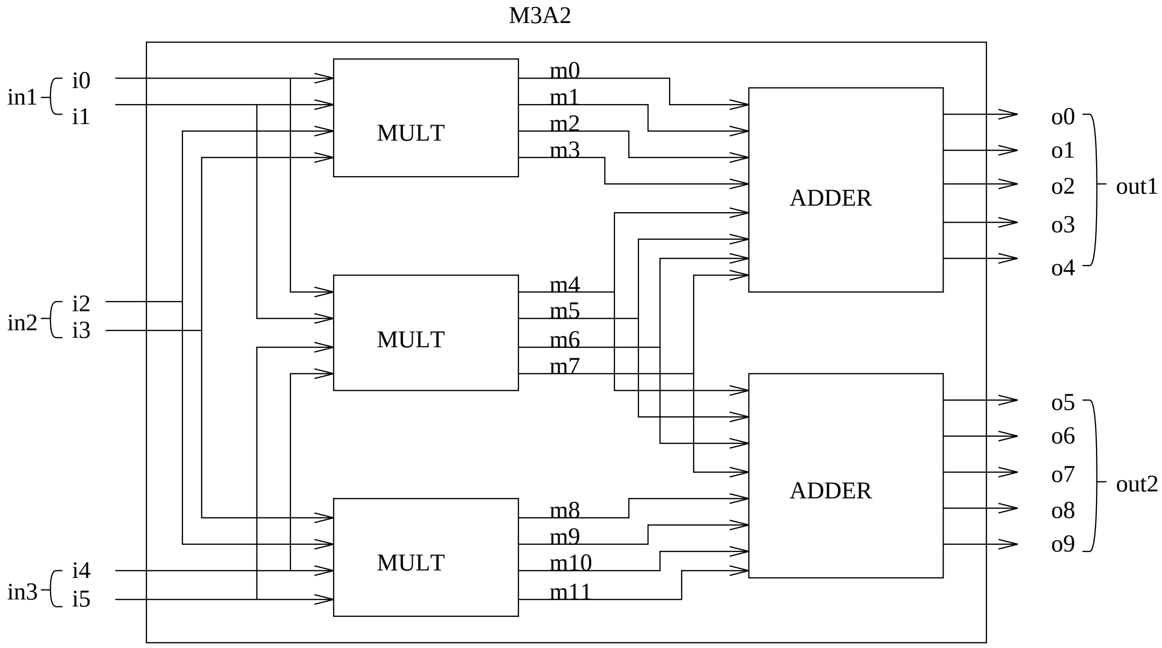


Figure 5.1: `M3A2` with 2-bit inputs and 5-bit outputs

$$out2 = in1 * in3 + in2 * in3$$

Figure 5.1 shows a bit-level schematic of the `M3A2` circuit in which inputs consist of 2 bits and outputs consist of 5 bits. In this circuit, $in1$ is represented by $i0$ and $i1$, where $i1$ is a higher-order bit. So, if $i1$=1 and $i0$=0, $in1$ has the value of 2. Here, constants 1 and 0 stand for logic high and logic low, respectively. In a similar way, $out1$ is represented by $o0$ through $o4$, where $o4$ is the highest-order bit.

From this specification, the `M3A2` circuit is represented by the definitional method as

$$\begin{aligned}
&MULT(i0, i1, i2, i3, m0, m1, m2, m3) \wedge \\
&MULT(i0, i1, i4, i5, m4, m5, m6, m7) \wedge \\
&MULT(i2, i3, i4, i5, m8, m9, m10, m11) \rightarrow \\
&\quad (ADDER(m0, m1, m2, m3, m4, m5, m6, m7, o0, o1, o2, o3, o4) \wedge \\
&\quad ADDER(m4, m5, m6, m7, m8, m9, m10, m11, o5, o6, o7, o8, o9) \rightarrow \\
&\qquad M3A2(i0, i1, i2, i3, i4, i5, o0, o1, o2, o3, o4, o5, o6, o7, o8, o9))
\end{aligned}$$

The reason a nested rule is used in this case is to give the order of antecedent processing in a parallel environment. That is, $ADDER$ patterns are processed only after $MULT$ patterns are instantiated so that the $ADDER$ patterns will use instantiated values for the variables m0 through m11 from the $MULT$ patterns.

$ADDER$ is a 4-bit full adder constructed from 4 1-bit full adders (as shown in Figure 5.2(c)), and is represented as

$$\begin{aligned}&FA(a0, b0, 0, s0, t1) \wedge\\ &FA(a1, b1, t1, s1, t2) \wedge\\ &FA(a2, b2, t2, s2, t3) \wedge\\ &FA(a3, b3, t3, s3, s4) \rightarrow\\ &\qquad ADDER(a0, a1, a2, a3, b0, b1, b2, b3, s0, s1, s2, s3, s4)\end{aligned}$$

$MULT$ is a 2-bit multiplier that as shown in Figure 5.2(d), and is represented as

$$\begin{aligned}&ANDG(a0, b0, c0) \wedge ANDG(a0, b1, t1) \wedge\\ &ANDG(a1, b0, t2) \wedge ANDG(a1, b1, t3) \wedge\\ &HA(t1, t2, c1, t4) \wedge HA(t3, t4, c2, c3) \rightarrow\\ &\qquad MULT(a0, a1, b0, b1, c0, c1, c2, c3)\end{aligned}$$

FA is a 1-bit full adder constructed from 2 half adders as shown in Figure 5.2(b).

