University
of Southern
California

Induction as Knowledge Integration

Benjamin Douglas Smith

USC/Information Sciences Institute

December 1995
ISI/RR-95-431

INFORMATION
SCIENCES
INSTITUTE

310/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Induction as Knowledge Integration

Benjamin Douglas Smith

USC/Information Sciences Institute

December 1995
ISI/RR-95-431

REPORT DOCUMENTATION PAGE

FORM APPROVED
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1995	Research Report

4. TITLE AND SUBTITLE

Induction as Knowledge Integration

5. FUNDING NUMBERS

NASA-Ames: Cooperative Agreement NCC 2-538
ARPA/NCCOSC: N66001-95-C-6013

6. AUTHOR(S)

Benjamin Douglas Smith

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

USC INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY
MARINA DEL REY, CA 90292-6695

8. PERFORMING ORGANIZATON REPORT NUMBER

ISI/RR-95-431

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

ARPA
3701 N. Fairfax Dr.
Arlington, VA 22203-1714

NASA-Ames Research Center
Moffett Field, CA 94035-1000

NCCOSC RDT&E
DIV CODE 02211
53560 Hull Street
San Diego, CA 92152-5410

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT

UNCLASSIFIED/UNLIMITED

12B. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Accuracy and efficiency are the two main evaluation criteria for induction algorithms. One of the most powerful ways to improve performance along these dimensions is by integrating additional knowledge into the induction process. However, integrating knowledge that differs significantly from the knowledge already used by the algorithm usually requires rewriting the algorithm.

This dissertation presents KII, a Knowledge Integration framework for Induction, that provides a straightforward method for integrating knowledge into induction, and provides new insights into the effects of knowledge on the accuracy and complexity of induction. The idea behind KII is to express *all* knowledge uniformly as constraints and preferences on hypotheses. Knowledge is integrated by conjoining constraints and disjoining preferences. A hypothesis is induced from the integrated knowledge by finding a hypothesis consistent with all of the constraints and maximally preferred by the preferences.

Theoretically, just about any knowledge can be expressed in this manner. In practice, the constraint and preference languages determine both the knowledge that can be expressed and the complexity of identifying a consistent hypothesis. RS-KII, an instantiation of KII based on a very expressive set representation, is described. RS-KII can utilize the knowledge of at least two disparate induction algorithms--AQ-11 and CEA ("version spaces")--in addition to knowledge neither algorithm can utilize. It seems likely that RS-KII can utilize knowledge from other induction algorithms, as well as novel kinds of knowledge, but this is left for future work. RS-KII's complexity is comparable to these algorithms when using only the knowledge of a given algorithm, and in some cases RS-KII's complexity is dramatically superior. KII also provides new insights into the effects of knowledge on induction that are used to derive classes of knowledge for which induction is not computable.

14. SUBJECT TERMS

bias, induction, knowledge intensive induction, machine learning

15. NUMBER OF PAGES

183

16. PRICE CODE

17. SECURITY CLASSIFICTION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to stay within the lines to meet optical scanning requirements.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element numbers(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the repor.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA- Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17.-19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contins classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

INDUCTION AS KNOWLEDGE INTEGRATION

by

Benjamin Douglas Smith

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

December 1995

Dedication

To my wife Christina, for all her love, support, and patience.

Acknowledgments

This work benefited from discussions with several people, particularly Haym Hirsh, who helped me to shape this work in its early stages. I would also like to thank the other members of my guidance committee, Yolanda Gil, Shankhar Rajamoney, Jean-Luc Gaudiot, and my advisor Paul Rosenbloom. The Soar group at USC provided many useful comments throughout the evolution of this work, and made the graduate program fun as well.

Thanks are also due to my parents for their support and their assurances that there was indeed a light at the end of the tunnel. Finally, I would not be where I am today without my wife Christina, who persevered through the ups and downs of my graduate school career, and provided me with love and encouragement throughout.

This work was partially supported by the National Aeronautics and Space Administration (NASA Ames Research Center) under cooperative agreement number NCC 2-538, and by the Information Systems Office of the Advanced Research Projects Agency (ARPA/ISO) and the Naval Command, Control and Ocean Surveillance Center RDT&E Division (NRaD) under contract number N66001-95-C-6013.

Contents

List Of Tables

List Of Figures

Abstract

Accuracy and efficiency are the two main evaluation criteria for induction algorithms. One of the most powerful ways to improve performance along these dimensions is by integrating additional knowledge into the induction process. However, integrating knowledge that differs significantly from the knowledge already used by the algorithm usually requires rewriting the algorithm.

This dissertation presents KII, a Knowledge Integration framework for Induction, that provides a straightforward method for integrating knowledge into induction, and provides new insights into the effects of knowledge on the accuracy and complexity of induction. The idea behind KII is to express *all* knowledge uniformly as constraints and preferences on hypotheses. Knowledge is integrated by conjoining constraints and disjoining preferences. A hypothesis is induced from the integrated knowledge by finding a hypothesis consistent with all of the constraints and maximally preferred by the preferences.

Theoretically, just about any knowledge can be expressed in this manner. In practice, the constraint and preference languages determine both the knowledge that can be expressed and the complexity of identifying a consistent hypothesis. RS-KII, an instantiation of KII based on a very expressive set representation, is described. RS-KII can utilize the knowledge of at least two disparate induction algorithms—AQ-11 and CEA ("version spaces")—in addition to knowledge neither algorithm can utilize. It seems likely that RS-KII can utilize knowledge from other induction algorithms, as well as novel kinds of knowledge, but this is left for future work. RS-KII's complexity is comparable to these algorithms when using only the knowledge of a given algorithm, and in some cases RS-KII's complexity is dramatically superior. KII also provides new insights into the effects of knowledge on induction that are used to derive classes of knowledge for which induction is not computable.

Chapter 1

Introduction

Induction is the process of going beyond what is known in order to reach a conclusion that is not deductively implied by the knowledge. One form of induction commonly studied in machine learning is *classifier learning* (e.g, [Mitchell, 1982, Brieman *et al.*, 1984, Quinlan, 1986, Pazzani and Kibler, 1992]). In this form of induction, the objective is to learn an operational description of a *classifier* that maps objects from some predefined universe of *instances* into classes. The induction process is provided with knowledge about the classes, but there is usually not enough knowledge to derive the classifier from the knowledge deductively. It is necessary to go beyond what is known in order to induce the classifier. The desired classifier is usually referred to as the *target concept*, and a classifier is often referred to as a *hypothesis*.

The classifier induced from the knowledge is not necessarily the target concept. In order to go beyond what is known deductively, it is necessary to make a number of assumptions and guesses, and these are not always correct. The induced classifier may classify some instances incorrectly with respect to the target concept. The degree to which the induced classifier makes the same classifications as the target concept is a measure of its accuracy. Hopefully, the induced hypothesis is fairly accurate, and ideally it is equivalent to the target concept.

The more knowledge there is to induce from, the more accurate the induced classifier can be. The assumptions and guesses made in the inductive process can be based on a broader base of knowledge, so there is less opportunity for making an incorrect decision. In the extreme case, there is enough knowledge to deduce the classifier, in which case no mistakes are made, and the induced classifier exactly matches the target concept.

An ideal induction algorithm would be able to utilize all of the available knowledge in order to maximize the accuracy of the induced classifier. In practice, most existing algorithms are written with certain kinds of knowledge in mind. If some of the available knowledge is not of these types, then the algorithm cannot use it, or can only use it with difficulty.

One way an algorithm can make use of such knowledge is to express it in terms of the kinds of knowledge already used by the algorithm. For instance, Quinlan suggested casting knowledge as pseudo-examples and providing the pseudo-examples to the induction algorithm as input [Quinlan, 1990]. However, it is not always possible to express new knowledge in terms of the existing knowledge, in which case the knowledge still cannot be used.

If the first approach fails, a second approach is to rewrite the algorithm to make use of the new knowledge. Rewriting an algorithm to utilize a new kind of knowledge is difficult. It also fails to solve the underlying problem. If yet another kind of knowledge is made available, the algorithm may have to be modified once again.

Existing induction algorithms therefore fall short of the ideal algorithm. Existing algorithms each use varying ranges of knowledge, but none can utilize all of the available knowledge in every learning scenario. The accuracy of the hypotheses induced by these algorithms is less than it could be since not all of the knowledge is being utilized. The methods mentioned above for integrating additional knowledge into the induction algorithm can be used to extend the algorithm, but these methods are of limited practicality.

1.1 Research Goals

The goal of this dissertation is to construct an induction algorithm that can utilize arbitrary knowledge, and therefore maximize the accuracy of the learned hypothesis. Utilizing all of the available knowledge may either increase or decrease the computational complexity compared to using less knowledge. The induction process is guided by the knowledge, so additional knowledge may reduce the amount of computational effort. However, there is more knowledge to utilize, and the cost of utilizing that knowledge may overwhelm any computational benefits it provides. Thus there is a potential trade-off between accuracy and computational complexity.

An algorithm that utilizes all of the knowledge may be prohibitively costly, or even uncomputable. It may be necessary to accept a reduction in accuracy in order to reduce the computational complexity. The second goal of this research is to investigate the ways in which expressiveness (breadth of utilizable knowledge) can be exchanged for computational cost.

The idea is to express all knowledge about the target concept uniformly in terms of constraints and preferences on the hypothesis space. Knowledge is integrated by conjoining the constraints and computing the union of the preferences. The integrated constraints and preferences specify a constrained optimization problem (COP). Solutions to the COP are the hypotheses identified by the knowledge as possible target concepts. The knowledge cannot discriminate further among these hypotheses, so one is selected arbitrarily as the induced hypothesis. This idea is embodied in KII, a Knowledge Integration framework for Induction. KII is described in Chapter 2.

Theoretically, any knowledge that can be expressed in terms of constraints and preferences can be utilized by KII. However, it may be computationally intractable to solve the COP resulting from these constraints and preferences. It may even be undecidable whether a given hypothesis is a solution to the COP. The constraint and preference languages determine what constraints and preferences can be expressed, and the computational complexity of solving the resulting COP. Thus, these languages provide a way of trading expressiveness for computability. Each choice of languages yields an operationalization of KII that makes a different trade off between expressiveness and complexity.

KII provides a method for integrating arbitrary knowledge into induction and formalizes the trade-offs that can be made between expressiveness and complexity in terms of the constraint and preference languages. This relationship is used to determine the most expressive languages for which induction is computable. These analyses, and the space of possible constraint and preference languages, appear in Chapter 3.

KII can generate algorithms at practical and interesting trade-off points. One such trade-off is demonstrated by RS-KII, an instantiation of KII with expressive constraint and preference languages described in Chapter 4. RS-KII is expressive,

utilizing the knowledge used by at least two disparate induction algorithms, AQ-11 [Michalski, 1978] and, for certain hypothesis spaces, the Candidate Elimination Algorithm (CEA) [Mitchell, 1982]. In conjunction with this knowledge, RS-KII can also utilize knowledge that these algorithms can not, such as a domain theory and noisy examples with *bounded inconsistency* [Hirsh, 1990]. It seems likely that RS-KII can utilize the knowledge of other induction algorithms as well, although this is left for future work. This raises the possibility of combining knowledge from several induction algorithms in order to form hybrid algorithms (as in adding a domain theory to AQ-11). RS-KII's apparent expressiveness also raises the possibility of improving the accuracy of the induction by utilizing additional knowledge.

RS-KII is expressive, but it can also be computationally complex. In the worst case it is exponential in the number of knowledge fragments it utilizes. However, when RS-KII utilizes only the knowledge of AQ-11, it has a computational complexity that is slightly higher than that of AQ-11, but still polynomial in the number of examples. When RS-KII uses only the knowledge used by CEA, RS-KII's worst-case complexity is the same as that of CEA, but there are problems where RS-KII's complexity is $O(n^2)$ when the complexity of CEA is $O(2^n)$.

Emulations of these algorithms by RS-KII, and analyses of RS-KII's complexity with respect to AQ-11 and CEA, are discussed in Chapter 5 (AQ-11) and Chapter 6 (CEA).

1.2 Organization of the Dissertation

The remainder of this dissertation is organized as follows. The KII framework and an illustration of KII solving an induction problem are discussed in Chapter 2. Chapter 3 lays out the space of possible set representations and identifies the most expressive representations for which induction is computable. RS-KII, an instantiation of KII based on the regular set representation, is described in Chapter 4. Emulations of AQ-11 and IVSM by RS-KII are described in Chapter 5 and Chapter 6, respectively. These chapters demonstrate that RS-KII can utilize the knowledge used by these algorithms, and that RS-KII can also utilize this knowledge while using knowledge these algorithms cannot. RS-KII's computational complexity when utilizing this knowledge is compared to the complexity of the original algorithms. Related work

is discussed in Chapter 7. Future work is discussed in Chapter 8, and conclusions appear in Chapter 9.

Chapter 2

KII

KII is a Knowledge Integration framework for Induction. It can integrate arbitrary knowledge into induction, and provides a foundation for understanding the effects of knowledge on the complexity and accuracy of induction. Knowledge comprises examples, biases, domain theories, meta-knowledge, implicit biases such as those found in the search control of induction algorithms, and any other information that would justify the selection of one hypothesis over another as the induced hypothesis.

Knowledge can be defined more precisely as the examples plus all of the biases. Mitchell [Mitchell, 1980] defines bias to be any factor that influences the selection of one hypothesis over the other as the target concept, other than strict consistency with the examples. The distinction between biases and examples is somewhat arbitrary, since they both influence selection of the target concept. More recent definitions consider consistency with the examples as a form of bias [Gordon and desJardins, 1995]. The latter definition of bias is used here. The terms *knowledge* and *bias* will be used interchangeably.

By definition, the biases are the only factors that determine which hypothesis is selected as the target concept. Induction is then a matter of determining which hypotheses are identified (preferred for selection) by the biases. The biases may prefer several hypotheses equally. In this case, the best that can be done is to select one of the hypotheses at random, since there are no other factors upon which to base a selection of one hypothesis over another. If there were such a factor, it would be one of the biases by definition.

The hypotheses identified by the biases are termed the *deductive closure* of the knowledge, since these are the hypotheses deductively implied by the biases. Inducing a hypothesis by selecting a concept from the deductive closure of the knowledge may seem more like *deduction* than *induction.* However, there is still plenty of room for inductive leaps within this formulation. Inductive leaps come from two main sources. One inductive leap occurs when there is more than one hypothesis in the deductive closure, so that one of them must be selected arbitrarily as the induced hypothesis. The second place where inductive leaps occur is in the knowledge itself. The knowledge can include unsupported biases, assumptions, and guesses—that is, inductive leaps. Even if there is only one hypothesis in the deductive closure of the knowledge, it may not be the target concept if the knowledge is not firmly grounded in fact.

In the knowledge integration paradigm, a hypothesis is induced from a collection of knowledge (biases) as follows. Each bias is represented explicitly as an expression in some representation language. An expression representing the biases collectively is constructed by composing the expressions for each individual bias. The composite expression is then used to generate the hypotheses identified by the biases. Ideally, the composite expression *is* the set of hypotheses identified by the biases, but realistically it may be necessary to do some processing in order to extract the identified hypotheses.

One example of an algorithm in the knowledge integration paradigm is incremental version space merging (IVSM) [Hirsh, 1990]. Biases are translated into *version spaces* [Mitchell, 1982] consisting of the hypotheses identified by the bias (i.e., consistent with the knowledge). The version spaces for each of the biases are intersected to yield a new version space consistent with all of the knowledge. This composite version space is the set of hypotheses identified by the biases. A hypothesis can be selected arbitrarily from the composite version space as the target concept.

The version space representation for biases in IVSM allows the identified hypotheses to be easily extracted, but it is somewhat limited in the knowledge it can represent. It can only utilize biases that reject hypotheses from consideration, but it cannot utilize knowledge that prefers one hypothesis over another. Also, version spaces are represented as *convex sets*, and the expressiveness of this representation further limits the kinds of knowledge that can be represented. KII can utilize both

constraint knowledge and preference knowledge, and allows a range of representation languages. These are discussed further in Section 2.1.

KII consists of three operations: *translation*, *integration*, and *enumeration*. The first step is translation. The form in which knowledge occurs in the learning scenario is usually not the form in which KII represents knowledge. *Translators* [Cohen, 1992] convert knowledge into the form used by KII. Once translated, the knowledge is then *integrated* into a composite representation from which the deductive closure can be computed. At any time, one or more hypotheses can be *enumerated* from the deductive closure of the integrated knowledge. A hypothesis is induced from a collection of knowledge by translating the individual knowledge fragments (biases) in the collection, integrating the fragments together, and enumerating one of the hypotheses identified by the biases.

Other information about the set of hypotheses identified by the biases is also relevant to induction, such as whether the set is empty, or whether it contains a single hypothesis. KII provides this information via *queries*. These are discussed further in Section 2.2.4.

The rest of this chapter is organized as follows. KII's knowledge representation is described in Section 2.1. The translation, integration, enumeration, and query operations are described in Section 2.2. An example of KII solving an induction task is given in Section 2.3.

2.1 Knowledge Representation

KII's knowledge representation is subject to a number of constraints. First, it must be *expressive*. Knowledge that can not be expressed can not be utilized, and the goal of KII it to utilize all of the available knowledge. Second, the representation must be *composable*. This facilitates integration by allowing a collection of knowledge fragments to be represented as a composition of individual fragments. Finally, the representation must be *operational*. That is, it must be possible both to enumerate hypotheses from the deductive closure of the integrated knowledge and to answer queries about the deductive closure.

KII satisfies these three criteria on the representation language by expressing knowledge in terms of constraints and preferences on the hypothesis space. The

constraints correspond to representational biases, and the preferences correspond to procedural biases. For instance, a positive example might be expressed as a constraint that hypotheses in the deductive closure must cover the example. A negative example can be expressed as a constraint requiring hypotheses in the deductive closure not to cover the example. An information gain metric, used by such algorithms as ID3[Quinlan, 1986] and FOCL[Pazzani and Kibler, 1992], would be expressed as a preference for hypotheses with a higher information gain. There is more than one way to express a given knowledge fragment in terms of constraints and preferences. These issues are discussed further in Section 2.2.1.

Each fragment of knowledge is translated into constraints and preferences. The constraints and preferences of a knowledge fragment are represented in KII by a tuple of three sets, $\langle H, C, P \rangle$, where H is the hypothesis space, C is the set of hypotheses in H that satisfy the constraints, and P is a set of hypothesis pairs, $\langle a, b \rangle$, such that a is less preferred than b ($a < b$). When the hypothesis space is obvious, H can be omitted from the tuple.

The biases encoded by tuple $\langle H, C, P \rangle$ identify the most preferred hypotheses among those satisfying the constraints. Hypotheses that do not satisfy the constraints can not be the target concept, so these are eliminated. Among the remaining hypotheses, P indicates that some of these hypotheses should be selected over others. The less preferred hypotheses are eliminated, leaving a set of hypotheses that satisfy the constraints and are preferred by the preferences. The knowledge can not discriminate further among these hypotheses. This is the set of hypotheses identified by the biases in $\langle H, C, P \rangle$. This set is also referred to as the *deductive closure* of the knowledge in $\langle H, C, P \rangle$.

Specifically, the deductive closure of $\langle H, C, P \rangle$ is the set $\{h \in C \mid \forall_{h' \in C} \langle h, h' \rangle \notin P\}$. This is equivalent to Equation 2.1, below. In this equation, $\overline{A}$ is the complement of set A with respect to universe H, and *first* projects a set of pairs onto its first elements. For example, $\mathit{first}(\{\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \ldots\})$ returns the set $\{x_1, x_2, \ldots\}$.

$$\overline{\mathit{first}((C \times C) \cap P)} \cap C \tag{2.1}$$

The deductive closure of $\langle H, C, P \rangle$ tuple can also be thought of as the solution set of a constrained optimization problem (COP), where the domain of the COP is

the hypothesis space (H), the constraints are specified by C, and the optimization criteria are specified by P. For this reason, an $\langle H, C, P \rangle$ tuple is also called a COP, and the deductive closure of $\langle H, C, P \rangle$ is also referred to as the *solution set.*

The $\langle H, C, P \rangle$ representation is expressive and composable, but since KII does not specify particular set representations for H, C, and P, it is not operational. It is expressive, since just about any knowledge relevant to induction can be expressed in terms of constraints and preferences. Integration is defined for every pair of $\langle H, C, P \rangle$ tuples, so the representation is also composable. Constraints are integrated by conjoining them into a single conjunctive constraint, and preferences are integrated by computing their union. Specifically, the integration of $\langle H, C_1, P_1 \rangle$ and $\langle H, C_2, P_2 \rangle$ is $\langle H, C_1 \cap C_2, P_1 \cup P_2 \rangle$.