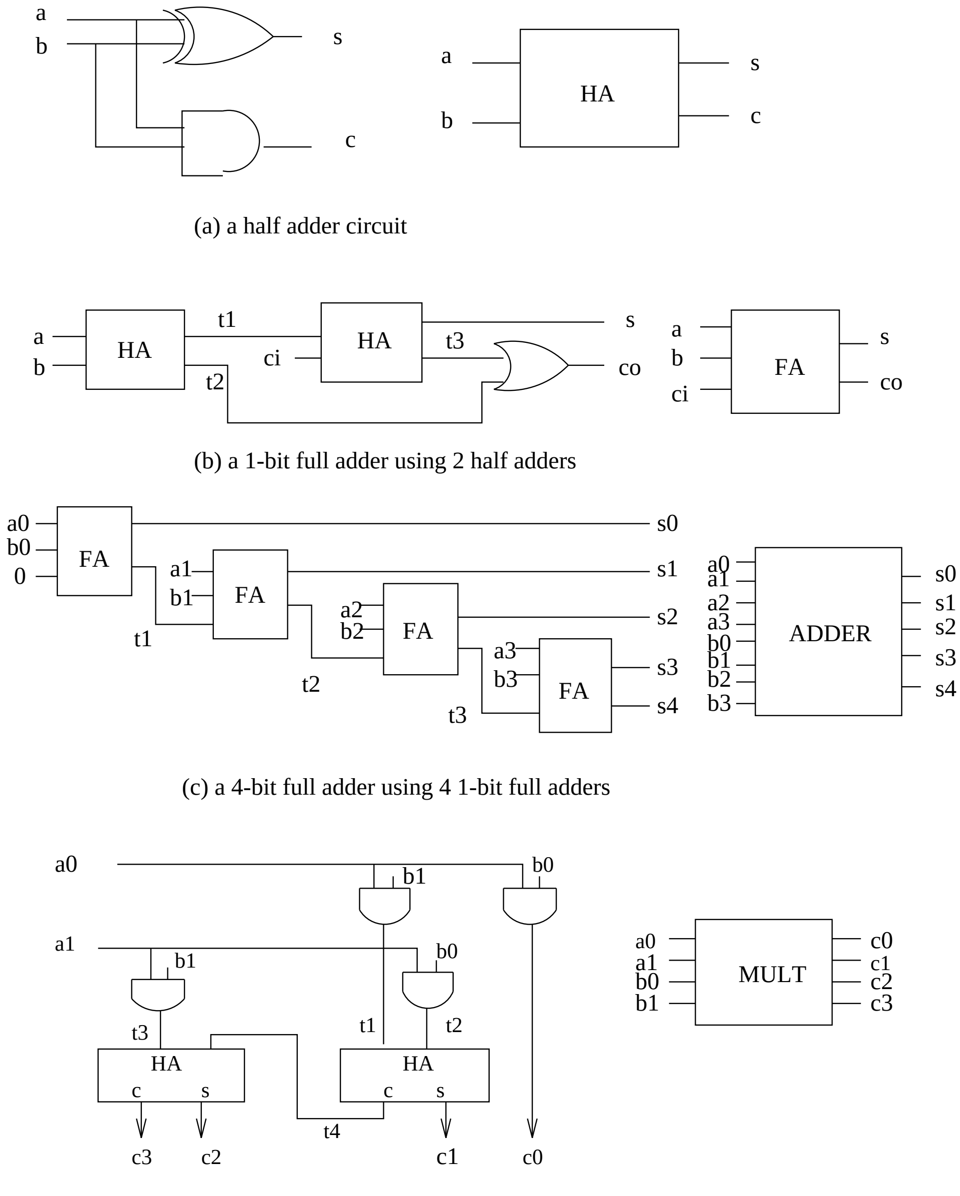
$$\begin{aligned}&HA(a, b, t1, t2) \wedge HA(t1, ci, s, t3) \wedge ORG(t2, t3, co) \rightarrow\\ &\qquad FA(a, b, ci, s, co)\end{aligned}$$

HA is a half adder circuit constructed from an XOR gate and an AND gate as shown in Figure 5.2(a).

$$XORG(a, b, s) \wedge ANDG(a, b, c) \rightarrow HA(a, b, s, c)$$

The primitive level of the definitional method is the gate level. Therefore, definitions of logical gates are given by facts which are similar to the standard truth tables. For example, an AND gate is represented by

$$ANDG(0, 0, 0)$$



(a) a half adder circuit

(b) a 1-bit full adder using 2 half adders

(c) a 4-bit full adder using 4 1-bit full adders

(d) a 2 by 2 multiplier

Figure 5.2: Hierarchical representation of adders and a multiplier

$ANDG(0, 1, 0)$

$ANDG(1, 0, 0)$

$ANDG(1, 1, 1)$

Here, $ANDG(1, 0, 0)$ indicates that if the first input is 1 and the second input is 0, the output is 0. Other logic gates are similarly represented.

This completes a hierarchical representation of the `M3A2` circuit with 2 bit inputs and 5 bit outputs. In general, the steps of a validation process are: (1) provide test input values (e.g., $in1$, $in2$, and $in3$ values of `M3A2`); (2) measure the actual output values generated from the given input values (e.g., $out1$ and $out2$ values of `M3A2`); and (3) feed the input and the measured output values to the rule of the circuit to check if the measured output values are correct according to the model of the circuit.

We are particularly interested in an exhaustive simulation that tests output values for all possible combinations of input values. Since there are three inputs with 2 bits each, there are total 64 (3 inputs * 2 bits = 6 bits, and $2^6 = 64$) cases of simulation. Each simulation is performed by a query with the form of

$M3A2(i0, i1, i2, i3, i4, i5, o0, o1, o2, o3, o4, o5, o6, o7, o8, o9)$

where $i0$ and $i1$ denote $in1$, $i2$ and $i3$ denote $in2$, $i4$ and $i5$ denote $in3$, $o0$ through $o4$ denote $out1$, and $o5$ through $o9$ denote $out2$. For example,

$M3A2(1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0)$ (`q46`)

asks if the measured outputs ($out1$ = 5, $out2$ = 8) are correct for given input values ($in1$ = 1, $in2$ = 3, $in3$ = 2).

query no.	query	$(in1, in2, in3 \rightarrow out1, out2)$
`q1`	$M3A2(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(0, 0, 0 \rightarrow 0, 0)$
`q2`	$M3A2(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(1, 0, 0 \rightarrow 0, 0)$
`q3`	$M3A2(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(2, 0, 0 \rightarrow 0, 0)$
`q4`	$M3A2(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(3, 0, 0 \rightarrow 0, 0)$
`q5`	$M3A2(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(0, 1, 0 \rightarrow 0, 0)$
`q6`	$M3A2(1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(1, 1, 0 \rightarrow 1, 0)$
`q7`	$M3A2(0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(2, 1, 0 \rightarrow 2, 0)$
`q8`	$M3A2(1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)?$	$(3, 1, 0 \rightarrow 3, 0)$

`q9`	$M3A2(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)?$	$(0,2,0 \rightarrow 0,0)$
`q10`	$M3A2(1,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0)?$	$(1,2,0 \rightarrow 2,0)$
`q11`	$M3A2(0,1,0,1,0,0,0,0,1,0,0,0,0,0,0,0)?$	$(2,2,0 \rightarrow 4,0)$
`q12`	$M3A2(1,1,0,1,0,0,0,1,1,0,0,0,0,0,0,0)?$	$(3,2,0 \rightarrow 6,0)$
`q13`	$M3A2(0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0)?$	$(0,3,0 \rightarrow 0,0)$
`q14`	$M3A2(1,0,1,1,0,0,1,1,0,0,0,0,0,0,0,0)?$	$(1,3,0 \rightarrow 3,0)$
`q15`	$M3A2(0,1,1,1,0,0,0,1,1,0,0,0,0,0,0,0)?$	$(2,3,0 \rightarrow 6,0)$
`q16`	$M3A2(1,1,1,1,0,0,1,0,0,1,0,0,0,0,0,0)?$	$(3,3,0 \rightarrow 9,0)$
`q17`	$M3A2(0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0)?$	$(0,0,1 \rightarrow 0,0)$
`q18`	$M3A2(1,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0)?$	$(1,0,1 \rightarrow 1,1)$
`q19`	$M3A2(0,1,0,0,1,0,0,1,0,0,0,0,1,0,0,0)?$	$(2,0,1 \rightarrow 2,2)$
`q20`	$M3A2(1,1,0,0,1,0,1,1,0,0,0,1,1,0,0,0)?$	$(3,0,1 \rightarrow 3,3)$
`q21`	$M3A2(0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0)?$	$(0,1,1 \rightarrow 0,1)$
`q22`	$M3A2(1,0,1,0,1,0,0,1,0,0,0,0,1,0,0,0)?$	$(1,1,1 \rightarrow 2,2)$
`q23`	$M3A2(0,1,1,0,1,0,0,0,1,0,0,1,1,0,0,0)?$	$(2,1,1 \rightarrow 4,3)$
`q24`	$M3A2(1,1,1,0,1,0,0,1,1,0,0,0,0,1,0,0)?$	$(3,1,1 \rightarrow 6,4)$
`q25`	$M3A2(0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0)?$	$(0,2,1 \rightarrow 0,2)$
`q26`	$M3A2(1,0,0,1,1,0,1,1,0,0,0,1,1,0,0,0)?$	$(1,2,1 \rightarrow 3,3)$
`q27`	$M3A2(0,1,0,1,1,0,0,1,1,0,0,0,0,1,0,0)?$	$(2,2,1 \rightarrow 6,4)$
`q28`	$M3A2(1,1,0,1,1,0,1,0,0,1,0,1,0,1,0,0)?$	$(3,2,1 \rightarrow 9,5)$
`q29`	$M3A2(0,0,1,1,1,0,0,0,0,0,0,1,1,0,0,0)?$	$(0,3,1 \rightarrow 0,3)$
`q30`	$M3A2(1,0,1,1,1,0,0,0,1,0,0,0,0,1,0,0)?$	$(1,3,1 \rightarrow 4,4)$
`q31`	$M3A2(0,1,1,1,1,0,0,0,0,1,0,1,0,1,0,0)?$	$(2,3,1 \rightarrow 8,5)$
`q32`	$M3A2(1,1,1,1,1,0,0,0,1,1,0,0,1,1,0,0)?$	$(3,3,1 \rightarrow 12,6)$
`q33`	$M3A2(0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)?$	$(0,0,2 \rightarrow 0,0)$
`q34`	$M3A2(1,0,0,0,0,1,0,1,0,0,0,0,1,0,0,0)?$	$(1,0,2 \rightarrow 2,2)$
`q35`	$M3A2(0,1,0,0,0,1,0,0,1,0,0,0,0,1,0,0)?$	$(2,0,2 \rightarrow 4,4)$
`q36`	$M3A2(1,1,0,0,0,1,0,1,1,0,0,0,1,1,0,0)?$	$(3,0,2 \rightarrow 6,6)$
`q37`	$M3A2(0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,0)?$	$(0,1,2 \rightarrow 0,2)$
`q38`	$M3A2(1,0,1,0,0,1,1,1,0,0,0,0,0,1,0,0)?$	$(1,1,2 \rightarrow 3,4)$
`q39`	$M3A2(0,1,1,0,0,1,0,1,1,0,0,0,1,1,0,0)?$	$(2,1,2 \rightarrow 6,6)$
`q40`	$M3A2(1,1,1,0,0,1,1,0,0,1,0,0,0,0,1,0)?$	$(3,1,2 \rightarrow 9,8)$
`q41`	$M3A2(0,0,0,1,0,1,0,0,0,0,0,0,0,1,0,0)?$	$(0,2,2 \rightarrow 0,4)$
`q42`	$M3A2(1,0,0,1,0,1,0,0,1,0,0,0,1,1,0,0)?$	$(1,2,2 \rightarrow 4,6)$
`q43`	$M3A2(0,1,0,1,0,1,0,0,0,1,0,0,0,0,1,0)?$	$(2,2,2 \rightarrow 8,8)$
`q44`	$M3A2(1,1,0,1,0,1,0,0,1,1,0,0,1,0,1,0)?$	$(3,2,2 \rightarrow 12,10)$
`q45`	$M3A2(0,0,1,1,0,1,0,0,0,0,0,0,1,1,0,0)?$	$(0,3,2 \rightarrow 0,6)$
`q46`	$M3A2(1,0,1,1,0,1,1,0,1,0,0,0,0,0,1,0)?$	$(1,3,2 \rightarrow 5,8)$
`q47`	$M3A2(0,1,1,1,0,1,0,1,0,1,0,0,1,0,1,0)?$	$(2,3,2 \rightarrow 10,10)$
`q48`	$M3A2(1,1,1,1,0,1,1,1,1,1,0,0,0,1,1,0)?$	$(3,3,2 \rightarrow 15,12)$
`q49`	$M3A2(0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0)?$	$(0,0,3 \rightarrow 0,0)$
`q50`	$M3A2(1,0,0,0,1,1,1,1,0,0,0,1,1,0,0,0)?$	$(1,0,3 \rightarrow 3,3)$
`q51`	$M3A2(0,1,0,0,1,1,0,1,1,0,0,0,1,1,0,0)?$	$(2,0,3 \rightarrow 6,6)$
`q52`	$M3A2(1,1,0,0,1,1,1,0,0,1,0,1,0,0,1,0)?$	$(3,0,3 \rightarrow 9,9)$
`q53`	$M3A2(0,0,1,0,1,1,0,0,0,0,0,1,1,0,0,0)?$	$(0,1,3 \rightarrow 0,3)$
`q54`	$M3A2(1,0,1,0,1,1,0,0,1,0,0,0,1,1,0,0)?$	$(1,1,3 \rightarrow 4,6)$
`q55`	$M3A2(0,1,1,0,1,1,0,0,0,1,0,1,0,0,1,0)?$	$(2,1,3 \rightarrow 8,9)$
`q56`	$M3A2(1,1,1,0,1,1,0,0,1,1,0,0,0,1,1,0)?$	$(3,1,3 \rightarrow 12,12)$
`q57`	$M3A2(0,0,0,1,1,1,0,0,0,0,0,0,1,1,0,0)?$	$(0,2,3 \rightarrow 0,6)$