In order for the $\langle H, C, P \rangle$ representation to be operational, there must be an algorithm for enumerating hypotheses from the deductive closure of $\langle H, C, P \rangle$. The deductive closure of $\langle H, C, P \rangle$ is always defined in terms of set operations over H, C, and P, but since KII does not specify any particular set representations, the set operations are not operational. The $\langle H, C, P \rangle$ representation is operationalized by providing set representations for H, C, and P. In theory, any knowledge can be represented in terms of constraints and preferences, but in practice the set representation determines the constraints and preferences that can be represented, and the complexity of integrating knowledge and extracting hypotheses from the deductive closure of the knowledge.

Specifying a set representation operationalizes KII, and produces an induction algorithm (or family of algorithms) at a particular level of complexity and expressiveness. This provides a useful handle for investigating the effects of knowledge on induction, by relating these effects to properties of set representations. Set representations and their properties are discussed further in Chapter 3. An instantiation of KII based on the regular-set representation is discussed in Chapter 4.

2.2 Operations

This section formally describes the operations provided by KII. The four families of operations are *translators* for expressing knowledge as $\langle H, C, P \rangle$ tuples, an *integrator* for integrating $\langle H, C, P \rangle$ tuples, an *enumerator* for enumerating hypotheses from the

deductive closure of a $\langle H, C, P \rangle$ pair, and *queries* about properties of the deductive closure.

An induction task consists of an unknown target concept and knowledge from which the target concept is to be induced. Knowledge provided by the induction task is expressed in some representation, but probably not the COP representation expected by KII. *Translators* convert knowledge from its task representation into COPs.

COPs for each knowledge fragment are integrated by KII's *integrate* operator into a composite COP representing all of the knowledge. A hypothesis is induced by selecting a hypothesis arbitrarily from the deductive closure of the integrated knowledge. This is achieved by the *enumerate* operator. The deductive closure of the integrated knowledge is exactly the solution set to the COP resulting from integrating all the translated knowledge. The *enumerate* operator returns a requested number of hypotheses from the solution set of this COP.

Also relevant to induction are certain properties of the solution set, such as whether the solution set is empty, whether the induced hypothesis is the only one in the solution set, and whether a given hypothesis is in the solution set. These properties, and others, are determined by solution-set *queries*.

The remainder of this section discusses the KII operations in more detail.

2.2.1 Translators

An induction task consists of an unknown target concept and some collection of knowledge from which the target concept is to be induced. In order for KII to use this knowledge, it must first be translated from whatever form it has in the task (its *naturalistic* representation [Rosenbloom *et al.*, 1993]) into the representation expected by KII—namely, a COP. This operation is performed by *translators* [Cohen, 1992].

A translator converts knowledge from some naturalistic representation into the constraints and preferences that make up a COP. The translators are highly task-dependent. They depend on the kind of knowledge being translated, the naturalistic representation of that knowledge, and the hypothesis space. The hypothesis space is a necessary input to the translator since the output of the translator consists of

constraints and preferences over the hypothesis space. It is necessary to know the domain being constrained in order to generate meaningful constraints, and likewise for preferences.

The dependence of the translator on the hypothesis space means that all translators for a given induction task must take the same hypothesis space as input, and output constraints and preferences over the same domain. Effectively, all of the knowledge fragments are dependent on the hypothesis space bias, which is also a knowledge fragment.

These restrictions mean that each induction task will usually need its own set of translators. However, this does not mean that reuse is impossible. If there is some knowledge used by several tasks, and the knowledge has the same naturalistic representation in all of the tasks, and the tasks use the same hypothesis space, then the same translator can be used for that knowledge in all of the tasks.

2.2.1.1 Examples of Translators

Specifications of translators for common knowledge sources are shown below. In the following, H is the hypothesis space, *pos* is a positive example, *neg* is a negative example, and T is a domain theory. A domain theory is a collection of horn-clause rules that explain why an instance is an example of the target concept.

- $PosExample(H, \text{pos}) \rightarrow \langle H, C, \{\}\rangle$ where
  $C = \{h \in H \mid h \text{ covers pos}\}$
  (e.g., FOCL [Pazzani and Kibler, 1992], AQ11 [Michalski, 1978], CN2 [Clark and Niblett, 1989], CEA [Mitchell, 1982])

  Every noise-free positive example is covered by the target concept. No hypothesis that fails to cover a noise-free positive example can be the target concept. A noise-free positive example is therefore translated as a constraint that is only satisfied by hypotheses that cover the example. This bias is known as strict consistency with the examples, and is used by many existing algorithms.

- *NegExample*$(H, \text{neg}) \rightarrow \langle H, C, \{\}\rangle$ where
  $C = \{h \in H \mid h \text{ does not cover neg}\}$
  (e.g., FOCL, AQ11, CN2, CEA)

The target concept does not cover noise-free negative examples. The translator for noise-free negative examples is similar to the translator for noise-free positive examples, except that the constraint is satisfied by hypotheses that do *not* cover the example.

- *NoisyExamples*$(H, \text{examples}) \rightarrow \langle H, \{\}, P\rangle$ where
  $P = \{\langle x, y\rangle \in H \mid x \text{ is consistent with fewer examples than } y\}$

In the real world, examples can contain errors, or noise. Demanding strict consistency with noisy examples could cause the target concept to be rejected, or there may be no hypotheses consistent with all of the examples.

One simple approach is to assume that the examples are mostly correct, and therefore hypotheses that are consistent with most of the examples are more preferred than those consistent with only a few examples. It is difficult to find a translation for individual examples that would yield this preference when their translations are integrated. Instead, all of the examples are translated collectively into a single set of preferences.

This translation is somewhat naive, but provides a simple illustration of how preferences are used, and how noisy examples can be handled. More sophisticated translators for noisy examples are discussed in Section 6.2.3.2.

- *CorrectDomainTheory*$(H, T) \rightarrow \langle H, C, \{\}\rangle$ where
  $C = \{h \in H \mid h \text{ covers exactly those instances explained by } T\}$
  (e.g., EBL[DeJong and Mooney, 1986, Mitchell *et al.*, 1986])

A complete and correct domain theory exactly describes the target concept. The theory is translated into a constraint that is satisfied only by the target concept, which is described by the theory. There is very little induction here,

since the target concept has been provided. This kind of theory is more often used in speed-up learning, where the objective is not to learn a new classifier, but to make a known classifier more efficient (e.g., [DeJong and Mooney, 1986]).

- *OverSpecialDomainTheory*$(H, T) \rightarrow \langle H, C, \{\}\rangle$ where
  $C = \{h \in H \mid h$ is a generalization of domain theory $T\}$
  (e.g., IOE [Flann and Dietterich, 1989])

  An overspecial domain theory can only describe why some of the instances covered by the target concept are examples of the target. The theory is a specialization of the target concept. The theory is translated as a constraint satisfied by all generalizations of the concept described by the theory. See Section 5.2.3 and Section 6.2.3.3 for more detailed domain theory translators.

A given fragment of knowledge can have several translations, depending on how the knowledge is to be used. One can think of the intended use as knowledge that is provided implicitly to the translator. For example, there are at least two translations for domain theories, depending on what assumptions are made about the correctness of the theory. Likewise, examples also have different translations, depending on whether the examples are assumed to be noisy or error free.

2.2.1.2 Independent and Dependent Knowledge Translation

A collection of knowledge fragments that can be translated independently of each other are said to be *independent.* The only inputs to a translator for an independent fragment is the fragment itself. There are no truly independent knowledge fragments in KII, since each fragment is translated into constraints and preferences over the hypothesis space, so both the fragment and the hypothesis space are necessary inputs to the translator. However, if each fragment is paired with the hypothesis space to form a new unit, then these units can all be translated independently.

Knowledge fragments that can only be translated in conjunction with each other are said to be *dependent.* This can occur either because the knowledge fragments are inherently dependent, or because the set representations for C and P cannot express the constraints and preferences for the individual fragments, but can express the

constraints and preferences imposed by the fragments collectively. A translator for a collection of dependent knowledge fragments takes the whole collection as input and produces a single $\langle H, C, P \rangle$ tuple. The inputs are termed a *dependent unit.*

Ideally, the dependencies among the knowledge are minimal, so that each dependent unit consists of only a few knowledge fragments. Every knowledge fragment is dependent on the hypothesis space, so there is at least one unit per knowledge fragment, each of which contains the fragment and the hypothesis space. Additional dependencies can decrease the number of units and increase the number of fragments in each, until at maximum dependence there is a single unit containing all of the knowledge. Independence among the knowledge leads to greater flexibility in deciding what knowledge to utilize, and ensures that knowledge integration occurs within KII's integration operator and not within the translators.

Independence leads to greater flexibility in using knowledge. When a collection of knowledge fragments is dependent, it is not possible to translate only some of the fragments in the collection (unless they participate in other dependent units as well). It is often an all or none proposition. This constrains the choices of what knowledge to utilize. The greater the independence among the knowledge, the smaller the dependent units tend to be, and the fewer constraints there are. At maximum independence, each unit consists of a knowledge fragment and the hypothesis space, so individual fragments can be utilized or omitted as desired.

When the knowledge fragments are independent, they can be translated and integrated incrementally. However, this does not mean that induction in KII is necessarily incremental. In order to induce a hypothesis from a body of knowledge in KII, the knowledge is translated and integrated, and a hypothesis is enumerated from the deductive closure of the knowledge. The translation and integration steps may be incremental when the knowledge is independent, but the enumeration step may not be. The enumeration algorithm may have to start over when new knowledge is integrated, instead of picking up where it left off. This subject is discussed further in Section 2.2.5.

Independence among the knowledge ensures that the integration work occurs within KII, and not within the translators. Independent knowledge fragments are translated independently, and the $\langle H, C, P \rangle$ tuples for the individual fragments are composed by the *Integrate* operator into a single $\langle H, C, P \rangle$ tuple. The integration

work occurs within KII. Collections of dependent knowledge fragments are translated directly into a single $\langle H, C, P \rangle$ tuple. The integration of the knowledge occurs within the translator, and not within KII.

When most of the induction process occurs outside of KII, the power of KII cannot be brought to bear in facilitating the integration of new knowledge. In the extreme case there is a single translator for all of the knowledge, and all of the integration takes place within that translator. The translator is effectively a special purpose integration algorithm replacing KII's integration operator. In order to utilize additional knowledge, a new translator has to be constructed. KII's knowledge integration capabilities are circumvented.

Independence of the knowledge is clearly desirable, but not always easy to obtain. Some knowledge is inherently dependent, and must always be translated as a unit. Other knowledge is dependent in one representation, but independent in another. Independence tends to increase with the expressiveness of the C and P representations, since it is more likely that translations of individual knowledge fragments can be expressed. However, complexity also increases with expressiveness, so it may be necessary to trade some independence among the knowledge for improved complexity.

2.2.2 Integration

Two COPs can be integrated, yielding a new COP whose solution set is the deductive closure of the knowledge represented by the two COPs being integrated. The integration operation is defined as follows. Both COPs must have the same hypothesis space, since it makes no sense to integrate COPs whose constraints and preferences are over entirely different universes of hypotheses.

$$\text{Integrate}(\langle H, C_1, P_1 \rangle, \langle H, C_2, P_2 \rangle) = \langle H, C_1 \cap C_2, P_1 \cup P_2 \rangle \tag{2.2}$$

The C and P sets of each COP represent reject knowledge and preference knowledge respectively. If a hypothesis is rejected by either COP, then it cannot be the target concept. Thus the constraints are conjoined, which corresponds to intersecting the C sets. Preference knowledge, by contrast, is unioned. If one COP has no

preference between hypotheses a and b, but the other COP prefers a to b, then a should be preferred over b. This corresponds to computing the union of the preferences in the two P sets.

2.2.3 Enumeration

The enumeration operator returns a requested number of hypotheses from the deductive closure of the knowledge represented by an $\langle H, C, P \rangle$ tuple. The solution set to $\langle H, C, P \rangle$ consists of the undominated elements of C, and is defined formally in Equation 2.3. In this definition, the function $\mathit{first}(\{\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \ldots\})$ is a projection returning the set of tuple first-elements, namely $\{x_1, x_2, \ldots\}$.

$$\begin{aligned}
\mathit{Solutions}(\langle H, C, P \rangle) &= \{x \in C \mid \forall_{y \in C}\ \langle x, y \rangle \notin P\} \\
&= \overline{\{x \in H \mid (x \in C \text{ and } \exists_{y \in C} \langle x, y \rangle \in P) \text{ or } x \notin C\}} \\
&= \overline{\{x \in H \mid x \in C \text{ and } \exists_{y \in C} \langle x, y \rangle \in P\} \cup \overline{C}} \\
&= \overline{\mathit{first}(\{\langle x, y \rangle \in C \times C \mid \langle x, y \rangle \in P\})} \cap C \\
&= \overline{\mathit{first}((C \times C) \cap P)} \cap C
\end{aligned} \tag{2.3}$$

Enumerate takes three arguments, $\langle H, C, P \rangle$, A, and n, where $\langle H, C, P \rangle$ is a COP, A is an arbitrary set from the same universe as C and P, and n is a non-negative integer. $\mathit{Enumerate}(\langle H, C, P \rangle, A, n)$ returns an extensional list of up to n hypotheses that are in both A and the deductive closure of $\langle H, C, P \rangle$. If there are fewer than n hypotheses in the intersection of A and the deductive closure of $\langle H, C, P \rangle$, then $\mathit{Enumerate}(\langle H, C, P \rangle, A, n)$ returns all of the hypotheses in the intersection. The argument A is necessary to implement some of the solution-set queries, as shown in Section 2.2.4. It can be effectively eliminated by setting A to C, or any superset of C, since the solution set is always a subset of C.

$$\begin{aligned}
&\mathit{Enumerate}(\langle H, C, P \rangle, A, n) \rightarrow \{h_1, h_2, \ldots h_m\} \\
&\qquad \text{where } m = \min(n, |\mathit{SolnSet}(\langle H, C, P \rangle) \cap A|)
\end{aligned}$$

The tuple $\langle H, C, P \rangle$ is the result of integrating the tuples generated by the translators. C is the intersection of several constraint sets, and P is a union of preference sets. Equation 2.3 assumes that P is transitively closed and a partial order. However, neither of these conditions is guaranteed by the integration operator. The preference sets produced by the translators must be transitively closed partial orders, but these conditions are not preserved by union. If P_1 has the preference $a < b$ and P_2 has the preference $b < a$, then $P_1 \cup P_2$ contains a cycle ($a < b$ and $b < a$). A partial ordering is antisymmetric and irreflexive by definition, so the union is not a partial ordering. Transitive closure is not preserved by union either. If $a < b$ and $c < d$ are in P_1, and $b < c$ is in P_2, $P_1 \cup P_2$ will not contain $a < d$, even though this is in the transitive closure.

The lack of transitive closure is dealt with by transitively closing P prior to calling *Enumerate*. The implementation of the closure operation depends on the set representation for P. Cycles in P are more problematic. Currently, KII simply assumes that the preferences are consistent, so that P has no cycles. If P does contain cycles, then hypotheses participating in the cycles are all dominated, and not part of the solution set. In other words, hypotheses about which there is conflicting knowledge are rejected, but the remaining hypotheses are considered normally for membership in the solution set.

Contradictions can also occur among the constraint knowledge. If the constraints are mutually exclusive, then C is empty, and so is the solution set. KII tolerates such conflicts, but does not have any sophisticated strategy for dealing with them. The solution space simply collapses, and it is up to the user to try a different set of knowledge.

A more sophisticated approach would be to detect mutually exclusive constraints and cycles in the preferences, either as part of integration or prior to enumeration, and resolve the conflicts according to a conflict resolution strategy. This is an area for future research.

2.2.4 Solution-set Queries

Solution-set queries return information about the deductive closure of a $\langle H, C, P \rangle$ tuple. There are four solution-set queries, based on those Hirsh proposed for version

spaces [Hirsh, 1992]. As discussed earlier, version spaces represent deductive closures of knowledge, so these queries should also be appropriate for KII.

There are four solution-set queries, defined as follows. In these definitions, $\langle H, C, P \rangle$ is a COP, and h is some hypothesis in H.

- $Member(h, \langle H, C, P \rangle) \rightarrow$ Boolean
  Returns true if h is a member of the solution set to $\langle H, C, P \rangle$. Returns false otherwise. $Member(h, \langle H, C, P \rangle)$ is equivalent to $h \in C$ and $(\{h\} \times C) \cap P = \emptyset$

- $Empty(\langle H, C, P \rangle) \rightarrow$ Boolean
  Returns true if the solution set to $\langle H, C, P \rangle$ is empty. Returns false otherwise.

- $Unique(\langle H, C, P \rangle) \rightarrow$ Boolean
  Returns true if the solution set to $\langle H, C, P \rangle$ contains exactly one member. Returns false otherwise.

- $Subset(\langle H, C, P \rangle, A) \rightarrow$ Boolean
  A is a set of hypotheses. Returns true if $\langle H, C, P \rangle \subseteq A$, and returns false otherwise. $Subset(\langle H, C, P \rangle, A) \equiv SolutionSet(\langle H, C, P \rangle) \cap \overline{A} = \emptyset$.

The queries take $\langle H, C, P \rangle$ as an argument instead of the solution set itself. This is because it may be possible to answer the query without computing the entire solution set, which could yield significant computational savings. Taking $\langle H, C, P \rangle$ as an argument also allows the queries to be implemented in terms of *Enumerate*, which facilitates both implementation of the queries and analysis of their computational complexity.

The computational complexity of the queries is essentially the complexity of enumerating one or two hypotheses. Whether enumerating a few hypotheses is significantly cheaper than computing the entire solution set depends on the set representation and on the COP itself. The computational complexity of enumeration as a function of set representations is discussed in Chapter 3, and the question of whether enumeration is cheaper than computing the whole solution set is discussed in Chapter 4.

The queries are defined in terms of *Enumerate* as follows.

- $Member(h, \langle H, C, P \rangle) \Leftrightarrow Enumerate(\langle H, C, P \rangle, \{h\}, 1) \neq \emptyset$

- $Empty(\langle H, C, P\rangle) \Leftrightarrow Enumerate(\langle H, C, P\rangle, H, 1) = \emptyset$
- $Unique(\langle H, C, P\rangle) \Leftrightarrow |Enumerate(\langle H, C, P\rangle, H, 2)| = 1$
- $Subset(\langle H, C, P\rangle, A) \Leftrightarrow Enumerate(\langle H, C, P\rangle, \overline{A}, 1) = \emptyset$

It is conjectured that these four queries plus the enumeration operator are sufficient for the vast majority of induction tasks. Most existing induction algorithms involve only the enumeration operator and perhaps an *Empty* or *Unique* query. The candidate elimination algorithm [Mitchell, 1982] and incremental version space merging (IVSM) [Hirsh, 1990] use all four queries, but do not use the enumeration operator.

2.2.5 Incremental versus Batch Processing

In an induction algorithm, a hypothesis is induced from all of the knowledge integrated so far. Inducing the hypothesis involves some amount of work. When new knowledge is integrated, the old induced hypothesis may no longer be valid in light of the new knowledge. A new hypothesis should be induced. In an incremental algorithm, the work done in inducing a hypothesis from the old knowledge can be applied towards inducing the new hypothesis.

For example, in the candidate elimination algorithm [Mitchell, 1982], the version space represents all of the hypotheses consistent with the examples. Any of these hypotheses can be selected as the induced hypothesis. The work done in computing the version space is conserved in computing a new version space consistent with the old examples plus a new example. The new version space is computed by removing inconsistent hypotheses from the old version space. The new version space does not need to be computed from scratch.

In a batch algorithm, the old work cannot be applied to inducing a new hypothesis. The algorithm must start over again from scratch. For example, in AQ-11 [Michalski, 1978], the hypothesis is found by a beam search guided by an information theoretic measure, which is derived from the examples. If a new example is added, the metric changes. The search must start from the beginning with the new metric.

KII is neither inherently batch nor inherently incremental. Rather, individual instantiations of KII can be incremental or batch, depending on the set representation and the translators. These factors can be manipulated and analyzed, which makes KII a potentially useful tool for experimenting with these factors, and may suggest ways for making algorithms more incremental.

A hypothesis is induced from a collection of knowledge by translating the fragments into $\langle H, C, P \rangle$ tuples, integrating the tuples into a single $\langle H, C, P \rangle$ tuple, and then enumerating a hypothesis from the solution set of this composite tuple. When new knowledge is added, it is translated into $\langle H, C', P' \rangle$, integrated with the old $\langle H, C, P \rangle$ tuple to yield $\langle H, C \cap C', P \cup P' \rangle$, and a new hypothesis is enumerated from $\langle H, C \cap C', P \cup P' \rangle$.

There are two places where work done in processing the old knowledge can be saved or lost when new knowledge becomes available. The work done in enumerating a hypothesis from the solution set of the $\langle H, C, P \rangle$ tuple can be applied to enumerating a hypothesis from the solution set of $\langle H, C \cap C', P \cup P' \rangle$. Work done in integrating the old knowledge can either be saved or lost in integrating the new knowledge fragment, $\langle H, C', P' \rangle$.