q58	$M3A2(1,0,0,1,1,1,1,0,1,0,0,1,0,0,1,0)?$	$(1,2,3 \rightarrow 5,9)$
q59	$M3A2(0,1,0,1,1,1,0,1,0,1,0,0,0,1,1,0)?$	$(2,2,3 \rightarrow 10,12$
q60	$M3A2(1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,0)?$	$(3,2,3 \rightarrow 15,15)$
q61	$M3A2(0,0,1,1,1,1,0,0,0,0,0,1,0,0,1,0)?$	$(0,3,3 \rightarrow 0,9)$
q62	$M3A2(1,0,1,1,1,1,0,1,1,0,0,0,0,1,1,0)?$	$(1,3,3 \rightarrow 6,12)$
q63	$M3A2(0,1,1,1,1,1,0,0,1,1,0,1,1,1,1,0)?$	$(2,3,3 \rightarrow 18,18)$
q64	$M3A2(1,1,1,1,1,1,0,1,0,0,1,0,1,0,0,1)?$	$(3,3,3 \rightarrow 12,15)$

An interesting phenomenon in this simulation is that a large number of associational facts for the adders and the multiplier are derived as intermediate results. For example, during processing `q1`, the system derives and asserts the following fact saying that $0 + 0 = 0$.

$$ADDER(0,0,0,0,0,0,0,0,0,0,0,0,0) \qquad \text{(a)}$$

This fact is an instance of the consequent pattern of the 4-bit adder rule

$$ADDER(a0,a1,a2,a3,b0,b1,b2,b3,s0,s1,s2,s3,s4) \qquad \text{(b)}$$

Not surprisingly, some of subsequent queries also try to derive (a) from the 4-bit adder rule whenever either $in1$ or $in3$ is 0. But this is a redundant step, since the only fact you can derive in this situation is (a). We can avoid this duplicate and redundant step by applying Shadowing Principle 2 that uses the most common general instance (`mgci`). The `mgci` of (a) and (b) is always (a), and as a result, Shadowing Principle 2 blocks the 4-bit adder rule from being used since the `mgci` is ground and asserted. This shadowing can be applied for any $ADDER$ fact that is used again in subsequent queries.

Figure 5.3 compares the CPU time results graphically. As you can observe in the graph, the CPU times are increasing steadily in SNIP2.1 because of the knowledge base expansion that produced more pattern matchings and more deduction branches. Compared to this, the CPU time in SNIP2.2 drops significantly in many cases, e.g. `q17, q33, q37, q49, q53`, and so on. This happens when Shadowing Principle 2 is able to to exploit already derived facts and hence avoid the activation of rules. However, those queries that ask unknown facts caused a significant increase of CPU time, which is obvious.