Whether any of the work involved in enumerating a hypothesis from the solution set of $\langle H, C, P \rangle$ can be applied to enumerating a hypothesis from the solution set of $\langle H, C \cap C', P \cup P' \rangle$ depends largely on the set representations for C and P, and the extent of the changes introduced by C' and P'. As an extreme example, if P and P' are both empty, then the solution set to $\langle H, C, P \rangle$ is just C, and the solution set to $\langle H, C \cap C', P \cup P' \rangle$ is just $C \cap C'$. A hypothesis can be enumerated from $C \cap C'$ by continuing the enumeration of C until a hypothesis is found that is also in C'. There is no need to start the enumeration of C over from the beginning, since any hypothesis not in C is not in $C \cap C'$ either. All of the previous work is saved.

The second place where KII can either save or lose previous work is in integrating knowledge. Work is lost only if the translation of the existing knowledge depends on the new knowledge. This usually occurs in translators where one of the inputs takes all knowledge of a given type (e.g., all the examples). When a new knowledge fragment of this type is added, the old fragments must be retranslated to take the new fragment into account. However, the old translation has already been integrated. If this translation is not retracted before the new translation is integrated, then

the old and new translations may conflict. Since KII has no facility for retracting translations, the only way to retract the old translation is to re-integrate all of the knowledge from scratch, omitting the offending translation. The previous integration work is lost, as is any enumeration work.

Not all translators of this type necessarily require the old translation to be retracted when new knowledge arrives. Consider a translator, $tran(E)$ that takes as input the set of all available examples. Let E be a collection of examples, and e a new example. Furthermore, let $tran(E)$ be $\langle H, C, P\rangle$ and $tran(E \cup \{e\})$ be $\langle H, C^*, P^*\rangle$. If $\langle H, C, P\rangle$ has already been integrated, then it usually must be retracted before integrating $tuple(H, C^*, P^*)$. However, consider what happens if $\langle H, C^*, P^*\rangle$ can be expressed as $\langle H, C \cap C_e, P \cup P_e\rangle$, where $\langle H, C_e, P_e\rangle$ is derived from e (and possibly E). The translator can simply output $\langle H, C_e, P_e\rangle$. This will be integrated with the previously integrated $\langle H, C, P\rangle$ producing $\langle H, C^*, P^*\rangle$, which is the correct translation of $E \cup \{e\}$. There is no need to retract $\langle H, C, P\rangle$ first.

2.3 KII Solves an Induction Task

An example of how KII can solve a simple induction task is given below. Sets have been represented extensionally in this example for illustrative purposes. This is not the only possible set representation, and is generally a very poor one. The space of possible set representations is investigated more fully in Chapter 3.

2.3.1 Hypothesis Space

The target concept is a member of a hypothesis space in which hypotheses are described by conjunctive feature vectors. There are three features `size`, `color`, and `shape`. The values for these features are `size` $\in$ {`small`, `large`, `any-size`}, `color` $\in$ {`black`, `white`, `any-color`}, and `shape` $\in$ {`circle`, `rectangle`, `any-shape`}. A hypothesis is some assignment of values to features for a total of 27 distinct hypotheses. Hypotheses are described as 3-tuples from `size`$\times$`size`$\times$`shape`. For shorthand identification, a value is specified by the first character of its name, except for the "any-x" values which are represented by a "?". So the hypothesis $\langle$`white`, `any-size`, `circle`$\rangle$ would be written as $\langle w, ?, c\rangle$ or just $w?c$.

2.3.2 Instance Space

Instances are the "ground" hypotheses in the hypothesis space. An instance is a tuple $\langle color, size, shape \rangle$ where $color \in \{\texttt{black}, \texttt{white}\}$, $size \in \{\texttt{small}, \texttt{large}\}$, and $shape \in \{\texttt{circle}, \texttt{rectangle}\}$.

A hypothesis *covers* an instance if the instance is a member of the hypothesis. Recall that a hypothesis is a subset of the instance space, as described in some language. In this learning scenario, an instance is a member of (covered by) all hypotheses that are more general than the instance. Specifically, an instance, $\langle z, c, s \rangle$ is covered by every hypothesis in the set $\{z, \texttt{any-size}\} \times \{c, \texttt{any-color}\} \times \{s, \texttt{any-shape}\}$.

2.3.3 Available Knowledge

The available knowledge consists of three examples (classified instances), and an assumption that accuracy increases with generality. There are three examples, two positive and one negative, as shown in Table 2.1. The target concept is $\langle s, ?, ? \rangle$. That is, `size = small`, and color and shape are irrelevant.

identifier	class	example
e-1	+	⟨small,white,circle⟩
e-2	+	⟨small,black,circle⟩
e-3	−	⟨large,white,rectangle⟩

Table 2.1: Examples.

2.3.4 Translators

The first step is to translate the knowledge into constraints and preferences. Three translators are constructed, one for each type of knowledge: the positive examples, negative examples, and the generality preference. These translators are shown in Figure 2.1. As with all translators in KII, each of these translators takes the hypothesis space, H, as one of its inputs. Since the hypothesis space is understood, $\langle H, C, P \rangle$ tuples will generally be referred to as just $\langle C, P \rangle$ tuples for the remainder of this illustration.

- *TranPosExample*$(H, \langle z, c, s \rangle) \rightarrow \langle C, \{\} \rangle$ where

$$\begin{aligned} C &= \{x \in H \mid x \text{ covers } \langle z, c, s \rangle\} \\ &= \{z, \texttt{any-size}\} \times \{c, \texttt{any-color}\} \times \{s, \texttt{any-shape}\} \end{aligned}$$

- *TranNegExample*$(H, \langle z, c, s \rangle) \rightarrow \langle C, \{\} \rangle$ where

$$\begin{aligned} C &= \{x \in H \mid x \text{ does not cover } \langle z, c, s \rangle\} \\ &= \text{complement of } \{z, \texttt{any-size}\} \times \{c, \texttt{any-color}\} \times \{s, \texttt{any-shape}\} \end{aligned}$$

- *TranPreferGeneral*$(H) \rightarrow \langle \{\}, P \rangle$ where

$$\begin{aligned} P &= \{\langle x, y \rangle \in H \times H \mid x \text{ is more specific than } y\} \\ &= \{\langle sbr, ?br \rangle, \langle sbr, s?r \rangle, \langle sbr, ??r \rangle, \langle swr, ?wr \rangle, \ldots\} \end{aligned}$$

Figure 2.1: Translators.

The examples are translated in this scenario under the assumption that they are correct. Under this assumption, hypotheses that do not cover all of the positive examples, or that cover any of the negative examples, can be eliminated from consideration. Beyond these constraints, examples do not express preferences for certain hypotheses over others. Thus, positive examples are translated into $\langle C, P \rangle$ pairs in which P is empty and C is satisfied only by hypotheses that cover the example. Negative examples are translated similarly, except that C is satisfied by hypotheses that do not cover the example. The translated examples are shown in Table 2.2. The hypothesis space is omitted for brevity.

The assumption that general hypotheses are more accurate is represented as a a preference. Specifically, the assumption is translated into a $\langle C, P \rangle$ pair where C is empty (it rejects nothing), and $P = \{\langle x, y \rangle \in H \times H \mid x \text{ is more specific than } y\}$, where H is the hypothesis space. Hypothesis x is more specific than hypothesis y if x is equivalent to y except that some of the values in y have been replaced by "any" values. For example, $\langle s, w, r \rangle$ is more specific than $\langle ?, w, r \rangle$, but there is no ordering between $\langle ?, w, c \rangle$ and $\langle s, w, r \rangle$. There are too many preferences to list explicitly,

Example	Class	C
$e_1 : swc$	+	swc, $sw?$, $s?c$, $?wc$, $s??$, $?w?$, $??c$, $???$
$e_2 : sbc$	+	sbc, $sb?$, $s?c$, $?bc$, $?b?$, $s??$, $??c$, $???$
$e_3 : lwr$	−	swc, swr, lwr, sbc, sbr, lbc, lbr, $lb?$, $sb?$, $sw?$, $?wc$, $?bc$, $?br$, $s?c$, $s?r$, $l?c$, $?b?$, $s??$, $??c$

Table 2.2: Translated Examples.

but the first few are shown in the definition for the *TranPreferGeneral* translator in Figure 2.1.

2.3.5 Integration and Enumeration

The COPs are integrated into a single COP, $\langle C, P\rangle$. C is the intersection of the C sets for each knowledge fragment, and P is the union of preferences for each knowledge fragment. This is shown in the following equation. The elements of each $\langle C, P\rangle$ pair are subscripted with the knowledge fragment from which the pair was translated: $e1$, $e2$, and $e3$ for the examples, and mg for the "more general" preference. In the following, $\langle C_1, P_1\rangle \oplus \langle C_2, P_2\rangle$ is shorthand for $Integrate(\langle C_1, P_1\rangle, \langle C_2, P_2\rangle)$.

$$\begin{aligned}\langle C, P\rangle &= \langle C_{e1}, \{\}\rangle \oplus \langle C_{e2}, \{\}\rangle \oplus \langle C_{e3}, \{\}\rangle \oplus \langle H, P_{mg}\rangle \\ &= \langle C_{e1} \cap C_{e2} \cap C_{e3} \cap H, \{\} \cup \{\} \cup \{\} \cup P_{mg}\rangle \\ &= \langle \{s??, ??c, s?c\}, \{\langle sbr, ?br\rangle, \langle sbr, s?r\rangle, \ldots\}\rangle\end{aligned}$$

A hypothesis is induced by making a call to $Enumerate(\langle C, P\rangle, H, 1)$, which returns a hypothesis selected arbitrarily from the deductive closure of $\langle C, P\rangle$. The deductive closure consists of the undominated elements of C with respect to the dominance relation P. One way to compute this set is to partially order C according to P and find the elements at the top of the order.

C contains three elements, $\langle s, ?, ?\rangle$, $\langle ?, ?, c\rangle$ and $\langle s, ?, c\rangle$. P prefers both $\langle s, ?, ?\rangle$ and $\langle ?, ?, c\rangle$ to $\langle s, ?, c\rangle$, but there is no preference ordering between $\langle s, ?, ?\rangle$ and $\langle ?, ?, c\rangle$. The deductive closure contains the undominated elements of C, namely $\langle s, ?, ?\rangle$ and $\langle ?, ?, c\rangle$. Figure 2.2 illustrates this computation. The arrows indicate

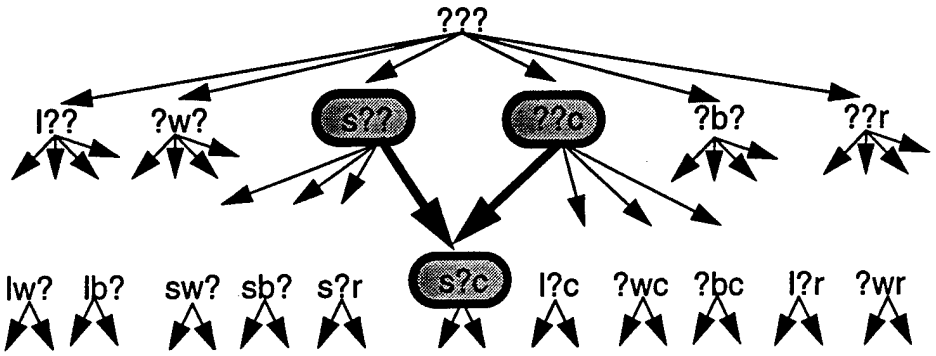

Figure 2.2: Computing the Deductive Closure of $\langle C, P \rangle$.

the partial ordering imposed by P, and the highlighted hypotheses are the elements of C.

Enumerate$(\langle C, P \rangle, H, 1)$ returns one hypothesis selected arbitrarily from the deductive closure of $\langle C, P \rangle$ as the induced hypothesis. If it selects $\langle s, ?, ? \rangle$, then it has correctly induced the target concept. However, it may just as easily select $\langle ?, ?, c \rangle$, the other element of the deductive closure. The selection of a hypothesis is an *inductive leap*, and not guaranteed to be correct. The amount of ambiguity in the leap can be reduced by utilizing more examples or other knowledge, thus reducing the number of hypotheses in the deductive closure.

Additional information about the deductive closure of $\langle C, P \rangle$ that is relevant to induction can be obtained through the solution-set queries. For example, we may wish to know whether or not the induced hypothesis was the only one in the deductive closure. This and other queries are shown below.

- *Unique*$(\{\langle s, ?, ? \rangle, \langle ?, ?, c \rangle\}) =$ `false`
- *Empty*$(\{\langle s, ?, ? \rangle, \langle ?, ?, c \rangle\}) =$ `false`
- *Member*$(\{\langle s, ?, ? \rangle, \langle ?, ?, c \rangle\}, \langle ?, ?, c \rangle) =$ `true`

Chapter 3

Set Representations

The KII framework discussed in Chapter 2 is not operational. Knowledge in KII is represented as $\langle C, P \rangle$ pairs, where C is a set of hypotheses and P is a set of hypothesis pairs. The integration, enumeration, and query operators are all defined in terms of set operations on $\langle C, P \rangle$ pairs. KII does not specify any particular set representation for C and P—in fact, these sets may be arbitrarily expressive. This expressiveness allows KII to utilize any knowledge that can be expressed in terms of constraints and preferences. However, arbitrarily expressive sets are not operational, and so neither is KII.

In order to operationalize KII, operational set representations must be provided for C and P. These representations determine which C and P sets can be expressed, and therefore the kinds of knowledge that KII can utilize. The more expressive the representation, the more knowledge that KII can use. However, set operations in more expressive representations tend to have higher computational complexities. Since KII's integration, enumeration, and query operators are defined in terms of set operations, their computational complexity is determined by the set representation.

KII can be operationalized with many different set representations. The expressiveness of the representation, and the computational complexity of its set operations, determine the knowledge that the operationalization can utilize, and the cost of its integration, enumeration, and query operations. If expressiveness is at a premium, one could select a very expressive set representation in exchange for increased complexity. If speed is of the essence, an inexpressive yet low complexity representation may be more appropriate. There may also be a representation that blends

the best of both worlds: expressive enough to represent most knowledge, yet with a reasonable computational cost.

In order to understand the space of operationalizations, it is necessary to understand the space of possible set representations. Section 3.1 casts the space of set representations in terms of the well understood space of grammars.

The set representations for C and P should also be closed under integration, although this is not strictly necessary. This condition is discussed in Section 3.2, and languages that are closed under integration are identified.

The complexity of induction also restricts the choice of set representations for C and P. Operationalizations of KII with expressive representations tend to have high induction costs, and for some representations, induction may even be undecidable. Clearly, such representations cannot be used to operationalize KII. The most expressive representations for which induction is decidable within the KII framework are identified in Section 3.3.

3.1 Grammars as Set Representations

Every computable set is the language of some grammar. Similarly, every computable set representation is equivalent to some family of grammars. These families include, but are not limited to, the families of the Chomsky hierarchy [Chomsky, 1959]—regular, context free, context sensitive, and recursively enumerable. Table 3.1 lists the language families in the Chomsky hierarchy in addition to other important language families. The list is in order of decreasing expressiveness, with the most expressive languages at the top and the least expressive at the bottom. The complexity of set operations generally increases with the expressiveness of the language. Each family in the list properly contains all of the families below it.

Every computable set representation either corresponds to one of the families in the list of Figure 3.1, or is subsumed by one of them. Mapping set representations onto families of grammars clarifies the space of possible representations, and makes it possible to use results from automata and formal language theory to analyze relevant properties of a set representation, such as expressiveness, and the decidability and computational complexity of set operations.

Recursively Enumerable (r.e.)
Recursive
Context Sensitive (CSL)
Context Free (CFL)
Deterministic Context Free (DCFL)
Regular
Finite

Table 3.1: Language Families.

3.2 Closure of C and P under Integration

The integration operator is defined in terms of intersection and union, as shown below.

$$\text{Integrate}(\langle C_1, P_1\rangle, \langle C_2, P_2\rangle) = \langle C_1 \cap C_2, P_1 \cup P_2\rangle \tag{3.1}$$

The representation for C must be closed under intersection, and the representation for P must be closed under union. This ensures that the $\langle C, P\rangle$ pair resulting from integration can be expressed in these representations. Table 3.2 indicates which of the languages in Table 3.1 are closed under intersection and union. Some language are not closed under intersection, but are closed under intersection with regular languages. Intersection with a regular language is often denoted as $\cap R$.

Operations	Language					
	Regular	DCFL	CFL	CSL	recursive	r.e.
$\cap$	√			√	√	√
$\cap R$	√	√	√	√	√	√
$\cup$	√		√	√	√	√

Table 3.2: Closure under Union and Intersection.

3.3 Computability of Induction

In order to induce a hypothesis, or to answer the solution set queries, it is necessary to enumerate one or more hypotheses from the deductive closure, or *solution set*,

of $\langle C, P \rangle$. This is only possible if the solution set is at most recursively enumerable. Otherwise, the solution set cannot be enumerated by any Turing machine, and therefore the enumeration and query operations are uncomputable.

Whether or not the solution set is recursively enumerable depends on the set representations for C and P. Recall that the solution set is computed from C and P according to the following equation:

$$\overline{first((C \times C) \cap P)} \cap C \tag{3.2}$$

A set representation is a class of grammars, and a specific set corresponds to a grammar in the class. Closing the set representations for C and P under Equation 3.2 must yield at most the recursively enumerable languages. Otherwise, the representations can express C and P sets for which the solution set is not recursively enumerable.

It is possible to determine the most expressive set representations for C and P that will yield a recursively enumerable solution set for every C and P set expressible in those representations. The closure properties of the set operations in Equation 3.2 can be used to express the solution set representation as a function of the representations for C and P. Inverting this function yields the most expressive representations for which the solution set is recursively enumerable.

3.3.1 Closure Properties of Set Operations

The closure properties of intersection and complement are well known for most language classes, although it is an open problem whether the context sensitive languages are closed under complementation [Hopcroft and Ullman, 1979]. These properties are summarized in Table 3.3. Cartesian product and projection (*first*) present more of a problem. A Cartesian product can be represented a number of different ways, each with different closure properties and expressive power. The representations differ in the way that a pair, $\langle x, y \rangle$, in the Cartesian product is mapped onto a string in the grammar that represents the product. Projection is the inverse of Cartesian product, and its definition depends on the mapping used to represent Cartesian products.

Operations	Language					
	Regular	DCFL	CFL	CSL	recursive	r.e.
$\cap$	$\surd$			$\surd$	$\surd$	$\surd$
complement	$\surd$	$\surd$		?	$\surd$	

Table 3.3: Closure Properties of Languages under Union and Intersection.

The most straightforward mapping is concatenation. That is, a tuple $\langle x, y \rangle$ in the Cartesian product is represented as the string xy in the grammar for the product. Under this mapping, the set $A \times B$ is represented as $A \cdot B$. Other mappings interleave the symbols in xy in various ways in order to represent the pair $\langle x, y \rangle$. For example, the symbols in x and y could be strictly alternated, or *shuffled*, so that $\langle x, y \rangle$ is represented as $x_1 y_1 x_2 y_2 \ldots x_n y_n$ where $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_n$. An interleaving maintains the order of the symbols in each of the component strings, so that if x_i comes before x_{i+1} in x, then x_i comes before x_{i+1} in the interleaving as well. However, there may be an arbitrary number of symbols from y between x_i and x_{i+1}.

For a given Cartesian product, $A \times B$, not all mappings of $A \times B$ can be represented in the same language class. For example, context free grammars are closed under concatenation, but not under shuffling [Hopcroft and Ullman, 1979]. If A and B are context free grammars, then under the concatenation mapping, $A \times B$ is represented as AB, which is also a context free grammar. Under the shuffle mapping, $A \times B$ is represented by the shuffle of A and B, which could be a context sensitive grammar. Whether a language class is closed under Cartesian product depends on the mapping. The expressive power of various mappings are discussed in more detail in Section 4.1.2.4.

Although the closure properties of languages under Cartesian product depends on the mapping, the closure properties of languages under projection are fixed. Let X be some subset of $A \times B$. Regardless of the mapping, X will be represented as some interleaving of the strings in A and B. Let Σ_A be the alphabet for A and let Σ_B be the alphabet for B. The string in X corresponding to $\langle x, y \rangle$ is some interleaving of the symbols in x and y—that is, $\langle x, y \rangle$ maps onto a string in $(\Sigma_A \cup \Sigma_B)^*$. Call this string w. Projecting w onto its first element extracts x from the interleaved string.

If Σ_A and Σ_B are disjoint, this can be done by deleting from the string symbols that belong to Σ_B. Mappings preserve the order of the symbols in x and y, so the remaining symbols in the strings are the symbols of x in the correct order.

If the alphabets of A and B are not disjoint, they can be made disjoint by applying appropriate *homomorphisms* [Hopcroft and Ullman, 1979]. Applying a homomorphism, h, to a language, L, replaces each symbol in the alphabet of L with a string from some other alphabet (each symbol can be replaced by a different string). The strings in $h(L)$ are the same as in L, except that the symbols in each strings have been substituted according to h. For example, if h replaces a by 1 and replaces b by 2, then $h(\{ab, bb\})$ is $\{12, 22\}$.

Let Σ_A be the alphabet of A and let Σ_B be the alphabet of B. If Σ_A and Σ_B are not disjoint, define homomorphisms h_1 and h_2 such that h_1 maps symbols in Σ_A into corresponding symbols in $\Sigma_{A'}$, and h_2 maps symbols in Σ_B to corresponding symbols in $\Sigma_{B'}$. The alphabets $\Sigma_{A'}$ and $\Sigma_{B'}$ are selected so as to be disjoint. The Cartesian product of A and B is defined to be some interleaving of the strings in $h_1(A)$ and $h_2(B)$. In the case of the solution set formula, the only Cartesian product is $C \times C$, which is represented by some interleaving of the strings in $h_1(C)$ and $h_2(C)$.

Let X be a subset of $h_1(A) \times h_2(B)$, where h_1 and h_2 are defined as described above. The projection $\mathit{first}(X)$ is computed by erasing from strings in X symbols that belong to $\Sigma_{B'}$, where $\Sigma_{B'}$ is the alphabet of $h_2(B)$. This is accomplished by a homomorphism that maps every symbol in $\Sigma_{B'}$ to the empty string, ϵ. The resulting strings are members of $\Sigma^*_{A'}$, where $\Sigma_{A'}$ is the alphabet of $h_1(A)$. However, strings in $\mathit{first}(X)$ should be composed of symbols from Σ_A, not symbols from $\Sigma_{A'}$. A second homomorphism is applied that maps symbols in $\Sigma_{A'}$ to corresponding symbols in Σ_A. This is the inverse of h_1. The last two homomorphisms can be accomplished by a single homomorphism, h', that maps symbols in $\Sigma_{B'}$ onto ϵ, and maps symbols in $\Sigma_{A'}$ onto symbols in Σ_A. This definition of projection in terms of homomorphisms is summarized in Figure 3.1.

The closure properties of languages under arbitrary homomorphisms are well known—specifically, the regular, context free, and recursively enumerable languages are closed under arbitrary homomorphisms, and the rest are not.[1]

[1]More precisely, full trios and full AFLs are closed under arbitrary homomorphisms [Hopcroft and Ullman, 1979].

Let X be a subset of $h_1(A) \times h_2(B)$ where
$h_1(\sigma_i) =$ the i^{th} symbol in $\Sigma_{A'}$
$h_2(\sigma_i) =$ the i^{th} symbol in $\Sigma_{B'}$
$\Sigma_{A'} \cap \Sigma_{B'} = \emptyset$

$first(X) = h'(X)$ where

$$h'(\sigma_i) = \begin{cases} \text{the } i^{th} \text{ symbol in } \Sigma_A & \text{if } \sigma_i \in \Sigma_{A'} \\ \epsilon & \text{if } \sigma_i \in \Sigma_{B'} \end{cases}$$

Figure 3.1: Projection Defined as a Homomorphism.

The closure properties of languages under projection, intersection, intersection with a regular grammar, and complement are summarized in Table 3.4. The closure properties of Cartesian product depend on the mapping, as discussed earlier.

Operations	Language					
	Regular	DCFL	CFL	CSL	recursive	r.e.
$\cap$	√			√	√	√
$\cap R$	√	√	√	√	√	√
complement	√	√		?	√	
projection (homomorphisms)	√		√			√

Table 3.4: Closure Under Operations Needed to Compute the Solution Set.

3.3.2 Computability of Solution Set

The closure information in Table 3.4 is sufficient to determine the most expressive class of languages for C and P for which the solution set is recursively enumerable (computable). First, expressiveness bounds on $(C \times C) \cap P$ are established from the r.e. bounds on the solution set $(\overline{first((C \times C) \cap P)} \cap C)$ using the known closure properties of projection, intersection, and complement. The next step is to establish expressiveness bounds on C and P from the bounds on $(C \times C) \cap P$. The dependence

between the closure properties of languages under Cartesian product and the mapping used to represent the product makes this somewhat imprecise, but an effective bound can still be obtained on the expressiveness of C and P.

3.3.2.1 Expressiveness Bounds on $(C\times C)\cap P$

The first step is to establish expressiveness bounds on $(C\times C)\cap P$. This is accomplished with the following theorem.

Theorem 1 *$\overline{\text{first}((C\times C)\cap P)}\cap C$ is computable (r.e.) if and only if $(C\times C)\cap P$ is at most context free and C is at most r.e.*

Proof. The proof of this theorem follows from the closure properties of the language representing $(C\times C)\cap P$ under projection, complement, and intersection. If $(C\times C)\cap P$ is context free, then $\mathit{first}((C\times C)\cap P)$ is also context free. Closing the context free languages under complement yields the recursive languages, so the set $\overline{\mathit{first}((C\times C)\cap P)}$ could require a recursive language to express. Intersecting this set with C yields the solution set. Both recursive and r.e. languages are closed under intersection, so the solution set is r.e. as long as C is at most r.e. (i.e., C is not uncomputable).

If $(C\times C)\cap P$ is not context free, then $\mathit{first}((C\times C)\cap P)$ can be recursively enumerable, since this is the next most expressive language that is closed under projection. Recursively enumerable sets are not closed under complement. The complement of a recursively enumerable set is not recursively enumerable.[2] Therefore, $\overline{\mathit{first}((C\times C)\cap P)}$ is not computable. Intersecting $\overline{\mathit{first}((C\times C)\cap P)}$ with C yields the solution set. The intersection of an uncomputable set with any other set is also uncomputable, so the solution set is uncomputable. □

3.3.2.2 Expressiveness Bounds on C and P

The expressiveness bounds on C and P can be derived from the bounds on $(C\times C)\cap P$ established in the previous subsection. $(C\times C)\cap P$ can be at most context free.

[2]For every r.e. language, L, $\overline{L}$ is not r.e. [Hopcroft and Ullman, 1979]. This is stronger than simply stating that the r.e. languages are not closed under complement. If only the latter were true, it might be possible to find r.e. languages whose complements are also r.e. However, the first statement says that there are no such languages.

Context free sets are not closed under intersection, so $C \times C$ and P cannot both be context free. Table 3.4 indicates that at most, one of $C \times C$ and P can be regular and the other can be context free. This follows from the closure of context free grammars under intersection with regular grammars.

The expressiveness bound on C depends on what mapping is used to represent $C \times C$, since different mappings yield different closure properties for languages under Cartesian product. Regular grammars are closed under Cartesian product for the concatenation mapping, and for arbitrary interleavings (e.g., shuffling). This follows from the closure of regular grammars under concatenation and interleaving [Hopcroft and Ullman, 1979]. Context free grammars are closed under concatenation, but not under arbitrary interleavings [Hopcroft and Ullman, 1979]. C can be at most context free if the concatenation mapping is used, or at most regular if an interleaving mapping such as *shuffle* is used.

Another possible set representation is the null representation. The only set expressible in this representation is the empty set. An instantiation of KII that represents P with the null representation cannot use any preference information—the P set is always empty. When P is represented this way, the solution set equation reduces to just C. This follows from the closure of the null representation under intersection with any set (even an uncomputable one). The intersection of any set with the empty set is just the empty set. Since the solution set reduces to C when P is represented by the null representation, C can be at most recursively enumerable.

It is also possible to choose the null representation for C instead of for P. However, choosing this representation for C means that C is always empty. When C is always empty, so is the solution set, so this is not a very useful representation for C.

Table 3.5 summarizes the expressiveness bounds on $(C \times C) \cap P$, C, and P, under the assumption that the representations for C and P are closed under the given mapping for Cartesian product.

The requirement that the solution set be computable restricts the choice of languages for C and P to at most context free for one of them, and regular for the other. This holds when the mapping for Cartesian product is concatenation, but for interleaving-type mappings, C can be at most regular, and P can be at most context free.

C	P	$(C \times C) \cap P$	$\overline{first((C \times C) \cap P)} \cap C$
$\leq$ regular	$\leq$ regular	$\leq$ regular	$\leq$ regular
$\leq$ regular	$\leq$ DCFL	$\leq$ DCFL	$\leq$ recursive
$\leq$ regular	$\leq$ CFL	$\leq$ CFL	$\leq$ recursive
$\leq$ CFL	$\leq$ regular	$\leq$ CFL	$\leq$ recursive
$\geq$ CFL	$\geq$ CFL	$\geq$ CSL	uncomputable
		$>$ CFG	uncomputable
null	$\leq$ r.e.	null	null
$\leq$ r.e.	null	null	$\leq$ r.e.

Table 3.5: Summary of Expressiveness Bounds.

The integration operator further requires that the language for C be closed under intersection, and that the language for P be closed under union. Both the regular and context free languages are closed under union, so both of these are appropriate representations for P. Of these two languages, only the regular languages are closed under intersection, so C is limited to the regular languages. The only choice allowed by the restrictions of the integration operator is to represent P as at most a context free grammar, and C as at most a regular grammar.

It is possible for C to be context free and P to be regular under certain limited conditions. Context free languages are not closed under intersection, but they are closed under intersection with regular sets. This means that at most one of the $\langle C, P \rangle$ pairs being integrated can have a context free C set as long as the C sets of the remaining pairs are regular. Integrating all of these pairs results in a $\langle C, P \rangle$ pair in which C is context free and P is regular.

Chapter 4

RS-KII

RS-KII is an implementation of KII in which C, and P are represented as regular sets. This section describes the regular set representation, and implementations of all the KII operations for regular sets: *integrate*, *enumerate*, and the solution-set queries. Translators are not discussed since translator specifications and implementations are not fixed, but are provided by the user for each knowledge source and hypothesis space.

4.1 The Regular Set Representation

A regular set is specified by a regular grammar, and contains all strings recognized by the grammar. Regular sets are closed under intersection, union, complement, concatenation, and a number of other useful operations [Aho *et al.*, 1974]. Since the solution set is defined in terms of these operations, the solution set is also expressible as a regular grammar.

For every regular grammar there are equivalent deterministic finite automata (DFAs). Since DFAs are easier to work with than regular grammars, regular sets are implemented as DFAs. DFAs are discussed in Section 4.1.1, and the set operations are defined in Section 4.1.2.

4.1.1 Definition of DFAs

A DFA is defined as a 5-tuple $\langle Q, s, \delta, F, \Sigma \rangle$ where Q is a finite set of states, $s \in Q$ is the start state, δ is a state transition function from $Q \times \Sigma$ to Q, F is a set of final

(accept) states, and Σ is an alphabet of input symbols. A second move function, δ^* from $Q \times \Sigma^*$ to Q, that is defined over strings instead of single symbols can be derived from δ. For example, $\delta^*(q, w)$ returns the state reached from q by path $w \in \Sigma^*$. A string $w \in \Sigma^*$ is accepted by a DFA iff $\delta^*(s, w) \in F$.

Every regular set can be represented by several equivalent DFAs. Among these DFAs there is exactly one *minimal* DFA[Hopcroft and Ullman, 1979]. A minimal DFA has the fewest states needed to recognize a given regular set. A non-minimal DFA can be minimized as follows.

1. Remove all states that can not be reached from the start state.

2. Remove all states that can not reach an accept state (these are *dead states*).

3. Merge equivalent states.

Two states, p and q, are equivalent if for every input string w, $\delta^*(p, w)$ is in F if and only if $\delta^*(q, w)$ is in F. That is, the DFAs $\langle Q, p, \delta, F, \Sigma \rangle$ and $\langle Q, q, \delta, F, \Sigma \rangle$ recognize exactly the same strings. There are well known algorithms for minimizing DFAs, but we will not describe them here. See [Hopcroft and Ullman, 1979] for more information on this topic. In general, minimizing a DFA takes time $O(|\Sigma||Q|^2)$.

4.1.2 Definitions of Set Operations

The set operations used by KII for integrating $\langle H, C, P \rangle$ tuples are intersection and union. The representation for C must be closed under intersection, and the representation for P must be closed under union. The solution set is defined in terms of complement, Cartesian product, projection, intersection and union. The representations for C and P do not need to be closed under these operations, since the solution set may require a more expressive representation than that used for either C or P. However, regular grammars happen to be closed under all of these operations. This guarantees that the solution set can also be expressed as a regular grammar. These operations are defined below for regular grammars.

Finally, an operation is needed for computing the transitive closure of P. The enumeration operator assumes that P is transitively closed, but this condition is not preserved by integration. An operator is needed for transitively closing P prior to

enumerating the solution set. This operator is described below, along with the other set operations.

4.1.2.1 Union

The union of two DFAs, A and B, results in a DFA that recognizes a string if the string is recognized by either A or B. The DFA for the union can be constructed from A and B as shown in Figure 4.1. A state in this DFA is a pair of states, $\langle q_A, q_B \rangle$, one from A and one from B. The move function for the union is computed from the move functions of A and B. On input σ, $A \cup B$ simulates a move in A from q_A and a move in B from q_B, both on input σ. A state $\langle q_A, q_B \rangle$ in the union is a final state if q_A is a final state in A or if q_B is a final state in B.

$$
\begin{aligned}
\langle Q, s, \delta, F, \Sigma \rangle &= \langle Q_1, s_1, \delta_1, F_1, \Sigma_1 \rangle \cup \langle Q_2, s_2, \delta_2, F_2, \Sigma_2 \rangle \\
\text{where} & \\
Q &= Q_1 \times Q_2 \\
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F &= F_1 \times Q_2 \cup Q_1 \times F_2 \\
\Sigma &= \Sigma_1 \cup \Sigma_2
\end{aligned}
$$

Figure 4.1: Definition of Union.

4.1.2.2 Intersection

A string is accepted by the intersection of two DFAs, A and B, if it is accepted by both A and B. The DFA for the intersection of A and B can be constructed from A and B as shown in Figure 4.2. As with union, a state in the DFA for $A \cap B$ is a pair of states, $\langle q_A, q_B \rangle$, one from A and one from B. On input σ, $A \cap B$ simulates a move in A from q_A and a move in B from q_b, both on input σ. If both q_A and q_B are accept states in their respective DFAs, then $\langle q_A, q_B \rangle$ is an accept state in the intersection.

$$
\begin{aligned}
\langle Q, s, \delta, F, \Sigma \rangle &= \langle Q_1, s_1, \delta_1, F_1, \Sigma_1 \rangle \cap \langle Q_2, s_2, \delta_2, F_2, \Sigma_2 \rangle \\
\text{where} & \\
Q &= Q_1 \times Q_2 \\
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F &= F_1 \times F_2 \\
\Sigma &= \Sigma_1 \cap \Sigma_2
\end{aligned}
$$

Figure 4.2: Definition of Intersection.

4.1.2.3 Complement

The complement of set A with respect to the universe Σ^* is written $\overline{A}$. A string is in $\overline{A}$ if and only if it is not accepted by the DFA for set A. The DFA for $\overline{A}$ is exactly the same as the DFA for A, except that accept states in $\overline{A}$'s DFA correspond to the reject states in A's DFA, and vice versa. A reject state is any state that is not an accept state. The construction for the complement of a DFA is shown in Figure 4.3.

$$
\langle Q, s, \delta, \overline{F}, \Sigma \rangle = \overline{\langle Q, s, \delta, F, \Sigma \rangle}
$$

Figure 4.3: Definition of Complement.

4.1.2.4 Cartesian Product

The Cartesian product of two sets, A and B, is a set of tuples $\{\langle a, b \rangle \mid a \in A \text{ and } y \in B\}$. A set of tuples can not be directly represented as a regular grammar, or indeed as the language of any grammar. This is because the language of a grammar consists of strings, not tuples *per se*. A mapping between tuples and strings is required. Under mapping $\mathcal{M}$, the set $A \times B$ is represented by a regular grammar whose language is $\{\mathcal{M}(x, y) \mid \langle x, y \rangle \in A \times B\}$. This language is also denoted as $\mathcal{M}(A \times B)$. The choice of mapping greatly determines which sets of tuples can be expressed by regular grammars.

One obvious mapping is concatenation. That is, the Cartesian product of two regular sets is represented by the concatenation of the two sets. Formally, $\mathcal{M}(\langle x, y\rangle) = xy$, and $\mathcal{M}(A{\times}B) = AB$. Regular grammars are closed under concatenation, so regular grammars are also closed under Cartesian product for this mapping.

A less obvious mapping is to map $\langle x, y\rangle$ onto a string w where w is an interleaving of the strings x and y. For example, the tuple $\langle x_1 x_2 \ldots x_n, y_1 y_2 \ldots y_n\rangle$ would be represented by the string "$x_1 y_1 x_2 y_2 \ldots x_n y_n$", where the symbols of x and y alternate. This mapping is called *shuffling*. Under this mapping, $A{\times}B$ is represented by $\mathit{shuffle}(A, B)$, which is the set $\{\mathit{shuffle}(x, y) \mid x \in A \text{ and } y \in B\}$. Regular grammars are closed under shuffling. A DFA for the shuffle of any two DFAs is specified in Section 4.1.2.5. This proves by construction that regular grammars are closed under shuffling.

Differences in Expressiveness. The Cartesian product of any two regular sets can be expressed as a regular set under both the concatenation and shuffle mappings. This follows from the closure of regular sets under both concatenation and shuffling. Every subset of a Cartesian product that can be expressed as a regular grammar under the concatenation mapping can also be expressed as a regular grammar under the shuffle mapping. However, some subsets can be expressed as regular grammars under the shuffle mapping, but not under the concatenation mapping. The shuffle mapping is strictly more expressive than the concatenation mapping in this sense.

Let S be a subset of $A{\times}B$, where A and B are sets, but not necessarily regular sets. If S can be expressed as a regular grammar under the concatenation mapping, then S can also be expressed as a regular grammar under the shuffle mapping. To see why this is so, consider that S can be written as $\bigcup_i A_i{\times}B_i$, where $A_i \subset A$ and $B_i \subset B$. S can be expressed as a regular grammar under the concatenation mapping if and only if S can be written as a finite union of $A_i{\times}B_i$ pairs, where each A_i and B_i is a regular set. Under the concatenation mapping, this union is represented as $\bigcup_{i=1}^{k} A_i B_i$, where k is finite. Regular grammars are closed under concatenation and finite union, so $\bigcup_{i=1}^{k} A_i B_i$ is a regular grammar. Under the shuffle mapping, S is represented as $\bigcup_{i=1}^{k} \mathit{shuffle}(A_i, B_i)$. Since regular sets are closed under shuffling, this is also a regular grammar. Therefore, every subset of $A{\times}B$ expressible as a regular

grammar under the concatenation mapping is also expressible as a regular grammar under the shuffle mapping.

The reverse is not necessarily true. Some subsets of a Cartesian product can be expressed as regular grammars under the shuffle mapping, but not under the concatenation mapping. For example, consider the set $\{\langle w, w\rangle \mid w \in (a|b)^*\}$. Under the concatenation mapping, $\mathcal{M}(\{\langle w, w\rangle \mid w \in (a|b)^*\}) = \{ww \mid w \in (a|b)^*\}$, which is a context sensitive language [Hopcroft and Ullman, 1979]. Under the shuffle mapping, $\mathcal{M}(\{\langle w, w\rangle \mid w \in (a|b)^*\}) = \{\mathit{shuffle}(w, w) \mid w \in (a|b)^*\}$, which is $((aa)|(bb))^*$.[1]

An Illustration. The following is a simple example that will hopefully provide some intuition for the differences between these representations. Let H be a set of strings recognized by the regular grammar $(a|b)^*\$$, where \$ indicates the end of a string. Let P be the preference $\{\langle x, y\rangle \in H \times H \mid x < y\}$, where $<$ is the standard dictionary ordering (e.g., "$aab\$ < ab\$$").