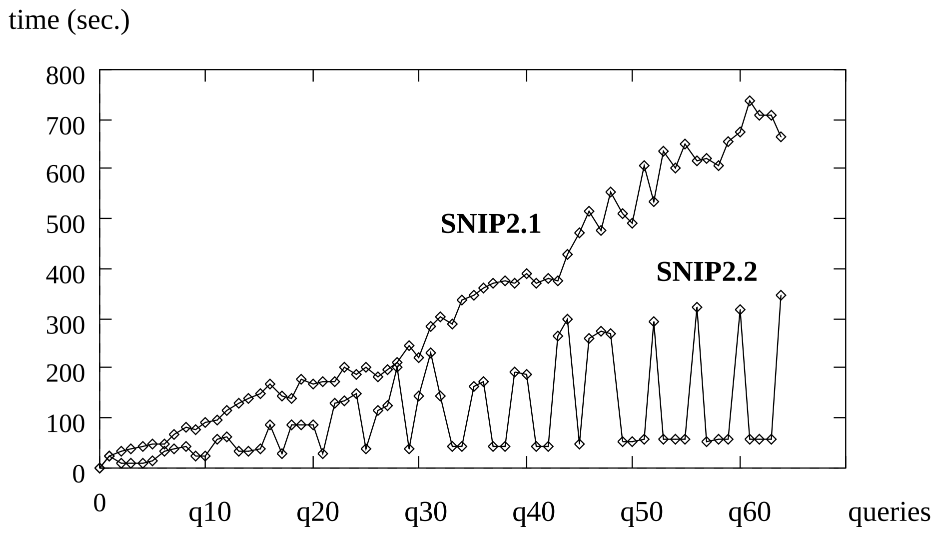


Figure 5.3: CPU time comparisons of `M3A2` simulation

Chapter 6

Conclusions

6.1 Summary

We have presented schemes that allow deductive reasoning systems to learn from and exploit previous experience, and consequently to make future reasoning more efficient and more effective. Both the wide applicability of general knowledge and the speed of specific knowledge are achieved by managing a multi-level knowledge structure and developing techniques of migrating general knowledge to specific knowledge and preventing general knowledge from being used when specific knowledge migrated from it is available and applicable. By these techniques, the system is able to decide the best possible branch in a deduction tree by choosing an appropriate rule among many applicable rules. Experience is represented by the instance relationship between the general knowledge and the specific knowledge migrated from it, and stored in the expertise base. This information about instances contributes to the selection of the best possible deduction branch. Learning and reasoning are integrated by managing learning modules as subcomponents of the inference engine in a reasoning system.

Two learning modules, knowledge migration and knowledge shadowing, are provided for experience-based rule selection. Knowledge migration migrates general knowledge to specific knowledge during reasoning, which causes knowledge redundancy by storing migrated knowledge in the knowledge base. Experiential information is obtained during knowledge migration. Knowledge shadowing recognizes redundancy, recalls the experiential information from the expertise base and compares it with the current situation, and prevents those rules that are determined to be unnecessary from being used. Decision-making about which rule must be blocked and which rule must be used depends on the content of the expertise base and the nature of the current query.

We have also presented techniques of implementing efficient rules of inference in natural deduction systems by caching and recalling the history of rule activation steps that alleviate duplicate pattern matchings and binding conflict resolutions. To reduce the complexity of manipulating rule activation steps from exponential to polynomial, methods of distributing the information about rule activation steps were developed that minimize the number of activation steps and the number of substitution compatibility tests among shared variables. **S-indexing** distributes rule activation information by using the variable substitution of the instances, and is applied to the rules whose antecedents have the same set of variables. **P-tree** is applied to conjunctive rules, and distributes rule activation information over the nodes of a **P-tree** that is compiled from the set of antecedents of the rule.

6.2 Contributions

Contributions of this research are summarized as below.

(1) This work is a formal approach to the integration of automated reasoning and machine learning.

Many machine learning algorithms have dealt with some kind of experience information to improve overall performance, but not many of them have mentioned a concrete form of experience. We have formally defined deduction experience as the amalgamation of instance set information and origin set information. Then, we also have formalizations of how this experience can be accumulated and how it can be utilized. The accumulation of experience has been achieved by knowledge migration that is defined as a system state change function where a system state includes the contents of the knowledge base and the expertise base. The utilization of experience has been achieved by two knowledge shadowing principles, and the completeness of these principles have been proved.

(2) We employ inexpensive, but very effective learning components.

The ultimate goal of speedup learning is system efficiency, and more specifically, faster processing of similar problems. Hence, to evaluate the overall quality of a learning algorithm, the performance metric should include not only the problem solving costs but also the amount of effort consumed by the learning process. In this sense, we prefer simple learning algorithms with low complexity that are strong enough to make a large difference in system behavior. Our view of EBL and other learning methods is that those systems spend a considerable amount of time to learn or apply problem solving experience. For instance, it is costly to traverse the explanation structure to build a new description in EBL, or to match a number of complex chunks in the chunking method. Our system employs a rather simple learning component that is incorporated within a general deductive inference engine, and as a result, the cost of learning and the cost of applying learned knowledge to subsequent problem solving can be negligible compared to the overall process.

(3) Knowledge redundancy is well exploited.

One of the main problems in speedup learning systems is performance degradation caused by the increased computation cost of matching the patterns of the learned rules in addition to matching the original rules. In fact, the learned rules in speedup learning systems belong to the system's deductive closure, and hence the addition of the learned rules to the knowledge base causes knowledge redundancy. To avoid system slowdown caused by knowledge redundancy, knowledge migration and knowledge shadowing provide a simple but clever indexing mechanism linked with heuristics for managing learned knowledge. The control of knowledge redundancy is done by relating learned rules with the original rules in a principled fashion. A simple indexing mechanism also reduces the cost of applying experience to future problem solving.

(4) We bridge the gap between performance and expressibility in deductive reasoning systems, especially in natural deduction systems.