P can be expressed under the shuffled mapping, but not under the concatenation mapping. In the shuffled mapping, the tuple $\langle x = x_1x_2\ldots, y = y_1y_2\ldots\rangle$ maps onto the string $x_1y_1x_2y_2\ldots$. If $x_1 = a$ and $y_1 = b$, then $x < y$, so $\langle x, y\rangle$ is in P. If $x_1 = b$ and $y_1 = a$, then $y < x$, so $\langle x, y\rangle$ is not in P. If $x_1 = y_1$, then a similar comparison is made between x_2 and y_2. If x_2 and y_2 are equal, then x_3 and y_3 are compared, and so on until the compared symbols differ, or one of the symbols is the "\$" string termination symbol. If x terminates before y, but the two strings are equal up to that point, then x comes before y in dictionary order, so $\langle x, y\rangle$ is also in P. If both terminate at the same time, or x is longer than y, then $x \not< y$, so $\langle x, y\rangle$ is not in P. A regular grammar that recognizes $\{\mathit{shuffle}(x, y) \mid \langle x, y\rangle \in (a|b)^*\$ \times (a|b)^*\$ \text{s.t.} x < y\}$ is shown in Figure 4.4

This grammar recognizes a shuffled string, $x_1y_1x_2y_2\ldots$, if adjacent pairs of symbols, x_iy_i, in the string are the same up to some pair x_ky_k. This is recognized by the production SAME. In the next pair, $x_{k+1}y_{k+1}$, x_{k+1} must be lexicographically less than y_{k+1}, or x_{k+1} must be the end-of-string symbol (\$). These conditions are recognized by X_LESS_THAN_Y and $\$(a|b)$, respectively. In the first case, the remaining symbols alternate between symbols from x and y until one of the strings has no

[1] $\mathit{shuffle}(x_1x_2\ldots x_n, x_1x_2\ldots x_n) = x_1x_1x_2x_2\ldots x_nx_n$. Symbol x_i can be either a or b, so $x_1x_1x_2x_2\ldots x_nx_n$ is a member of $((aa)|(bb))^*$.

$$
\begin{array}{rcl}
\text{S} & \rightarrow & \text{SAME X\_LESS\_THAN\_Y ANY\_XY} \\
 & \rightarrow & \text{SAME } (\$(a|b)) \text{ ANY\_Y} \\
\text{SAME} & \rightarrow & (aa)|(bb) \\
\text{X\_LESS\_THAN\_Y} & \rightarrow & ab \\
\text{ANY\_XY} & \rightarrow & (a|b)(a|b) \text{ ANY\_XY} \\
 & \rightarrow & (a|b)\$ \text{ ANY\_X} \\
 & \rightarrow & \$(a|b) \text{ ANY\_Y} \\
\text{ANY\_X} & \rightarrow & (a|b)^*\$ \\
\text{ANY\_Y} & \rightarrow & (a|b)^*\$
\end{array}
$$

Figure 4.4: Regular Grammar for $\{shuffle(x, y) \mid x, y \in (a|b)^*\$ \text{ s.t. } x < y\}$

more symbols. After this, the remaining symbols are from the longer string. Strings of this form are recognized by ANY_XY. In the second case, the string x has terminated, so the remaining symbols are all from y. Strings of this form are recognized by ANY_Y.

P cannot be recognized by a regular grammar under the concatenation grammar. Under the concatenation mapping, $\langle x, y \rangle$ maps onto xy. Every symbol in x is seen by the DFA for the grammar before any symbol of y is seen. The DFA for the grammar would have to store enough information about x to determine whether $x < y$. However, in order to make this determination, all the symbols of x must be saved. The string x can be arbitrarily long, but by definition a DFA can have only a finite number of states. Therefore, the DFA cannot store enough information about x to make the determination. The shuffled grammar gets around this problem by changing the order in which the symbols are seen, so that all the information needed to make the determination is available locally. Very little state has to be saved, if any.

More formally, P cannot be expressed as a regular grammar under the concatenation mapping because P would not be a regular language. That this language is not regular can be proven by the pumping lemma, which says that a language L is not regular if for all n it is possible to select an integer i and a string z from L such that for every way of breaking z into strings u, v, and w, the string uv^iw is not in L. The length of uv must be less than n, and v cannot be the empty string.

P is a set of strings of the form $w\alpha w\beta$, where w is a string in $(a|b)^*$, and either α is in $a(a|b)^*$ and β is in $b(a|b)^*\$$, or $\alpha = \$$ and β is a string in $(a|b)^+\$$. For an arbitrary but fixed n, let z be the string $b^n a\$b^n b\$$. For all ways of breaking z into u, v, and w such that $|uv| \leq n$ and v is not empty, uv is the string b^k where $1 \leq k \leq n$, and w is the string $b^{n-k}a\$b^n b\$$. The string string $uv^i w$ is $b^{k-|v|}b^{|v|i}b^{n-k}a\$b^n b\$$, which is equivalent to $b^n b^{|v|(i-1)}a\$b^n b\$$. This string is not in P when $|v|(i-1) \geq 1$, since $b^n b^{|v|(i-1)}a\$$ comes after $b^n b\$$ in dictionary order under this condition. Since v must be chosen to be non-empty, $|v|(i-1) \geq 1$ when $i > 1$. Therefore, there exists at least one i for which $uv^i w$ is not in P when uvw is in P. Therefore, according to the pumping lemma, P is not regular.

Definition of Cartesian Product as Shuffling. Because of superior expressive power of the shuffle mapping, RS-KII defines the Cartesian product of two regular sets, A and B, as *shuffle*(A, B). The DFA for *shuffle*(A, B), is constructed from DFAs A and B as shown in Figure 4.5.

The input to the DFA are strings of the form *shuffle*(x, y). As mentioned above, this returns a string in which the symbols from x and y alternate. This is defined here a little differently than it was above in order to address some subtle points. First, each symbol is preceded by an identifier that indicates whether the symbol belongs to x or y. This helps the DFA sort out the symbols. For example, *shuffle*$(abc, xyz) =$ $1a\,2x\,1b\,2y\,1c\,2z$. Second, x may have fewer symbols than y, or vice versa. If one string has fewer symbols than the other, then the symbols alternate until the shorter string is exhausted, after which the remaining symbols are all from the longer string. For example, *shuffle*$(abcd, xy) = 1a\,2x\,1b\,2y\,1c\,1d$.

The DFA for *shuffle*(A, B) processes the input string *shuffle*(x, y) by simulating moves in A on the symbols from x and simulating moves in B on the symbols from y. The shuffled DFA accepts the string *shuffle*(x, y) iff x is accepted by A and y is accepted by B.

More formally, a state in the shuffled DFA is a pair $\langle q_A, q_B \rangle$ where q_A is a state in A, q_B is a state in B. On input σ, a move is simulated in A if σ is from Σ_A, and simulated in B if σ is from Σ_B. The shuffled DFA accepts when both A and B are in accept states.

$$
\begin{aligned}
\langle Q, s, \delta, F, \Sigma \rangle &= \langle Q_1, s_1, \delta_1, F_1, \Sigma_1 \rangle \times \langle Q_2, s_2, \delta_2, F_2, \Sigma_2 \rangle \\
\text{where} & \\
Q &= Q_1 \times Q_2 \\
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma_{id}\, \sigma) &= \begin{cases} \langle \delta_1(q_1, \sigma), q_2 \rangle & \text{if } \sigma_{id} = 1 \\ \langle q_1, \delta_2(q_2, \sigma) \rangle & \text{if } \sigma_{id} = 2 \end{cases} \\
F &= F_1 \times F_2 \\
\Sigma &= \Sigma_1 \cup \Sigma_2 \cup \{1, 2\} \text{ where 1 \& 2 not in } \Sigma_1 \text{ or } \Sigma_2
\end{aligned}
$$

Figure 4.5: Definition of Cartesian Product.

4.1.2.5 Projection

The function *first*(A) projects a set of pairs, A, onto the set of first elements of the pairs. For example, $\mathit{first}(\{\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \ldots, \langle x_n, y_n \rangle\})$ returns $\{x_1, x_2, \ldots, x_n\}$. As discussed in Section 4.1.2.4, A is a set of pairs represented by a regular set of the form $\{\mathcal{M}(x, y) \mid \langle x, y \rangle \in A\}$, where $\mathcal{M}$ is a mapping from x, y pairs onto strings. Given this representation for A, the regular set representing $\mathit{first}(A)$ is of the form $\{x \mid \exists y\, \mathcal{M}(x, y) \in A\}$, where $\mathcal{M}(x, y)$ is the same mapping used in A.

The DFA for $\mathit{first}(A)$ recognizes a string x if there is a string y such that $\mathcal{M}(x, y)$ is recognized by the DFA for A. The DFA is provided with the string x as input, but it must make a guess as to what y should be. It can then test whether $\mathcal{M}(x, y)$ is recognized by the DFA for A.

If the mapping $\mathcal{M}$ is concatenation, then the DFA for $\mathit{first}(A)$ is relatively straightforward. It simulates moves in the DFA for A on input x. After consuming x, the DFA is in state q. If there is a path from q to an accept state, then there exists a y such that $\mathcal{M}(x, y)$ is accepted by the DFA for A. The string y corresponds to the path. If there is no such path, so that q is a dead state, then x is not accepted by the DFA for $\mathit{first}(A)$.

RS-KII does not use the concatenation mapping, however. It uses the shuffle mapping instead, due to its greater expressiveness (see Section 4.1.2.4). The shuffle mapping interleaves the symbols from x and y, which complicates the DFA for

first(A). The DFA is provided with x as input, but must guess a value for y. If *shuffle*(x, y) is recognized by the DFA for A, then x is in *first*(A). The symbols of x and y are interleaved by shuffling, so that a symbol from x is followed by one from y. After simulating a move in A on symbol x_i from x, a symbol is guessed for y_i, the next symbol of y, and a move is simulated in A on y_i.

A DFA that makes "guesses" of this sort is most easily described by a non-deterministic DFA (NDFA). An NDFA is just like a DFA, except that it can have a choice of several next-states on any given input. On any given move, one of the possible next-states is selected arbitrarily, and the NDFA moves to this state. The possible next-states are shown as a set of states in the definition of the NDFA's δ function. Every NDFA can be converted into an equivalent DFA. The NDFA for the projection will be described first, for clarity, and then this grammar will be converted into an equivalent DFA.

Constructing the NDFA. The NDFA for *first*(A) is constructed as follows. The input to the NDFA is a string x. The NDFA guesses a string y, such that *shuffle*(x, y) is in A. On each input, σ_x from x, the NDFA simulates a move in A on σ_x. This is followed by a move in A on symbol σ_y. This second symbol is the NDFA's guess for the next symbol in y. The set of possible guesses for σ_y is the alphabet for y, Σ_y. Thus from a state q in A, the NDFA can move on input σ_x to any state in $\{\delta_A(q', \sigma_y) \mid q' = \delta_A(q, \sigma_x) \text{ and } \sigma_y \in \Sigma_y\}$. The actual next state is selected non-deterministically on every move. The NDFA effectively creates an input string for A composed of alternating symbols from x and its guess for y.

An NDFA accepts a string x if there is some sequence of non-deterministic moves on input x that will lead to an accept state. In other words, the NDFA described above accepts x if there is some way to select symbols for y such that *shuffle*(x, y) is recognized by A.

Modifying the NDFA. This NDFA works fine as long as x and y are the same length, but must be modified slightly to deal with cases where x and y are of different lengths. Let $\langle x, y \rangle$ be a pair such that *shuffle*(x, y) is in A. If x is shorter than y, then *shuffle*(x, y) alternates symbols from x and y until x is exhausted, after which the remaining symbols are all from y. After the NDFA sees the last symbol in x,

it will have to guess the remaining symbols in y, and make moves in A on each of those symbols. The NDFA described above does not do this—it guesses only one y symbol for each symbol in x.

A similar condition holds when x is longer than y. In this case, *shuffle*(x, y) alternates symbols from x and y until y ends, after which the remaining symbols are all from x. After y terminates, the NDFA should not make any guesses for y. This is equivalent to guessing the empty string for y on each move. However, the NDFA described above does not guess a string for y after each symbol from x, but instead guesses a single symbol for y.

For both cases, the NDFA must be able to guess a string of symbols for y after each symbol from x instead of guessing only a single symbol. The NDFA can be made to do this by changing the next-state function as follows. From state q on input σ_x, the NDFA can move to any state in $\{\delta_A(q', w) \mid w \in \Sigma_y^* \text{ and } q' = \delta_A(q, \sigma_x)\}$. Thus on every input x, the NDFA can move to any state in A reachable by a string in $\sigma_x \Sigma_y^*$. In most cases, the only states reachable in A from q are of the form $\sigma_x \sigma_y$. In this case the modified NDFA behaves just like the NDFA described above. However, when x and y are differing lengths, and all of the symbols in one of the strings have been exhausted, there will be states reachable from q of the form $\sigma_x \Sigma_y^*$.

One final modification is also needed. A expects symbols in x to be preceded by the identifier "1", and symbols in y to be preceded by the identifier "2". The NDFA's move function must be modified to provide these additional identifier symbols to A. The NDFA with all of the above modifications is shown in Figure 4.6.

$$
\begin{aligned}
\langle Q, s, \delta', F, \Sigma \rangle &= \mathit{first}(\langle Q, s, \delta, F, \Sigma \rangle) \\
\text{where} & \\
\delta'(q, \sigma) &= \{\delta(q', w) \mid w \in (2\,\Sigma)^* \text{ and } q' = \delta(q, 1\,\sigma)\}
\end{aligned}
$$

Figure 4.6: NDFA for Projection (First).

Converting the NDFA to a DFA. The NDFA in Figure 4.6 can be converted into an equivalent DFA. Each state in the DFA corresponds to a set of states in the

NDFA. The state moved to from state q in the DFA on input σ is the set of NDFA states that can be reached non-deterministically on input σ from any of the NDFA states in q. A set of NDFA states is an accept state in the DFA if the set contains at least one state that is an accept state in the NDFA. The equivalent DFA for the NDFA of Figure 4.6 is shown in Figure 4.7.

$$
\begin{aligned}
\langle Q', s', \delta', F', \Sigma \rangle &= \mathit{first}(\langle Q, s, \delta, F, \Sigma \rangle) \\
\text{where} & \\
Q' &= 2^Q \ (\textit{the power set of } Q) \\
s' &= \{s\} \\
\delta'(\{q_1, q_2, \ldots, q_n\}, \sigma) &= \bigcup_{q_i} \{\delta(q', w) \mid w \in (2\,\Sigma)^* \text{ and } q' = \delta(q_i, 1\,\sigma)\} \\
F' &= \{q \in 2^Q \mid q \cap F \neq \emptyset\}
\end{aligned}
$$

Figure 4.7: DFA for Projection (First).

DFA for Second. Although it is not strictly necessary, the function $\mathit{second}(A)$ is occasionally useful. This function projects a set of pairs onto their second elements. The DFA for $\mathit{second}(A)$ is essentially the same as the DFA for $\mathit{first}(A)$, except that guesses are made for the first element instead of the second. The DFA is shown in Figure 4.8.

$$
\begin{aligned}
\langle Q', s', \delta', F', \Sigma \rangle &= \mathit{first}(\langle Q, s, \delta, F, \Sigma \rangle) \\
\text{where} & \\
Q' &= 2^Q \\
s' &= \{s\} \\
\delta'(\{q_1, q_2, \ldots, q_n\}, \sigma) &= \bigcup_{q_i} \{\delta(q', 2\,\sigma) \mid q' = \delta(q_i, w) \text{ and } w \in (1\,\Sigma)^*\} \\
F' &= \{q \in 2^Q \mid q \cap F \neq \emptyset\}
\end{aligned}
$$

Figure 4.8: DFA for Projection (Second).

4.1.2.6 Transitive Closure

The preference set, P, is assumed to be transitively closed. However, this condition is not preserved by integration (union). An explicit operator is therefore defined for the purpose of re-establishing the transitive closure of a preference set.

Let $tc(P)$ be the transitive closure of P. The regular grammar for $tc(P)$ accepts $\langle x, y \rangle$ if and only if $\langle x, y \rangle \in P$, or there is a finite sequence of one or more strings, $z_1, z_2, \ldots, z_n$, such that $\langle x, z_1 \rangle \in P$ and $\langle z_1, z_2 \rangle \in P$ and $\langle z_2, z_3 \rangle \in P$ and ... and $\langle z_n, y \rangle \in P$. That is, $\langle x, y \rangle$ is in the transitive closure of P either if $\langle x, y \rangle$ is in P, or if there is a finite sequence of relations in P connecting x to y.

There are a number of algorithms for computing the transitive closure of a binary relation defined over a finite set of objects (e.g., Warshall's algorithm [Warshall, 1962]). Every preference set, P, expressible as a regular grammar can be mapped onto a finite relation. The finite relation can be transitively closed using one of the standard algorithms, and the result mapped back into a regular grammar for the transitive closure of P.

Every regular set can be expressed as a finite union of This is simplest to see when P is expressed as a regular grammar under the concatenation mapping. Under this mapping, the regular grammar for P can be expressed as $\bigcup_{i=1}^{k}(A_i B_i)$, where A_i and B_i are regular subsets of the hypothesis space, H (see Section 4.1.2.4). Every hypothesis in A_i is less preferred than every hypothesis in B_i. That is, $A_i < B_i$. P can be mapped onto a finite relation, R, over the A_i and B_i sets. Each of these sets is a single object, X_i, in R. The finite relation is then transitively closed. For every pair of relations $X_1 < X_2$ and $X_2 < X_3$ in R, the relation $X_1 < X_3$ is added to R if it is not already there. At most m such relations are added, where $m \leq k^2$. The set $tc(P)$ is $P \cup \bigcup_{i=1}^{m} R_i$, where R_i is one of the relations added to R. Each relation is of the form $X_i < X_j$, and is represented by the regular grammar $X_i X_j$.

If P is represented with the shuffle mapping, then P is a finite union of the form $\bigcup_{i=1}^{k} \mathit{shuffle}(X_i, Y_i)$ where X_i and Y_i are regular subsets of the hypothesis space. Every hypothesis in X_i is less preferred than every hypothesis in Y_i. The transitive closure of P is computed just as it was for the concatenation mapping. A relation R is defined where $\langle X_i, Y_j \rangle$ is in R if $\mathit{shuffle}(X_i, Y_i)$ is in the finite union that specifies P. R is transitively closed using any standard algorithm (e.g., Warshall's

algorithm). The set $tc(P)$ is $P \cup \bigcup_{i=1}^{m} R_i$, where R_i is one of the relations added to R. Each of these relations is of the form $A < B$, where A and B are elements of $\{X_1, X_2, \ldots, X_k, Y_1, Y_2, \ldots Y_k\}$. The relation $A < B$ is represented by the regular grammar $shuffle(A, B)$.

4.1.3 DFA Implementation

A DFA is implemented in RS-KII in terms of the following components.

- A start state, s.
- An alphabet of input symbols, Σ.
- A next-state function, $\delta : (Q \times \Sigma) \rightarrow Q$
- A final-state function, $F : Q \rightarrow$ Boolean.
- A dead-state function, $Dead : Q \rightarrow \{t, f, ?\}$.
- An accept-all-state function, $AcceptAll : Q \rightarrow \{t, f, ?\}$.

With the exception of the last two functions, these components correspond to obvious elements of the standard $\langle Q, s, \delta, F, \Sigma \rangle$ 5-tuple that specifies a DFA. The last two functions allow for fast identification of at least some dead states and accept-all states, respectively. A dead state is a reject (non-accept) state from which there is no path to an accept state. An accept-all state is an accept state from which every path leads to an accept state (i.e., no path leads to a reject state). The dead-state and accept-all state function can only identify some dead states and accept-all states. If a state can not be identified by the appropriate function, the function returns "?" or "unknown".

Determining whether a state in a non-minimal DFA is a dead state or an accept-all state can require an exhaustive search of the DFA. However, many set operations preserve, or partially preserve, this information. These two functions provide a way to save such information. The ability to identify dead-states and accept-all states inexpensively is central to the solution-set enumeration algorithm, which is described in Section 4.2.2.

Notably absent from the list of DFA components is Q, the set of states. States are generated as needed by applying the next-state function to the current state. In order to find a path through the DFA from start state to accept state, or to determine whether a string is accepted by the DFA, only a few of the states in the DFA are usually visited. The complexity of these searches is proportional to the number of states visited. However, if Q is represented extensionally, then the time needed to create the DFA far exceeds the cost of searching it.

By not representing the states explicitly, the time and space complexity of searching the DFA can be kept proportional to the number of states visited. In order for this to work, the time and space complexity of the move-function and other components of the DFA must be significantly less than $O(|Q|)$. A DFA that meets these criteria is called an *intensional* DFA. If the time or space complexity of the components are proportional to $O(|Q|)$, then the DFA is said to be *extensional.* This occurs, for example, when the next-state function is implemented as an explicit look-up table.

DFAs constructed from set operations on other DFAs can be represented *intensionally.* Let R be the DFA resulting from a set operation over DFAs A and B. The idea is to implement the next-state function of R in terms of the next-state functions of A and B. The other components of R are implemented similarly. The space complexity of R's components is $O(1)$, and their time complexity is proportional to that of A's components and B's components. DFAs represented this way are called *recursive*, since they are defined "recursively" in terms of more primitive DFAs. All of the DFAs resulting from set operations in RS-KII are represented as recursive DFAs. Recursive DFAs are discussed further in Section 4.1.3.1

The DFAs for the H, C, and P sets are called *primitive* DFAs. These DFAs can be either intensional or extensional. Primitive DFAs are discussed further in Section 4.1.3.2.

4.1.3.1 Recursive DFAs

Recursive DFAs are the results of set operations. A DFA resulting from an expression of set operations is essentially an expression tree of DFAs, where the root is the DFA representing the result of the expression, the nodes are DFAs resulting from

intermediate set operations, and the leaves are the primitive DFAs input to the expression. The components of the DFA at each node are defined in terms of the components of the DFAs at the node's child nodes. The DFA at each node requires only enough space for pointers to its children, and definitions for the components. This is essentially constant space for each DFA.

The total space for the expression is the sum of the space consumed by each of the primitive DFAs at the leaves, plus a small constant amount of space for each node in the tree. If the tree is balanced and binary, there are about as many internal nodes as leaves. If there are n primitive DFAs, each consuming k units of space, then the total space cost is $O(nk)$. The root DFA could have as many as k^n states, depending on the set operations in the expression tree, so considerable space is saved by not representing the DFA's states explicitly.

DFAs represented as expression trees are called *recursive* DFAs. The components of these DFAs are defined recursively in terms of the components of other DFAs. Eventually, this regression must ground out in DFAs in which the components are extensionally defined. These are called *primitive* DFAs. The C and P sets specifying the integration of two $\langle C, P \rangle$ tuples are represented in RS-KII by recursive DFAs. The DFAs for the C and P sets generated by translators are primitive DFAs, by definition.

4.1.3.2 Primitive DFAs

Primitive DFAs are DFAs in which the components are implemented extensionally. The start state is represented extensionally, as is the alphabet. In practice, all of the translators for a given hypothesis space can be made to generate DFAs that use the same alphabet. Thus in practice, the alphabet is stored once, and each of the primitive DFAs have a pointer to it. There are several ways to implement the functions of a primitive DFA, but the most straightforward way is with explicit lookup tables.

It is generally a good idea to reduce the space complexity of primitive DFAs, since the complexity of recursive DFAs—such as the solution set—is proportional to the space complexity of the primitive DFAs from which the recursive DFA is constructed. The time complexities of the next-state and other functions are similarly related, so

it is also worth using implementations of these functions for primitive DFAs with as low a time complexity as possible.

A simple lookup table can consume up to $O(|Q||\Sigma|)$ units of space for the next-state function, and $O(|Q|)$ units for each of the other functions. Table lookup is a constant-time operation, so the DFAs functions can all be executed in constant time. Compressing the lookup tables can reduce their space complexity, but may increase the time complexity of table lookup. Good implementations of compressed tables have lookup times of at most $\log(n)$, where n is the size of the table. In some DFAs, the next state can be computed from the current state and the input symbol in time proportional to the size of the state. This is an effectively constant time and space implementation as long as the size of the state is not a function of some relevant scale-up variable.

Using minimal DFAs is another way to keep the space complexity down without increasing the time complexity. Translators should be able to generate minimal DFAs directly. If it is not possible to generate a minimal DFA directly, then the translator can create a non-minimal DFA and then minimize it. However, the space savings from using a minimal DFA may not be worth the computational complexity of explicit minimization.

One way to dramatically reduce the space complexity of any primitive DFA in exchange for increased time complexity is to explicitly store an equivalent NDFA, and use the NDFA to simulate the DFA's functions, such as next-state and final-state. An NDFA generally has exponentially fewer states than an equivalent DFA, and the DFA move function can be simulated from the NDFA move function in $O(n)$ time, where n is the number of states in the NDFA[Aho *et al.*, 1974]. Testing whether a state is final is also $O(n)$, as are the other functions.

4.2 RS-KII Operator Implementations

Recall that the operations defined by KII are *translation*, *integration*, *enumeration*, and *queries*. This section discusses RS-KII's implementation of the enumeration and integration operations. Translator and query implementations are not discussed. Translators depend on the hypothesis space and the knowledge being translated, so the translators are provided by the user for each hypothesis space and knowledge

source rather than being a fixed part of RS-KII. Translators for specific induction tasks are discussed in Chapter 5 and Chapter 6. The queries are implemented in terms of the enumeration operator, as described in Section 2.2.4 of Chapter 2, and do not require further discussion. The integration operator is discussed in Section 4.2.1, and the enumeration operator is discussed in Section 4.2.2.

4.2.1 Integration

As discussed in Chapter 2, integration is defined as follows.

$$\text{Integrate}(\langle D, C_1, P_1\rangle, \langle D, C_2, P_2\rangle) = \langle D, C_1 \cap C_2, P_1 \cup P_2\rangle \tag{4.1}$$

This requires intersection and union. Implementations of these set operations are described below.

4.2.1.1 Intersection

The intersection of two DFAs, A and B, is implemented as a recursive DFA, as shown in Figure 4.9. The components of this DFA are derived from the components of A and B according to the definition of intersection given in Figure 4.2. Tuples in the intersection are implemented as pairs of pointers to states in A and B. The alphabet, Σ, is $\Sigma_A \cap \Sigma_B$, the intersection of the alphabets for A and B. If Σ_A and Σ_B are different alphabets, then their intersection is computed and stored extensionally in the DFA for $A \cap B$. If Σ_A and Σ_B are the same alphabet, then $\Sigma_A \cap \Sigma_B$ is just Σ_A (or equivalently, Σ_B). In this case, the DFA for $A \cap B$ just stores a pointer to Σ_A instead of storing an explicit copy. This is the most common case, since translators with the same domain can usually be made to output DFAs with the same alphabet.

The *Dead* function takes as input a state, $\langle q_1, q_2\rangle$, in $A \cap B$, and determines whether or not the state is dead. A state is dead if there is no path from the state to an accept state. Determining this fact can require an exhaustive search of the states reachable from $\langle q_1, q_2\rangle$. However, there are cases where it can be quickly determined whether $\langle q_1, q_2\rangle$ is dead from information about q_1 and q_2. If the status of $\langle q_1, q_2\rangle$ can be quickly determined, then $Dead(\langle q_1, q_2\rangle)$ returns `true` or `false`, appropriately.

$$\langle s_1, \delta_1, F_1, Dead_1, AccAll_1, \Sigma_1 \rangle \cap \langle s_2, \delta_2, F_2, Dead_2, AccAll_2, \Sigma_2 \rangle = \langle s, \delta, F, Dead, AcceptAll, \Sigma \rangle \text{ where}$$

$$\begin{aligned}
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F(\langle q_1, q_2 \rangle) &= F_1(q_1) \wedge F_2(q_2) \\
Dead(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true if} & Dead_1(q_1) = \texttt{true or} \\ & Dead_2(q_2) = \texttt{true} \\ \text{else } \texttt{unknown} \end{cases} \\
AcceptAll(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true if} & AccAll_1(q_1) = \texttt{true and} \\ & AccAll_2(q_2) = \texttt{true} \\ \texttt{false if} & AccAll_1(q_1) = \texttt{false or} \\ & AccAll_2(q_2) = \texttt{false} \\ \text{else } \texttt{unknown} \end{cases} \\
\Sigma &= \begin{cases} \Sigma_1 \cap \Sigma_2 & \text{if } \Sigma_1 \neq \Sigma_2 \\ \text{Pointer to } \Sigma_1 & \text{if } \Sigma_1 = \Sigma_2 \end{cases}
\end{aligned}$$

Figure 4.9: Intersection Implementation.

Otherwise, it returns **unknown**, and it is up to the caller to decide whether to perform the expensive search or to do without the information.

It can be quickly determined that a state, $\langle q_1, q_2 \rangle$, in $A \cap B$ is dead if q_1 is a known dead state in A or if q_2 is a known dead state in B. This information can be obtained from the *Dead* functions for A and B, respectively. In all other cases, it is necessary to perform an exhaustive search of the states reachable from $\langle q_1, q_2 \rangle$ to determine whether it is a dead state. If q_1 and q_2 are known to be non-dead states in A and B, then there is at least one path from q_1 to an accept state in A, and at least one path from q_2 to an accept state in B. However, if there is no path in common between q_1 and q_2, then the state $\langle q_1, q_2 \rangle$ is a dead state in $A \cap B$. The *Dead* function for $A \cap B$ returns **unknown** in this case. It also returns **unknown** if A's *Dead* function returns **unknown** for q_1, or if B's *Dead* function returns **unknown** for q_2.

The *AcceptAll* function is essentially the dual of the *Dead* function. A state, $\langle q_1, q_2 \rangle$, in $A \cap B$, is an accept-all state if every path from $\langle q_1, q_2 \rangle$ leads to an accept state ($\langle q_1, q_2 \rangle$ need not be an accept state, though, unless there is a path from the

state to itself). The function *AcceptAll*($\langle q_1, q_2 \rangle$) returns `true` if it can be quickly determined that $\langle q_1, q_2 \rangle$ is an accept-all state, returns `false` if it can be quickly determined that it is not an accept-all state, and `unknown` otherwise.

The state $\langle q_1, q_2 \rangle$ is an accept-all state when q_1 is an accept-all state in A and q_2 is an accept-all state in B. The state is not an accept-all state if either q_1 or q_2 is not an accept-all state in its respective DFA. Both of these cases can be quickly determined from the *AcceptAll* functions of A and B, respectively. The accept-all status of $\langle q_1, q_2 \rangle$ is `unknown` in $A \cap B$ only if the accept-all status of q_1 is `unknown` in A or the accept-all status of q_2 is `unknown` in B.

The *AcceptAll* function "preserves" accept-all state information from A and B, in the sense that if the accept-all status of every state in A and B is known (i.e, either `true` or `false`), then the accept-all status of every state in $A \cap B$ is also known. That is, the *AcceptAll* function for $A \cap B$ never returns `unknown` for any state in $A \cap B$ unless the *AcceptAll* function for A or B returns `unknown` for some state in A or B, respectively.

The *Dead* function for $A \cap B$ does not preserve dead state information from A and B. Given two DFAs, A and B, in which all of the dead states are known, $A \cap B$ can contain states for which the *Dead* function returns `unknown`, as was described above.

4.2.1.2 Union

The union of two DFAs, A and B, is implemented similarly to intersection. The main differences are that the components of the union are derived from A and B according to the definition of union in Figure 4.1, and the alphabet Σ is $\Sigma_A \cup \Sigma_B$ instead of $\Sigma_A \cap \Sigma_B$. If Σ_A and Σ_B are the same alphabet, as is usually the case, then Σ is just a pointer to one of these alphabets. Otherwise, Σ is a pointer to a new alphabet containing the symbols in both Σ_A and Σ_B. The implementation of union is shown in Figure 4.10.

A state in $A \cup B$ is of the form $\langle q_1, q_2 \rangle$, where q_1 is a state in A and q_2 is a state in B. The function *Dead*($\langle q_1, q_2 \rangle$) returns `true` or `false`, accordingly, when it can be quickly determined whether $\langle q_1, q_2 \rangle$ is a dead state in $A \cup B$. When this determination cannot be made quickly, the function returns `unknown`. A state, $\langle q_1, q_2 \rangle$, in $A \cup B$ is

$$\langle s_1, \delta_1, F_1, Dead_1, AccAll_1, \Sigma_1 \rangle \cup \langle s_2, \delta_2, F_2, Dead_2, AccAll_2, \Sigma_2 \rangle = \langle s, \delta, F, Dead, AcceptAll, \Sigma \rangle \text{ where}$$

$$\begin{aligned}
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F(\langle q_1, q_2 \rangle) &= F_1(q_1) \vee F_2(q_2) \\
Dead(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true} \text{ if} & Dead_1(q_1) = \texttt{true} \text{ and } Dead_2(q_2) = \texttt{true} \\ \texttt{false} \text{ if} & Dead_1(q_1) = \texttt{false} \text{ or } Dead_2(q_2) = \texttt{false} \\ \text{else } \texttt{unknown} & \end{cases} \\
AcceptAll(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true} \text{ if} & AccAll_1(q_1) = \texttt{true} \text{ or } AccAll_2(q_2) = \texttt{true} \\ \text{else } \texttt{unknown} & \end{cases} \\
\Sigma &= \begin{cases} \Sigma_1 \cup \Sigma_2 & \text{if } \Sigma_1 \neq \Sigma_2 \\ \text{Pointer to } \Sigma_1 & \text{if } \Sigma_1 = \Sigma_2 \end{cases}
\end{aligned}$$

Figure 4.10: Union Implementation.

a dead state if and only if there is no path from $\langle q_1, q_2 \rangle$ to an accept state in $A \cup B$. This only occurs when there is no path from q_1 to an accept state in A, and no path from q_2 to an accept state in B. Otherwise, $\langle q_1, q_2 \rangle$ is not a dead state. Both of these conditions can be quickly determined from the *Dead* functions of A and B, respectively. The *Dead* function for $A \cup B$ only returns `unknown` for $\langle q_1, q_2 \rangle$ when the dead state status of one of q_1 and q_2 is `unknown`, and the status of the other state is either `unknown` or `false`.

The *Dead* state function for the union of two DFAs, A and B, preserves the dead state information of A and B, in that the dead state status of every state in $A \cup B$ is known if the dead state status of every state in A and B is known. Union is said to preserve dead state information.

A state $\langle q_1, q_2 \rangle$ in $A \cup B$ is an accept-all state if every path from $\langle q_1, q_2 \rangle$ leads to an accept state. The function *AcceptAll* returns `true` or `false`, accordingly, when this determination can be made cheaply, and `unknown` when it cannot. It can be cheaply determined that $\langle q_1, q_2 \rangle$ is an accept-all state if q_1 is an accept-all state in

A or if B is an accept-all state in B. Otherwise, the determination cannot be made cheaply, and $AcceptAll(\langle q_1, q_2 \rangle)$ returns `unknown`.

If neither q_1 nor q_2 are accept-all states in A and B, respectively, then it still possible that $\langle q_1, q_2 \rangle$ is an accept-all state in $A \cup B$. This occurs when the paths that do not lead to accept states from q_1 in A do lead to accept states in B from q_2, and vice-versa. In general, ascertaining this fact requires an exhaustive search of the states reachable from $\langle q_1, q_2 \rangle$ in $A \cup B$ to determine whether they are all accept states. The function $AcceptAll(\langle q_1, q_2 \rangle)$ returns `unknown` in this case. It also returns `unknown` if the accept-all status of one of q_1 and q_2 is `unknown`, and the accept-all status of the other state is either `false` or `unknown`.

The *AcceptAll* function for the union of two DFAs, A and B, does not preserve the accept-all information of A and B. Even if the accept-all status of every state in A and B is known, there can still be states in $A \cup B$ for which *AcceptAll* returns `unknown`. Union does not preserve accept-all information.

4.2.1.3 Minimizing after Integration

The integration of several $\langle H, C, P \rangle$ tuples is accomplished by by intersecting all of the C sets and computing the union of the P sets. The result is a single tuple. The intersection and union operations in RS-KII return recursive DFAs. These operations are constant time, and the resulting DFAs have a very low space complexity. However, the resulting DFAs have not been minimized, which means they can have far more states than the corresponding minimal DFAs, and that they may have states for which the *Dead* and *AcceptAll* functions return `unknown`.

The combination of extraneous states in C and P, and the presence of dead states that cannot be identified by the *DeadState* function can increase the cost of enumerating a hypothesis from the solution set by introducing similar states into the DFA for the solution set. A hypothesis in the solution set can be generated by finding a path from the start state of the DFA to one of its accept states—a simple graph search. The presence of dead states that cannot be detected by the DFA's *DeadState* function can cause backtracking. All of the state reachable from the unidentified dead state must be visited before it is obvious that the state cannot

reach an accept state, and is therefore dead. If the DFA is non-minimal, there may be many such states reachable from each unidentifiable dead state.

If the DFA's *DeadState* function can detect every dead state in the DFA, then the search will never backtrack, and the extraneous states are not detrimental. Backtracking only occurs when there is no path from the current state to any of the accept states—that is, when the current state is a dead state. If the DFAs *DeadState* function can always determine whether a given state is dead, then the states reached by each edge out of the current state can be tested by this function. One of the non-dead states is visited next. The current state always has at least one such child state. If it did not, then the current state would also be dead, and it would never have been visited in the first place.

The DFA for C is the intersection of several DFAs, and since intersection does not preserve dead state information, the DFA may have several states for which the *DeadState* function returns `unknown`. The problem is that a state, $\langle q_1, q_2 \rangle$, in the intersection is dead if q_1 or q_2 are dead in their respective DFAs, or if neither q_1 or q_2 is dead, but there is no path (string) from q_1 to an accept state in the first DFA that is also a path from q_2 to an accept state in the second DFA. The first case is easy to detect, but the second requires an exhaustive search of the states reachable from q_1 and q_2, which can be expensive.

One way to reduce this cost is to minimize the DFAs for C and P after each integration, or at least do an exhaustive search of the states in C to identify all of the dead states. However, minimization takes time proportional to $O(|\Sigma||Q|^2)$, and identifying dead states takes time proportional to $O(|Q|)$. This cost generally negates any benefit from storing the DFAs intensionally. The motivation for representing DFAs intensionally is that only a few of the states are visited while searching the DFA for a path from the start state to an accept state. Only in the worst case are all of the states visited. By minimizing the DFA, or by identifying the dead states, the cost is proportional to the number of states in the average case as well as in the worst case. Therefore, RS-KII does not minimize after integration, or try to identify dead states in C. It is generally better to identify and remove only those dead states encountered during the search, since this usually involves only some of the states in the DFA.

Although identifying and removing dead states from C after integration is not *generally* cost-effective, there is at least one case where it can be beneficial. If $\langle Q, s, \delta, F, \Sigma \rangle$ is the intersection of $\langle Q_1, s_1, \delta_1, F_1, \Sigma_1 \rangle$ and $\langle Q_2, s_2, \delta_2, F_2, \Sigma_2 \rangle$, then the size of Q is at most $|Q_1||Q_2|$. However, many of the states in Q may be dead states. If after removing these states, the size of Q is bounded by the size of the larger of the two original DFAs, then the cost of intersecting and removing dead states from k such DFAs is only $k|Q^*|^2$, where $|Q^*|$ is the number of states in the largest of the k DFAs. Removing dead states after each intersection can be much cheaper in this case than removing them after all of the DFAs have been intersected.

RS-KII does not have an explicit operation for removing dead states, but the enumeration and query operations do identify any dead states they happen to encounter. Applying a query, such as *Empty*, after each $\langle H, C, P \rangle$ tuple is integrated will identify and remove many of the dead states in C and P. This approach is used in the RS-KII emulation of the Candidate Elimination algorithm [Mitchell, 1982] in Section 6.4 in order to achieve benefits similar to those described in the previous paragraph.

4.2.2 Enumeration

The enumeration operator returns hypotheses from the deductive closure of $\langle C, P \rangle$. It is used both to select the induced hypothesis and to implement solution-set queries. *Enumerate*$(\langle C, P \rangle, A, n)$ returns a list of n hypotheses that are both in the solution set of $\langle C, P \rangle$ and in the regular set A. If there are only $m < n$ hypotheses in the intersection of A and the solution set, *Enumerate* returns all m hypotheses. The A and n arguments are necessary to implement the queries. In order to induce a hypothesis, it is only necessary to select a single hypothesis from the deductive closure. This can be done by setting $n = 1$ and $A = \Sigma^*$.

The deductive closure of $\langle C, P \rangle$ consists of the most preferred hypotheses in C, according to the preference relation P. Specifically, this is the set shown in Equation 4.2.

$$\overline{\mathit{first}((C \times C) \cap P)} \cap C \tag{4.2}$$

In this equation, P is assumed to be transitively closed. However, P is the union of several DFAs, and transitive closure is not preserved by union (see Section 2.2.3).

The first action of *Enumerate* is to reestablish the transitive closure of P by applying the *TransitiveClosure* operator (see Section 4.1.2.6). In the remainder of this section, P is assumed to be transitively closed unless stated otherwise.

One way to enumerate the solution set is to compute the DFA for the solution set, and generate strings from this DFA using a graph search. The solution set can be expressed as a DFA since regular grammars are closed under all of the operations in Equation 4.2.

Although RS-KII could compute the solution set directly using the set operations defined in Section 4.1.2, this option is not used for efficiency reasons. The DFAs resulting from the set operations are non-minimal, possibly containing a large number of dead states. When a graph search enters a region of dead states, it must eventually backtrack, since there is no way to reach an accept state from such a region. In the worst case, all the states in the region must be visited before the search discovers that it has to backtrack. The problem is that information about which states are dead is not preserved by intersection, and the only way to tell that a state is dead is to visit all of the states reachable from the state, and determine that none of them are accept states.

The amount of backtracking can be reduced by using certain regularities in the structure of the DFA to increase the amount of dead-state information available to the search. For example, P is a partial order, and therefore irreflexive, antisymmetric, and transitive. The solution set equation also contains a $C \times C$ term. This introduces a certain amount of symmetry into the DFA. These regularities can be easily exploited by a more sophisticated search algorithm, but are not captured in the DFA in such a way that a graph search can easily make use of them. These regularities are meta-information true of all solution sets.