Normally, these two properties seldom come together. Representations that are expressible are slow, and representations that are fast to execute are not expressible. For instance, the logic programming community based on Horn clause representations is more concerned with performance. Hence, logic programs are usually very fast to execute. However, Horn-clause representations for most natural language expressions require a large number of clauses that are unreadable and far from what humans might represent. On the other hand, expressibility is more emphasized in most systems based on human reasoning including natural deduction systems, semantic network representation/reasoning, conceptual graphs, and so on. Also the introduction of non-standard connectives and quantifiers and the exploitation of nested rule representations lead to representation that is expressible, readable, and closer to the mode of human reasoning. However, the implementation of rules of inference for these connectives has intrinsically exponential complexity. Even if we emphasize the expressibility, this seems to be intolerable. Therefore, in order to have expressible representations as well as reasonable reasoning performance, **S-indexing** and **P-tree** are developed to implement the rules of inference of the various connectives and quantifiers in polynomial time. In addition, these methods can be used in backward chaining, forward chaining, and also bi-directional chaining, so they provide more flexibility.

6.3 Future Work

Some issues to be considered in the future are:

(1) The origin set information can be more extensively used.

Although the origin set information is a part of deduction experience, its usage in knowledge shadowing has been minimal. Compared to the instance set information which only provides the instance relationship between a general rule and a specific rule, the origin set information provides more flexible information about which knowledge was used to derive new knowledge. In this sense, we can trace more than one level of inference chaining by using the origin set information. As a result, if a rule is more deeply nested, it is possible to shadow more than one general rule. In order to do that, we might have to use the subset-superset relationship among several origin sets. But, we still have to use the instance set information in order to differentiate between a specific query and a general query. Note that Shadowing Principle 1 is effective as long as the level of rule nesting is limited to one, since it can achieve the same functionality without employing origin set information.

(2) Tests are to be done in parallel environments.

Performance measurements for the evaluation of several learning techniques are done mainly in a sequential processing environment. As we have discussed in Chapter 3, the effect of shadowing is different in a parallel environment, since both general rule and specific rule can be executed concurrently.

For example, suppose a general rule $\mathbf{r}_g$ and a specific rule $\mathbf{r}_s$ are applicable. In a sequential environment, the total cost is approximately $C_{RULE}(\mathbf{r}_g) + C_{RULE}(\mathbf{r}_s)$ without shadowing because both rules are triggered and executed. With the shadowing method, this cost is reduced to $C_{RULE}(\mathbf{r}_s) + C_{\mathcal{EB}}$ when $\mathbf{r}_g$ is shadowed. $C(ExpRet)$ refers to the cost of retrieving the expertise information. Therefore, shadowing will be effective when $C_{\mathcal{EB}}$ is much smaller than $C_{RULE}(\mathbf{r}_g)$.

In a parallel environment where $\mathbf{r}_g$ and $\mathbf{r}_s$ are executed concurrently, the total cost without shadowing is approximately $C_{RULE}(\mathbf{r}_g)$, since $C_{RULE}(\mathbf{r}_g)$ is generally greater than $C_{RULE}(\mathbf{r}_s)$. With shadowing, if $\mathbf{r}_g$ is shadowed, the total cost becomes $C_{RULE}(\mathbf{r}_s) + C_{\mathcal{EB}}$. Therefore, if $C_{\mathcal{EB}}$ is small, the shadowing methods are also effective in parallel environments.

Bibliography

[Anderson, 1989] John R. Anderson. A theory of the origins of human knowledge. *Artificial Intelligence*, 40:313–351, 1989.

[Bain, 1986] W. M. Bain. A case-based reasoning system for subjective assessment. In *Proceedings of National Conference on Artificial Intelligence*, pages 523–527, Philadelphia, PA, August 1986. Morgan Kaufmann, Los Altos.

[Bibel, 1986] Wolfgang Bibel. Methods of automated reasoning. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence: An Advanced Course, Lecture Notes in Computer Science No. 232*, pages 171–217. Springer-Verlag, Berlin, New York, 1986.

[Buntine, 1988] Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, September 1988.

[Burstein, 1989] M. H. Burstein. Analogy vs. CBR: The purpose of mapping. In *Proceedings of the DARPA Workshop on Case-Based Reasoning*, pages 133–136, Pensacola Beach, FL, 1989. Morgan Kaufmann.

[Chandrasekaran and Milne, 1985] B. Chandrasekaran and Robert Milne. Reasoning about structure, behavior and function. *SIGART Newsletter*, 93:4–7, July 1985.

[Chandrasekaran and Mittal, 1983] B. Chandrasekaran and Sanjay Mittal. Deep versus compiled knowledge approaches to diagnostic problem solving. *International Journal of Man-Machine Studies*, 19:425–436, 1983.

[Chandrasekaran *et al.*, 1985] B. Chandrasekaran, T. Bylander, and V. Sembugamoorthy. Functional representation and behavior composition by consolidation : Two aspects of reasoning about devices. Technical Report 85-BC-SIGART, Department of Computer and Information Science, Ohio State University, 1985.

[Choi and Shapiro, 1991a] Joongmin Choi and Stuart C. Shapiro. Experience-based deductive learning. In *Third International Conference on Tools for Artificial Intelligence TAI '91*, pages 502–503. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[Choi and Shapiro, 1991b] Joongmin Choi and Stuart C. Shapiro. Learning in deduction by knowledge migration and shadowing. In *AAAI-91 Workshop on Knowledge Acquisition: From Science to Technology to Tools*, Anaheim, CA, 1991. AAAI.

[Choi and Shapiro, 1992] Joongmin Choi and Stuart C. Shapiro. Efficient implementation of non-standard connectives and quantifiers in deductive reasoning systems. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 381–390. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Choi, 1990] Joongmin Choi. Fast reasoning with non-standard connectives and quantifiers in SNePS. In *Second Workshop on Current Trends in SNePS*, SUNY at Buffalo, October 1990.

[Clocksin, 1987] W. F. Clocksin. Logic programming and digital circuit analysis. *Journal of Logic Programming*, pages 59–82, 1987.

[Cohn, 1985] Anthony G. Cohn. On the solution of Schubert's streamroller in many sorted logic. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1169–1174, Los Angeles, CA, August 1985. Morgan Kaufmann, Los Altos.