For this reason, RS-KII enumerates the solution set using a modified branch-and-bound search, into which the regularities mentioned above are integrated. The solution set consists of the undominated hypotheses in C, where the domination relation is determined by P. This maps very well onto branch-and-bound, which also finds the undominated elements of a set. However, a few modifications are required.

The basic branch-and-bound algorithm takes a set of elements, X, and an evaluation function, $f: X \rightarrow \Re$, that assigns a real number evaluation to each hypothesis.

The algorithm returns an element of the set $\{x \in X \mid \forall y \in X\ f(x) \not< f(y)\}$. This algorithm is described in Section 4.2.2.1.

Two modifications to the basic branch-and-bound algorithm are required in order to implement the enumeration operator. *Enumerate*$(\langle C, P\rangle, A, n)$ returns n hypotheses from the set $\{x \in X \mid \forall y \in X\ x \not\prec_P y\} \cap A$, where $\prec_P$ is the partial ordering imposed by P. The first modification is to evaluate hypotheses according to the partial ordering $\prec_P$ instead of the total ordering imposed by f. This modification, described in Section 4.2.2.2, returns a single element of the set $\{x \in X \mid \forall y \in X\ x \not\prec_P y\}$, which is the deductive closure of $\langle C, P\rangle$. This is sufficient to implement *Enumerate* when $n = 1$ and $A = \Sigma^*$ (i.e., A is not used).

In order to implement *Enumerate*$(\langle C, P\rangle, A, n)$ for arbitrary values of n and A, the modified branch-and-bound algorithm must be extended to find n elements that are in both A and the deductive closure of $\langle C, P\rangle$. These modifications are described in Section 4.2.2.3.

In order to perform efficiently, the modified branch-and-bound algorithm must implement certain data structures efficiently. These issues are discussed briefly in Section 4.2.2.4.

4.2.2.1 The Basic Branch-and-Bound Algorithm

Branch-and-Bound takes a set of elements, X, and an evaluation function f, and finds an *undominated* element of X. An element x is dominated if and only if there is an element y in X such that $x <_X y$, where $<_X$ is a domination relation over elements of X. The domination relation $<_X$ can be any ordering relation on X. In the basic branch-and-bound algorithm, $<_X$ is a total ordering on X derived from the evaluation function f, where $x <_X y$ if and only if $f(x) < f(y)$

The idea behind branch-and-bound is to find an undominated element of X by pruning regions of the search space, X, in which every element in the region is dominated by the best element of another region. This is the "bound" part of the algorithm. It is followed by the "branch", which splits the un-pruned regions into several sub-regions. This splitting and pruning process continues until all but one of the regions have been pruned, and the remaining region contains only a single element, x. This element is an undominated element of X.

A version of the basic branch-and-bound algorithm based on Kumar's formulation [Kumar and Laveen, 1983, Kumar, 1992] is shown in Figure 4.11. The search space is a set of elements, X, and regions of the search space are subsets of X. The algorithm maintains a collection of subsets, L, which initially contains only X. In each iteration, a subset X_s is selected from this collection by the *Select* function, and split into smaller subsets with the *Split* function. Dominated subsets are then pruned from the collection. Subset X_i is dominated by subset X_j if for every hypothesis in X_i there exists a preferable hypothesis in X_j. Formally, X_i is dominated by X_j if and only if $(\forall h{\in}X_i)(\exists h'{\in}X_j)h <_X h'$. This process continues until there is only one subset left in the collection, and this subset contains exactly one element, x. This element is returned as the undominated element found by the search.

BranchAndBound$(X, \tilde{D}_<)$ returns $x \in X$ s.t. $\forall_{y \in X}\ x \not< y$
    X is a set of elements.
    $\tilde{D}_<$ is a domination relation on subsets of X derived from $<$.
    $<$ is a total ordering over elements of X. BEGIN
    $L \leftarrow \{X\}$
    WHILE L is not empty DO
        IF L is a singleton, $\{X_i\}$, and X_i is a singleton, $\{x\}$, THEN
            RETURN x
        ELSE
            $X_s \leftarrow Select(L)$
            Replace X_s in L with $Split(X_s)$.
            `Dominated` $\leftarrow \{X_i \in L \mid \exists X_j \in L\ (X_i\ \tilde{D}_<\ X_j)\}$
            $L \leftarrow L -$ `Dominated`
    END WHILE
    RETURN `failure`
END *BranchAndBound*.

Figure 4.11: Branch-and-Bound where $\tilde{D}_<$ is a Total Order.

The Domination Relation among Subsets. The domination relation among subsets is derived from the domination relation among individual hypotheses, $<_X$, and is expressed as a binary relation, $D_<$, where $\langle X_i, X_j \rangle$ is an element of $D_<$ if and

only if subset X_i is dominated by subset X_j. This is written $X_i \; D_< \; X_j$. If $\langle X_i, X_j \rangle$ is not an element of $D_<$, then X_i is not dominated by X_j. This is written $X_i \not\!D_< X_j$.

In order for branch-and-bound to be cost-effective, the cost of testing whether a subset is dominated and should thus be pruned must be significantly cheaper than the cost of the search avoided by pruning the subset. It may only be possible to cheaply determine the domination relation between some pairs of subsets. For this reason, The basic branch and bound algorithm uses the incomplete but inexpensive domination relation, $\tilde{D}_<$, instead of the complete but potentially expensive relation $D_<$.

A pair of subsets, $\langle X_i, X_j \rangle$ is in $\tilde{D}_<$ if and only if $X_i \; D_< \; X_j$, and this relation can be determined by an inexpensive test. If $X_i \; D_< \; X_j$, but this fact cannot be determined inexpensively, $X_i \; \tilde{D}_< \; X_j$ will be false. $X_i \; \tilde{D}_< \; X_j$ is also false if X_i is not dominated by X_j. It is always possible to cheaply determine the domination relation between singleton sets, $\{x\}$ and $\{y\}$, since this is a trivial matter of determining whether $x <_X y$, and $<_X$ is defined between every pair of elements in X.

Even though $\tilde{D}_<$ is incomplete, this does not invalidate the branch and bound algorithm. Eventually, the subsets in the collection, L, will be split to the point where the subsets in L can be compared by $\tilde{D}_<$. This requires that the *Split* function be defined so that X is eventually split into a finite number of subsets that can be compared by $\tilde{D}_<$. If X is finite, then the branch and bound algorithm is always guaranteed to halt, since in the worst case, X will be split into a finite number of singleton subsets, and singleton subsets can always be compared by $\tilde{D}_<$.

Total and Partial Orderings. The basic branch and bound algorithm assumes that $D_<$ is a total ordering over the subsets of X. When $D_<$ is a total order, there is at most one dominant subset among any collection of subsets. This guarantees that after splitting X into subsets that can be compared by $\tilde{D}_<$, all but one subset will be pruned from the collection. The remaining subset can be split further until a singleton subset is identified that dominates the remaining subsets. The single element in this set is returned as the undominated element of X. If $D_<$ is a partial order, then in any collection of subsets there may be more than one undominated subset, in which case the algorithm will not halt.

The relation $D_<$ is derived from $<_X$. If $<_X$ is a total order, and $Split(X_s)$ partitions X_s into disjoint subsets, then $D_<$ is also a total order. Given any set of elements ordered by $<_X$, there is exactly one maximal element. Therefore, between two disjoint subsets, X_i and X_j, one subset is guaranteed to contain the maximal element of $X_i \cup X_j$, and this subset dominates the other. If the subsets are not disjoint, then both X_i and X_j may contain the maximal element in which case neither dominates the other. However, this is somewhat irrelevant since the dominant element is the same in both subsets. One subset can be arbitrarily selected as the dominant one without fear of pruning away the dominant element.

If $<_X$ is a partial ordering, then so is $D_<$. For example, Let a and b be the undominated elements of X_i, and let y and z be the undominated elements of X_j. If $a <_X y$ and $z <_X b$, but there is no relation between a and z nor between b and y, then X_i does not dominate X_j, nor does X_j dominate X_i. There is no domination relation between X_i and X_j.

4.2.2.2 Branch-and-Bound with Partially Ordered Hypotheses

The deductive closure of $\langle C, P \rangle$ consists of the undominated hypotheses of C, according to P, the partially ordered dominance relation over individual hypotheses. The basic branch-and-bound algorithm finds an undominated element of a set X according to a totally ordered dominance relation, $<_X$. In order to enumerate an element of the deductive closure of $\langle C, P \rangle$, the basic branch-and-bound algorithm must be modified to accept a partially ordered dominance relation over the individual hypotheses instead of requiring a totally ordered relation.

The problem with using a partial order in the basic branch-and-bound algorithm is that the termination condition assumes the dominance relation $<_X$ is a total order. The basic algorithm terminates when the collection contains only a single subset, and this subset is a singleton. In a total ordering, there is at most one undominated element. Eventually, all of the other elements will be pruned leaving only the undominated element in the collection. In a partial ordering, there can be several undominated elements. Eventually, all of the dominated elements will be pruned leaving only the undominated elements, but since there can be more than one undominated element, the algorithm may not halt.

One solution is to modify the termination condition so that the algorithm halts as soon as one of the subsets in the collection contains nothing but undominated elements. An arbitrary element of this subset is returned by the search as the undominated element. Other subsets in the collection may also contain undominated elements, but since only one element is needed, the termination condition described above is sufficient. One advantage of this approach is that it can be easily extended to return multiple hypotheses, as will be seen in Section 4.2.2.3.

A subset X_i in the collection, L, contains only undominated elements if X_i is undominated by all of the other subsets in L, and if no hypothesis in X_i dominates any other hypothesis in X_i. The first condition is true if $(\forall X_{j\neq i} \in L)\ X_i \not\mathrel{D_<} X_j$. The second condition is true if $X_i \not\mathrel{D_<} X_i$. That is, no hypothesis in X_i dominates any other hypothesis in X_i. These two conditions can be combined into one test, namely $(\forall X_j \in L)\ X_i \not\mathrel{D_<} X_j$. If X_i contains only undominated hypotheses, then any one of them can be returned by the search as the undominated element.

In order to evaluate this termination test, it must be possible to determine whether or not $\langle X_s, X_j \rangle \in D_<$, where X_s is the selected subset, and X_j is an element of the collection. If the test is true, X_s contains only undominated hypotheses. Unfortunately, the complete dominance relation, $D_<$, is not available to the branch and bound algorithm. Only the less expensive, but incomplete, relation $\tilde{D}_<$ is available. Since this relation is incomplete, $\langle X_i, X_j \rangle \notin \tilde{D}_<$ could mean either that $\langle X_i, X_j \rangle \notin D_<$, or that the domination relation between X_i and X_j cannot be determined cheaply. In the latter case, $\langle X_i, X_j \rangle$ may or may not be a member of $D_<$.

In order to discriminate between these two cases, the relation $\tilde{D}_{\not<}$ is defined. $X_i\ \tilde{D}_{\not<}\ X_j$ is true if and only if $X_i \not\mathrel{D_<} X_j$ and this relation can be determined cheaply for X_i and X_j. The relation $X_i\ \tilde{D}_{\not<}\ X_j$ is false either if the dominance relation between X_i and X_j cannot be determined cheaply, or if $X_i\ D_<\ X_j$.. The two cases cannot be distinguished. The $D_{\not<}$ relation requires an inexpensive test for determining when one subset does not dominate another.

The modified algorithm, *BranchAndBound-2*, is shown in Figure 4.12. It uses the $\tilde{D}_{\not<}$ relation to determine when one of the subsets in the collection contains only undominated elements. An element of this subset is selected arbitrarily and returned as the undominated element found by the search. The relation $\tilde{D}_<$ is used to prune

dominated subsets. The relations $\tilde{D}_<$ and $\tilde{D}_{\not<}$ are derived from the domination relation $<_X$ over individual hypotheses. The relation $<_X$ can be a partial order.

BranchAndBound-2$(X, \tilde{D}_<, \tilde{D}_{\not<})$ returns $x \in \{x \in X \mid \forall_{y \in X}\ x \not< y\}$
    X is a set of elements
    $\tilde{D}_<$ is a domination relation on subsets of X derived from $<$
    $\tilde{D}_{\not<}$ is a not-dominated relation on subsets of X derived from $<$
    $<$ is a partial ordering over the elements in X
BEGIN
    $L \leftarrow \{X\}$
    WHILE L is not empty DO
        $X_s \leftarrow$ *Select*(L)
        IF $(X_s\ \tilde{D}_{\not<}\ X_i)$ for every X_i in L and $(X_s\ \tilde{D}_{\not<}\ X_s)$ THEN
            RETURN any element of X_s
        ELSE
            Replace X_s in L with *Split*(X_s).
            `Dominated` $\leftarrow \{X_i \in L \mid \exists X_j \in L\ (X_i\ \tilde{D}_<\ X_j)\}$
            $L \leftarrow L -$ `Dominated`
    END WHILE
    RETURN `failure`
END *BranchAndBound-2*.

Figure 4.12: Branch-and-Bound where $<_X$ is a Partial Order.

Given appropriate arguments, *BranchAndBound-2* can find a single element of the deductive closure of $\langle C, P\rangle$, which is sufficient to implement the function *Enumerate*$(\langle C, P\rangle, \Sigma^*, 1)$. The arguments passed to *BranchAndBound-2* in order to implement *Enumerate*$(\langle C, P\rangle, \Sigma^*, 1)$ are shown in Figure 4.13, and described in detail below.

The arguments to *BranchAndBound-2* consist of X, and the dominance relations $\tilde{D}_<$ and $\tilde{D}_{\not<}$ over subsets of X, as derived from $<_X$. The *Split* and *Select* functions are also parameters to *BranchAndBound-2*, albeit implicit ones. The algorithm *BranchAndBound-2* finds a single element of the set $\{x \in X \mid \forall y \in X\ (x \not<_X y)\}$. In order to implement *Enumerate*$(\langle C, P\rangle, \Sigma^*, 1)$, the arguments must be set so that *BranchAndBound-2* finds a single element of the deductive closure of $\langle C, P\rangle$—namely, $\{x \in C \mid \forall y \in C\ (x \not<_P y)\}$.

$Enumerate(\langle C, P\rangle, \Sigma^*, \mathbf{1}) = BranchAndBound\text{-}2(C, \tilde{D}'_<, \tilde{D}_\not<)$ where

- A subset of C is denoted $\langle w, q\rangle$ where $w \in \Sigma^*$ and $q = \delta_C(s_C, w)$.
  C itself is $\langle \epsilon, s_C\rangle$, where ϵ is the empty string and s_C is the start state of C.

- $\langle w_1, q_1\rangle \tilde{D}_\not< \langle w_2, q_2\rangle$ iff $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P = \emptyset$ where

  $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P = \emptyset$ if
  $Dead_C(q_1) = \texttt{true}$ or $Dead_C(q_2) = \texttt{true}$ or
  $(q = delta^*_P(s_P, shuffle(w_1, w_2))$ and
  $AcceptAll_P(q) = \texttt{true})$

- $\langle w_1, q_1\rangle \tilde{D}_< \langle w_2, q_2\rangle$ iff $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$ and $\langle w_2, q_2\rangle \neq \emptyset$ where

  $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$ if
  $q = \delta_P(s_P, shuffle(w_1, w_2))$ and $AcceptAll_P(q) = \texttt{true}$

  $\langle w_2, q_2\rangle \neq \emptyset$ if $Dead_C(q_2) = \texttt{true}$

- $Split(\langle w, q\rangle) = \{\langle w \cdot \sigma, q'\rangle \mid \sigma \in \Sigma \text{ and } q' = \delta_C(q, \sigma)\}$

- $Select(L)$ returns $X_s \in L$ s.t. $(\forall X_i \in L)\langle X_s, X_i\rangle \notin D'_<$ where

  $\langle w_1, q_1\rangle \tilde{D}'_< \langle w_2, q_2\rangle$ iff
  $q = \delta_P(s_P, shuffle(w_1, w_2))$ and $AcceptAll_P(q) = \texttt{true}$

Figure 4.13: Parameters to *BranchAndBound-2*.

Specifying Subsets. The set of elements, X, is set to C. The relations $\tilde{D}_<$ and $\tilde{D}_\not<$, and the *Split* function all depend on the way in which subsets of X are represented. There are several ways to partition C into subsets, but one that seems to work well is to specify a subset by the tuple $\langle w, q\rangle$, where w is a string and q is the state in C reached on input w from the start state. This subset contains all strings in C of the form ww', where w' is a string leading from q to an accept state. Since C is a DFA, there is at most one state, q, reached by a given input, so $\langle w, q\rangle$ contains all strings in C that begin with w.

The notation $\langle w, q\rangle$ is in fact shorthand for the concatenation of two DFAs, one that recognizes w, and one that recognizes all strings w' such that $\delta^*(q, w')$ reaches

an accept state in C. The latter DFA can be derived from the DFA for C by simply making q the start state. Because of this simple derivation, all of the subsets of C can share the DFA for C, so new DFAs do not need to be created for each subset.

Subset $\langle w, q\rangle$ is a singleton if and only if q is an accept state and every path from q leads to a dead state. That is, w is in C, but no extension of w is in C. Subset $\langle w, q\rangle$ is empty if and only if q is a dead state in C, since no extension of w is recognized by C.

The Split Function. The *Split* function takes a subset of C, $\langle w, q\rangle$, and partitions it into subsets by extending w with each of the symbols in the alphabet, Σ:

$$Split(\langle w, q\rangle) = \{\langle w \cdot \sigma, q'\rangle \mid q' = \delta_C(q, \sigma)\}$$

Alternately, w could be extended by strings instead of by individual symbols. One possibility is to extend w by a string w' such that ww' is a hypothesis in H, though perhaps not one recognized by C. In this case a subset $\langle w, q\rangle$ contains all hypotheses that are extensions of hypothesis w, and also recognized by C. This version of the split function would be defined as follows:

$$Split(\langle w, q\rangle) = \{\langle w \cdot w', q'\rangle \mid q' = \delta^*_C(q, w) \text{ and } \delta^*_H(s, ww') \in F_H\}$$

The Dominance Relation. The dominance relations, $\tilde{D}_<$ and $\tilde{D}_{\not<}$ are incomplete versions of $D_<$ that are defined only on subsets for which the dominance relation can be determined inexpensively. The complete dominance relation over subsets of the form $\langle w, q\rangle$ will be defined first, followed by inexpensive definitions for $\tilde{D}_<$ and $\tilde{D}_{\not<}$.

A subset $\langle w_1, q_1\rangle$ is dominated by subset $\langle w_2, q_2\rangle$ if and only if $(\forall h \in \langle w_1, q_1\rangle)(\exists h' \in \langle w_2, q_2\rangle)\, \langle h, h'\rangle \in P$. That is, every hypothesis in $\langle w_1, q_1\rangle$ is less preferred than some hypothesis in $\langle w_2, q_2\rangle$, according to P.

This relation can be expressed in terms of set operations on P and the two subsets as follows. The set $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P$ is $\{\langle x, y\rangle \mid x < y \text{ and } x \in \langle w_1, q_1\rangle \text{ and } y \in \langle w_2, q_2\rangle\}$. Projecting this set onto its first elements yields the set of hypotheses in $\langle w_1, q_1\rangle$ that are dominated by one or more elements of $\langle w_2, q_2\rangle$. If the projection is exactly $\langle w_1, q_1\rangle$, then every hypothesis in $\langle w_1, q_1\rangle$ is dominated by elements of

$\langle w_2, q_2\rangle$, so $\langle w_1, q_1\rangle\ D_<\ \langle w_2, q_2\rangle$. Otherwise, some elements of $\langle w_2, q_2\rangle$ are not dominated, so $\langle w_1, q_1\rangle\ \not D_<\ \langle w_2, q_2\rangle$. The relation $D_<$ is therefore defined as shown in Equation 4.3.

$$\langle w_1, q_1\rangle\ D_<\ \langle w_2, q_2\rangle \text{ iff } \mathit{first}((\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P) = \langle w_1, q_1\rangle \tag{4.3}$$

Approximating the Dominance Relation. This test is expensive to compute. A less expensive, though possibly overspecific test is needed to define $\tilde{D}_<$. As a first approximation, a sufficient but not necessary condition for $\langle w_1, q_1\rangle\ D_<\ \langle w_2, q_2\rangle$ is $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$. When this is true, every hypothesis in $\langle w_1, q_1\rangle$ is dominated by every hypothesis in $\langle w_2, q_2\rangle$. To this condition a test must be added to ensure that $\langle w_2, q_2\rangle$ is not empty, since $D_<$ does not allow a subset to be dominated by an empty set. The complete test is summarized in Equation 4.4.

$$\langle w_1, q_1\rangle \tilde{D}_< \langle w_2, q_2\rangle \quad \text{iff} \quad (\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle)) \subseteq P \text{ and } \langle w_2, q_2\rangle \neq \emptyset \tag{4.4}$$

This definition is a step in the right direction, but it is still too expensive to compute. The second condition of this test determines whether $\langle w_2, q_2\rangle$ is empty. This is equivalent to testing whether q_2 is a dead state, which can be expensive if this information is not already known by the DFA. The first condition is also expensive. Specifically, it determines whether $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle)$ is a subset of P, which is equivalent to testing whether $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap \overline{P}$ is empty. This is also an emptiness test, and therefore potentially expensive.