[Conery and Kibler, 1983] John S. Conery and Dennis F. Kibler. AND parallelism in logic programs. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 539–543, Karlsruhe, Germany, August 1983. William Kaufmann, Los Altos.

[Davis and Hamscher, 1988] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting. AI Memo AIM-1059, MIT AI Laboratory, 1988.

[Davis, 1984] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[DeJong and Mooney, 1986] Gerald DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[Ellman, 1989] Thomas Ellman. Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, 1989.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Forgy, 1982] Charles L. Forgy. RETE: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Forsyth and Rada, 1986] Richard Forsyth and Roy Rada. *Machine Learning: Applications in Expert Systems and Information Retrieval.* Ellis Horwood, Chichester, 1986.

[Genesereth, 1984] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[Greiner and Likuski, 1989] Russell Greiner and Joseph Likuski. Incorporating redundant learned rules: A preliminary formal analysis. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 744–749, Detroit, MI, August 1989. Morgan Kaufmann, Los Altos.

[Greiner, 1991] Russell Greiner. Finding optimal derivation strategies in redundant knowledge bases. *Artificial Intelligence*, 50:95–115, 1991.

[Gupta *et al.*, 1989] Anoop Gupta, Charles Forgy, and Allen Newell. High-speed implementations of rule-based systems. *ACM Trans. on Computer Systems*, 7(2):119–146, 1989.

[Hammond, 1986] Kristian Hammond. *Case-based Planning: An Integrated Theory of Planning, Learning, and Memory.* PhD thesis, Yale University, 1986.

[Hammond, 1989] Kristian J. Hammond. Chef. In C. Reisbeck and R. Schank, editors, *Inside Case-Based Reasoning*, chapter 6. Lawrence Erlbaum Assoc., Hillsdale, NJ, 1989.

[Hamscher and Davis, 1987] Walter Hamscher and Randall Davis. Issues in model based troubleshooting. AI Memo AIM-893, MIT AI Laboratory, 1987.

[Hart, 1982] Peter E. Hart. Directions for AI in the eighties. *SIGART Newsletter*, 79:11–16, January 1982.

[Hull, 1986] Richard G. Hull. A new design for SNIP the SNePS Inference Package. SNeRG Technical Note 14, Dept. of Computer Science, State University of New York at Buffalo, 1986.

[Keller, 1990] Richard M. Keller. In defense of compilation. In *Proceedings of AAAI-90 Workshop on Model Based Reasoning*, pages 22–31, Boston, MA, 1990. AAAI.

[Kolodner and Kolodner, 1987] Janet L. Kolodner and Robert M. Kolodner. Using experience in clinical problem solving: Introduction and framework. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-17(3):420–431, 1987.

[Kolodner and Simpson, 1984] J. L. Kolodner and R. L. Simpson. Problem solving and dynamic memory. In J. L. Kolodner and C. K. Riesbeck, editors, *Memory, Experience, and Reasoning*. Hillsdale, NJ, 1984.

[Kolodner *et al.*, 1985] Janet L. Kolodner, Robert L. Simpson, and Katia Sycara-Cyranski. A process model of case-based reasoning in problem solving. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 284–290, Los Angeles, CA, August 1985. Morgan Kaufmann, Los Altos.

[Kolodner, 1983] Janet L. Kolodner. Towards an understanding of the role of experience in the evolution from novice to expert. *International Journal of Man-Machine Studies*, 19:497–518, 1983.

[Kolodner, 1984] Janet L. Kolodner. *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1984.

[Koton, 1985] Phyllis A. Koton. Empirical and model-based reasoning in expert systems. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 297–299, 1985.

[Koton, 1988] Phyllis A. Koton. *Using Experience in Learning and Problem Solving.* PhD thesis, Massachusetts Institute of Technology, 1988.

[Koton, 1989] Phyllis A. Koton. A method for improving the efficiency of model-based reasoning systems. *Applied Artificial Intelligence*, 3(2–3):357–366, 1989.

[Laird *et al.*, 1984] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Towards chunking as a general learning mechanism. In *Proceedings of National Conference on Artificial Intelligence*, pages 188–192, Austin, TX, August 1984. William Kaufmann, Los Altos.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Levi *et al.*, 1992] Keith R. Levi, David L. Perschbacher, Mark A. Hoffman, Christopher A. Miller, Barry B. Druhan, and Valerie L. Shalin. An explanation-based-learning approach to knowledge compilation: A pilot's associate application. *IEEE Expert*, 7(3):44–51, June 1992.

[Markovitch and Scott, 1988] Shaul Markovitch and Paul D. Scott. The role of forgetting in learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 459–465, Ann Arbor, MI, 1988. Morgan Kaufmann, Los Altos.

[Martins and Shapiro, 1988] João P. Martins and Stuart C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35:25–79, 1988.

[Martins, 1983] João P. Martins. Reasoning in muliple belief spaces. Technical Report 203, Department of Computer Science, State University of New York at Buffalo, May 1983.

[McDermott and Forgy, 1978] J. M. McDermott and C. Forgy. Production system conflict resolution strategies. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*. Academic Press, New York, 1978.

[McKay and Shapiro, 1981] Donald P. McKay and Stuart C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 368–374, Vancouver, Canada, August 1981.

[Michalski, 1983] Ryszard S. Michalski. A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 83–134. Morgan Kaufmann, Los Altos, 1983.

[Milne, 1986] Robert Milne. Fault diagnosis using structure and function. In D. Sriram and R. Adey, editors, *Applications of Artificial Intelligence in Engineering Problems, 1st International Conference*, pages 1043–1054, Berlin, New York, 1986. Springer-Verlag.

[Minton, 1988] Steve Minton. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, Boston, 1988.

[Miranker, 1987] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 42–47, Seattle, WA, July 1987. Morgan Kaufmann, Los Altos.

[Mitchell *et al.*, 1986] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[Mitchell, 1982] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mooney, 1990] R. J. Mooney. *A General Explanation-Based Learning Mechanism and Its Application to Narrative Understanding.* Pitman, London, 1990.

[Nayak *et al.*, 1988] P. Nayak, A. Gupta, and P. Rosenbloom. Comparison of the Rete and Treat production matchers for SOAR: A summary. In *Proceedings of National Conference on Artificial Intelligence*, pages 693–698, St. Paul, Mn, August 1988. Morgan Kaufmann, Los Altos.

[Neves and Anderson, 1981] David M. Neves and John R. Anderson. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In John R. Anderson, editor, *Cognitive Skills and Their Acquisition*, pages 57–84. Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[Plotkin, 1970] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Elsevier North-Holland, New York, 1970.

[Porter *et al.*, 1990] Bruce W. Porter, Ray Bareiss, and Robert C. Holte. Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, 45(1 and 2), 1990.

[Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of ACM*, 12:23–41, 1965.

[Rosenbloom and Laird, 1986] Paul S. Rosenbloom and John E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of National Conference on Artificial Intelligence*, pages 561–567, Philadelphia, PA, August 1986. Morgan Kaufmann, Los Altos.

[Sauers, 1988] Ron Sauers. Controlling expert systems. In L. Bolc and M. J. Coombs, editors, *Expert System Applications*, pages 79–197. Springer-Verlag, Berlin, New York, 1988.

[Schank, 1982] R. Schank. *Dynamic Memory: A Theory of Learning in Computers and People*. Cambridge University Press, Cambridge, England, 1982.

[Shapiro and Rapaport, 1992a] Stuart C. Shapiro and William J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In Leslie Burkholder, editor, *Philoshophy and the Computer*, pages 75–91. Westiew Press, Boulder, CO, 1992.

[Shapiro and Rapaport, 1992b] Stuart C. Shapiro and William J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, January–March 1992.

[Shapiro *et al.*, 1981] S. C. Shapiro, D. P. McKay, J. Martins, and E. Morgado. SNePSLOG: A "higher order" logic programming language. SNeRG Technical Note 8, Department of Computer Science, SUNY at Buffalo, 1981. Presented at the Workshop on Logic Programming for Intelligent Systems, R.M.S. Queen Mary, Long Beach, CA.

[Shapiro *et al.*, 1982] Stuart C. Shapiro, João Martins, and Donald P. McKay. Bi-directional inference. In *Proceedings of the Fourth Annual Meeting of the Cognitive Science Society*, pages 90–93, Ann Arbor, MI, 1982.

[Shapiro, 1977] Stuart C. Shapiro. Compiling deduction rules from a semantic network into a set of processes. In *Abstracts of Workshop on Automatic Deduction*, MIT, Cambridge, MA, 1977.

[Shapiro, 1979a] Stuart. C. Shapiro. Numerical quantifiers and their use in reasoning with negative information. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 791–796, Tokyo, Japan, August 1979.

[Shapiro, 1979b] Stuart C. Shapiro. SNePS semantic network processing system. In N.V. Findler, editor, *Associative Networks : Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[Shapiro, 1979c] Stuart. C. Shapiro. Using non-standard connectives and quantifiers for representing deduction rules in a semantic network. Invited paper presented at Current Aspects of AI Research, a seminar held at the Electrotechnical Laboratory, Tokyo, 1979.

[Shapiro, 1986] S. C. Shapiro. Symmetric relations, intensional individuals, and variable binding. *Proceedings of the IEEE*, 74(10):1354–1363, 1986.

[Shavlik and Dietterich, 1990] Jude W. Shavlik and Thomas G. Dietterich. Improving the efficiency of a problem solver. In *Readings in Machine Learning*, chapter 4. Morgan Kaufmann, Palo Alto, 1990.

[Simon, 1983] Herbert A. Simon. Why should machines learn? In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 25–37. Morgan Kaufmann, Los Altos, 1983.

[Simpson, 1985] R. L. Simpson. A computer model of case-based reasoning in problem solving : An investigation in the domain of dispute mediation. Technical Report GIT-ICS-85/18, School of Information and Computer Science, Georgia Institute of Technology, 1985.

[Smith, 1989] David E. Smith. Controlling backward inference. *Artificial Intelligence*, 39:145–208, 1989.

[Stickel, 1985] M. E. Stickel. Automated deduction by theory resolution. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1181–1186, Los Angeles, CA, August 1985. Morgan Kaufmann, Los Altos.

[Stickel, 1986] Mark E. Stickel. Schubert's streamroller problem: Formulations and solutions. *Journal of Automated Reasoning*, 2:89–101, 1986.

[Summers, 1972] G. Summers. *Test Your Logic*. Dover, New York, 1972.

[Sycara, 1987] E. P. Sycara. Resolving adversarial conflicts : An approach integrating case-based and analytic methods. Technical Report GIT-ICS-87/26, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[Tambe and Newell, 1988] Milind Tambe and Allen Newell. Some chunks are expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 451–458, Ann Arbor, MI, 1988. Morgan Kaufmann, Los Altos.

[Tambe and Rosenbloom, 1989] Milind Tambe and Paul Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 731–737, Detroit, MI, August 1989. Morgan Kaufmann, Los Altos.

[Walther, 1984] C. Walther. A mechanical solution of Schubert's streamroller by many sorted resolution. In *Proceedings of National Conference on Artificial Intelligence*, pages 330–334, Austin, TX, August 1984. William Kaufmann, Los Altos.

[Wos *et al.*, 1984] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-hall, Inc., Englewood Cliffs, NJ, 1984.

[Wos, 1987] Larry Wos. Some obstacles to the automation of reasoning, and the problem of redundant information. *Journal of Automated Reasoning*, 3:81–90, 1987.