The two tests of Equation 4.4 are expensive to compute in general, but for some subsets they can be computed cheaply. A sufficient condition for the first test, $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$, is $\delta^*_P(s, \mathit{shuffle}(w_1, w_2)) = q$ and $\mathit{AcceptAll}_P(q) = \texttt{true}$. That is, the string $\mathit{shuffle}(w_1, w_2)$ leads to an accept-all state in P, which means that every pair of strings in $\{\langle w_1 x, w_2 y\rangle \mid x \in \Sigma^* \text{ and } y \in \Sigma^*\}$ is a member of P. This certainly includes all the strings in $\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle$, so $\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle$ is a subset of P.

Computing $q = \delta_P(\mathit{start}_P, \mathit{shuffle}(w_1, w_2))$ takes time proportional to the lengths of w_1 and w_2. Determining whether q is an accept-all state is accomplished by the *AcceptAll* function, which only returns `true` or `false` when the answer can be

computed cheaply, and returns **unknown** when it cannot. This test can only compare subsets for which *AcceptAll* does not return **unknown**.

The second test in Equation 4.4 determines whether $\langle w_2, q_2 \rangle$ is empty—i.e., whether q_2 is a dead state. If it can be determined cheaply whether q_2 is a dead state, then the function $Dead_C(q_2)$ returns `true` or `false`, according to whether or not the state is dead. Otherwise, the function returns **unknown**. This leads to the definition for $\tilde{D}_<$ shown in Equation 4.6.

$$\begin{aligned} &\langle w_1, q_1 \rangle \tilde{D}_< \langle w_2, q_2 \rangle \text{ iff} \\ &\quad AcceptAll_P(\delta_P(start_P, shuffle(w_1, w_2))) = \texttt{true} \text{ and} && (4.5) \\ &\quad Dead_C(q_2) = \texttt{false} && (4.6) \end{aligned}$$

Defining the Not-Dominated Relation. The definition of $\tilde{D}_{\not<}$ is derived similarly. The complete and correct test for determining whether one subset is not dominated by another is shown in Equation 4.7. This is the complement of the dominance relation, $D_<$.

$$\langle w_1, q_1 \rangle \not\!\!D_< \langle w_2, q_2 \rangle \quad \text{iff} \quad first(\langle w_1, q_1 \rangle \times \langle w_2, q_2 \rangle) \cap P \neq \langle w_1, q_1 \rangle \tag{4.7}$$

This relation is expensive to determine. It must be replaced by a less expensive, though possibly incomplete relation. A sufficient condition for this relation is $(\langle w_1, q_1 \rangle \times \langle w_2, q_2 \rangle) \cap P = \emptyset$. When this condition is true, no hypothesis in $\langle w_1, q_1 \rangle$ is dominated by any hypothesis in $\langle w_2, q_2 \rangle$. The approximation of the not-dominated relation based on this test is shown in Equation 4.8.

$$\langle w_1, q_1 \rangle \not\!\!D_< \langle w_2, q_2 \rangle \quad \text{if} \quad (\langle w_1, q_1 \rangle \times \langle w_2, q_2 \rangle) \cap P = \emptyset \tag{4.8}$$

This test is still too expensive to compute, since it involves testing whether the intersection of two sets is empty. However, a sufficient condition for this test can be computed inexpensively. Namely, $(\langle w_1, q_1 \rangle \times \langle w_2, q_2 \rangle) \cap P = \emptyset$ if the string $shuffle(w_1, w_2)$ leads to a dead state in P, or if either q_1 or q_2 are dead states in C. The *Dead* functions for C and P return **unknown** if it is expensive to determine whether a given state is dead, and otherwise return `true` if it is dead, and `false` if

it is not. Fortunately, P is a union of several DFAs, so if the dead state information is known for all of the states in these DFAs, then the *Dead* function for P never returns **unknown** (see Section 4.2.1.2). The definition for $\tilde{D}_{\not<}$ is given formally in Equation 4.10.

$$\langle w_1, q_1\rangle \; \tilde{D}_{\not<} \; \langle w_2, q_2\rangle \text{ iff}$$

$$Dead_C(q_1) = \texttt{true} \text{ or } Dead_C(q_2) = \texttt{true} \text{ or} \tag{4.9}$$

$$Dead_P(\delta_P^*(start_P, shuffle(w_1, w_2))) = \texttt{true} \tag{4.10}$$

The Selection Criteria. Because of the way *Split* and $\tilde{D}_<$ are defined, it is possible for a subset, $\langle w, q\rangle$, in the collection to be empty. Subset $\langle w, q\rangle$ is empty if q is a dead state. This is expensive to determine, so empty subsets are not eliminated automatically by the *Split* function, nor are they identified as dominated subsets by $\tilde{D}_<$. Eventually, $\langle w, q\rangle$ will be split into child subsets, $\langle ww_1, q_1\rangle$ through $\langle ww_k, q_k\rangle$, whose emptiness *can* be cheaply determined. In the worst case, q_1 through q_k comprise all of the states reachable from q. At least one of these is a known dead state, and this information can then be propagated to the other states. However, the cost of splitting $\langle w, q\rangle$ to this point can be quite expensive since it is proportional to the number of states that are reachable from q.

This expense can be mitigated by being intelligent about which subset is selected in each iteration. Given a choice between $\langle w_1, q_1\rangle$ and $\langle w_2, q_2\rangle$, choose $\langle w_2, q_2\rangle$ if $shuffle(w_1, w_2)$ leads to an accept-all state in P, since this means that every string in $\langle w_2, q_2\rangle$ is preferred to every string in $\langle w_1, q_1\rangle$. The subset $\langle w_1, q_1\rangle$ could be pruned as dominated at this point, except that is not known whether $\langle w_2, q_2\rangle$ is empty. Recall that the domination relation $\tilde{D}_<$ does not allow a subset to be dominated by an empty subset. If the same selection criteria is applied consistently, the child subsets of $\langle w_2, q_2\rangle$ will also be selected over the subset $\langle w_1, q_1\rangle$. Eventually, $\langle w_2, q_2\rangle$ will be split into enough child subsets that it can either be determined that all of the children are empty, or that one of them is non-empty. In the former case, the empty children are pruned from the collection, since all empty sets are dominated, which frees up the selection criteria to select $\langle w_1, q_1\rangle$. In the latter case, there is at least one non-empty child subset of $\langle w_2, q_2\rangle$. Call this subset $\langle w^*, q^*\rangle$. Since it is known that this subset is not empty, and it is also known that every extension of w^*

is preferred to every extension of w_1, the subset $\langle w_1, q_1 \rangle$ is dominated by $\langle w^*, q^* \rangle$, and can be pruned from the collection.

Whether or not the selected subset eventually turns out to be empty, the selection criteria described above is often cheaper and never worse than the inverse policy. If $\langle w_1, q_1 \rangle$ and its children were preferred over $\langle w_2, q_2 \rangle$ in the above example, then the search cost would be higher. If $\langle w_1, q_1 \rangle$ is either empty or non-empty, and $\langle w_2, q_2 \rangle$ is non-empty, then the child subsets of $\langle w_1, q_1 \rangle$ are searched, followed by the children of $\langle w_2, q_2 \rangle$. In the above example, only the children of $\langle w_2, q_2 \rangle$ would be searched. If $\langle w_1, q_1 \rangle$ were non-empty, and $\langle w_2, q_2 \rangle$ were empty, then the children of both subsets must be searched using either selection criteria. The costs are the same in this case.

Subsets are therefore selected from the collection according to the first policy, since it can reduce the amount of search required. The selection policy prefers subset $\langle w_2, q_2 \rangle$ over $\langle w_1, q_1 \rangle$ when $\mathit{shuffle}(w_1, w_2)$ leads to an accept state in P. This is the same as the definition of $\tilde{D}_<$, except that the emptiness test for $\langle w_2, q_2 \rangle$ is omitted. Call this new relation $\tilde{D}'_<$. The subsets in the collection are partially ordered according to $\tilde{D}'_<$, and one of the subsets at the top of the order is selected. The relation $\tilde{D}'_<$ is defined formally in Equation 4.11, and the selection criterion is defined in Equation 4.12.

$$\langle w_1, q_1 \rangle \; \tilde{D}'_< \; \langle w_2, q_2 \rangle \text{ iff}$$
$$q = \delta_P(s_P, \mathit{shuffle}(w_1, w_2)) \text{ and } \mathit{AcceptAll}_P(q) = \texttt{true} \tag{4.11}$$
$$\mathit{Select}(L) \text{ returns } X_s \in L \text{ s.t. } (\forall X_i \in L)\ \langle X_s, X_i \rangle \notin \tilde{D}'_< \tag{4.12}$$

4.2.2.3 Fully Modified Branch-and-Bound Algorithm

The full implementation of $\mathit{Enumerate}(\langle C, P \rangle, A, n)$ must be able to enumerate up to n hypotheses that are both in the deductive closure of $\langle C, P \rangle$ and in the set A. This is accomplished with a few simple modifications, as shown in Figure 4.14. If the selected subset, X_s, contains only undominated hypotheses then instead of returning a single element of X_s as is done in *BranchAndBound-2*, X_s is intersected with A, and elements are enumerated from $X_s \cap A$ until all n hypotheses have been enumerated or there are no more hypotheses in the intersection. X_s is marked as

enumerated, and remains in the collection. If additional hypotheses are needed, the search continues until another undominated subset is found, and its elements are enumerated as mentioned above. When all n hypotheses have been enumerated, or all of the subsets in the collection have been marked, the search halts and the list of hypotheses enumerated so far is returned. This list contains no more than n hypotheses, but may contain fewer if the intersection of A and the undominated elements of X contains fewer than n hypotheses. The algorithm incorporating these modifications is called *BranchAndBound-3* and is shown in Figure 4.14.

BranchAndBound-3$(X, \tilde{D}_<, \tilde{D}_{\not<}, n, A)$ returns n elements of $\{x \in X \mid \forall_{y \in X}\ x \not< y\} \cap A$
    X is a set of hypotheses
    $\tilde{D}_<$ is a domination relation on subsets of X derived from $<$
    $\tilde{D}_{\not<}$ is a not-dominated relation on subsets of X derived from $<$
    n is the number of solutions requested
    $<$ is a partial ordering over the elements of X
BEGIN
    $L \leftarrow \{X\}$
    `Solutions` $\leftarrow \{\}$
    `Enumerated` $\leftarrow \{\}$
    WHILE |`Solutions`| $< n$ and $L -$ `Enumerated` is not empty DO
        $X_s \leftarrow$ *select*($L -$ `Enumerated`)
        IF $(X_s\ \tilde{D}_{\not<}\ X_i)$ for every X_i in L THEN
            Add hypotheses from $X_s \cap L$ to `Solutions` until |`Solutions`| $= n$ or there are no more hypotheses in $X_s \cap L$
        ELSE
            Replace X_s in L with *Split*(X_s)
            `Dominated` $\leftarrow \{X_i \in L \mid \exists X_j \in L\ (X_i\ \tilde{D}_<\ X_j)\}$
            $L \leftarrow L -$ `Dominated`
        END IF
    END WHILE
    RETURN `Solutions`
END *BranchAndBound-3*.

Figure 4.14: Branch-and-Bound That Returns n Solutions Also in A.

RS-KII implements *Enumerate*$(\langle C, P\rangle, A, n)$ by calling *BranchAndBound-3* with the arguments shown in Figure 4.15. These are the same arguments passed to

BranchAndBound-2 in order to implement $Enumerate(\langle C, P\rangle, \Sigma^*, 1)$, with the addition of the n and A arguments.

$Enumerate(\langle C, P\rangle, A, n) = BranchAndBound\text{-}3(C, \tilde{D}_<, \tilde{D}_{\not<}, A, n)$ where

- A subset of C is denoted $\langle w, q\rangle$ where $w \in \Sigma^*$ and $q = \delta_C(s_C, w)$.
- $\langle w_1, q_1\rangle \; \tilde{D}_{\not<} \; \langle w_2, q_2\rangle$ iff $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P = \emptyset$ where

  $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \cap P = \emptyset$ iff
  $Dead_C(q_1) = \texttt{true}$ or $Dead_C(q_2) = \texttt{true}$ or
  $(q = delta^*_P(s_P, shuffle(w_1, w_2))$ and $AcceptAll_P(q) = \texttt{true})$

- $\langle w_1, q_1\rangle \; \tilde{D}_< \; \langle w_2, q_2\rangle$ iff $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$ where

  $(\langle w_1, q_1\rangle \times \langle w_2, q_2\rangle) \subseteq P$ iff
  $q = \delta_P(s_P, shuffle(w_1, w_2))$ and $AcceptAll_P(q) = \texttt{true}$

- $Split(\langle w, q\rangle) = \{\langle w \cdot \sigma, q'\rangle \mid \sigma \in \Sigma \text{ and } q' = \delta_C(q, \sigma)\}$
- $Select(L)$ returns $X_s \in L$ s.t. $(\forall X_i \in L)\langle X_s, X_i\rangle \notin \tilde{D}'_<$ where

  $\langle w_1, q_1\rangle \; \tilde{D}'_< \; \langle w_2, q_2\rangle$ iff
  $q = \delta_P(s_P, shuffle(w_1, w_2))$ and $AcceptAll_P(q) = \texttt{true}$

Figure 4.15: Parameters to *BranchAndBound-3* for Implementing *Enumerate*.

4.2.2.4 Efficient Selection and Pruning

The collection L must be represented intelligently in order to minimize the complexity of selecting subsets and pruning dominated subsets. In RS-KII, L is maintained as a lattice representing the known dominance relations among the subsets in L. Each node in L is a subset, and an edge from X_i to X_j indicates that $X_j \; \tilde{D}'_< \; X_i$ is true. Selecting a hypothesis is a matter of selecting an element from the top of the lattice. The nodes at the top are maintained as a list, with each top-node pointing to the next top-node in the list.

A subset X_i is dominated if there is some subset X_j in the collection such that $X_i \; \tilde{D}'_< \; X_j$ and X_j is not empty. Thus, if the selected subset is discovered to be

non-empty, every subset below it in the lattice is pruned. If the selected subset is discovered to be empty, then it is removed from the top of the lattice. The subsets it used to dominated are promoted to the top of the lattice if they are not dominated by other top-nodes.

When the selected subset, X_s, is split, it is removed from the lattice and replaced with its children. If some subset X_i is below X_s in the lattice, then X_i is below each child of X_s as well. This is because $X_i \; \tilde{D}'_< \; S$ is true if and only if every hypothesis in X_i is less preferred than every hypothesis in X_s. Thus, the relation holds for every subset of X_s. However, $\tilde{D}'_<$ is not known for every pair of subsets. Thus, $\tilde{D}'_<$ may be known between some subset X_i and a child of X_s where it was not known between X_i and X_s. The children of X_s are tested against the other top-nodes, and edges are added between the children and these nodes accordingly.

Chapter 5

RS-KII and AQ11

AQ-11 [Michalski, 1978] is an induction algorithm that induces a hypothesis in the VL_1 hypothesis space language [Michalski, 1974] from examples and a user-defined evaluation function. RS-KII can emulate the AQ-11 algorithm by utilizing this knowledge and the biases implicit in AQ-11. When RS-KII utilizes only this knowledge, the computational complexity of RS-KII is a little worse than that of AQ-11.

RS-KII can also utilize knowledge that AQ-11 cannot, such as a domain theory. By using this knowledge in conjunction with the AQ-11 knowledge sources described above, RS-KII effectively integrates this knowledge into AQ-11, forming a hybrid induction algorithm.

RS-KII's ability to emulate AQ-11 and to integrate additional knowledge are demonstrated in this chapter. The AQ-11 algorithm is described in Section 5.1. This is followed in Section 5.2 by a description of the knowledge used by AQ11, and implementations of RS-KII translators for that knowledge. This section also describes RS-KII translators for domain theories, which AQ-11 cannot use.

In Section 5.3, RS-KII uses these translators to solve a synthetic induction task. The knowledge made available by this task includes the knowledge used by AQ-11, plus a domain theory. It is demonstrated that when RS-KII uses only the knowledge used by AQ-11, both RS-KII and AQ-11 induce the same hypothesis. When RS-KII uses the domain theory in addition to the AQ-11 knowledge, RS-KII induces a more accurate hypothesis than the one induced by AQ-11.

In Section 5.4, the computational complexity of RS-KII is analyzed and compared to that of AQ-11. When using only the knowledge available to AQ-11, RS-KII's complexity is a little worse than that of AQ-11.

5.1 AQ-11 Algorithm

The AQ-11 induction algorithm [Michalski, 1978, Michalski and Chilausky, 1980] induces a hypothesis from a list of noise-free positive and negative examples, and a user-defined evaluation function that Michalski calls a "lexicographic evaluation function," or LEF [Michalski, 1978].

Since the examples are assumed to be noise-free, AQ-11 induces a hypothesis that is strictly consistent with the examples. That is, the induced hypothesis covers all of the positive examples and none of the negative examples. There may be several hypotheses consistent with the examples, in which case the LEF is used to select among them. Ideally, the selected hypothesis is one that is most preferred by the LEF. However, finding a global maximum of the LEF is intractable, so AQ-11 settles for a local maximum. A locally optimal hypothesis is found by a beam search, which is guided by the LEF.

The hypothesis space for AQ-11 is one described by the VL_1 hypothesis space language. This language is described in Section 5.1.1. The instance space, from which the examples are selected, is described in Section 5.1.2. The algorithm itself in described Section 5.1.3.

5.1.1 The Hypothesis Space

The hypothesis space consists of sentences in the VL_1 concept description language [Michalski, 1974]. This is actually a family of languages, parameterized by a list of features and feature-values. Specific hypothesis space languages are obtained by instantiating VL_1 with these parameters.

Figure 5.1 describes a regular grammar for the VL_1 family of languages. This grammar is parameterized by a list of $\langle f_i, V_i \rangle$ pairs, where f_i is a feature name, and V_i is the set of values that the feature f_i can take. The set of values is expressed as a regular grammar, which simplifies the specification of the VL_{11} grammar. Any finite or countably infinite set of values can be expressed as a regular grammar by mapping each value onto a binary integer in $1(0|1)^*$ (i.e., the set of binary numbers greater than zero).

Parameters: $\langle f_1, V_1\rangle, \langle f_2, V_2\rangle, \ldots, \langle f_k, V_k\rangle$

VL1 $\rightarrow$ TERM | TERM or VL1
TERM $\rightarrow$ SELECTOR | SELECTOR TERM
SELECTOR $\rightarrow$
"[" $f_1 \rightarrow$ $(< \mid \leq \mid = \mid \neq \mid \geq \mid >)$ V_1 "]" |
"[" $f_2 \rightarrow$ $(< \mid \leq \mid = \mid \neq \mid \geq \mid >)$ V_2 "]" |
$\vdots$
"[" $f_k \rightarrow$ $(< \mid \leq \mid = \mid \neq \mid \geq \mid >)$ V_k "]"

Figure 5.1: The VL_1 Concept Description Language.

Sentences (hypotheses) in VL_1 languages are disjunctive normal form (DNF) expressions of *selectors*. A sentence in this language is a disjunct of terms, and a term is a conjunct of selectors. A selector is an expression of the form $[f_i \ \# \ v]$, where f_i is a feature, $\#$ is a relation from the set $\{=, \neq, <, \leq, \geq, >\}$, and v is one of the values that feature f_i can take. For example, the following is a hypothesis in one possible VL_1 language:

`[color = red][size` $>$ `20] or [size` $\leq$ `5].`

5.1.2 Instance Space

An instance in AQ-11 is an ordered tuple, $\langle v_1, v_2, \ldots, v_k\rangle$, in which v_i is the value of feature f_i for that instance. For example, if there are two features, `size` and `color`, then instances would be tuples such as ⟨50, red⟩ and ⟨25, green⟩.

Examples are classified instances. An example is *positive* if it is *covered* by the target concept, and *negative* if it is not. An instance is covered by a VL_1 hypothesis if it satisfies the hypothesis. That is, the hypothesis must contain at least one term in which every selector is satisfied by the instance. A selector $[f_i \ \# \ x_i]$ is satisfied by instance $\langle v_1, v_2, \ldots, v_k\rangle$ if $v_i \ \# \ x_i$. Recall that "#" is a relation from $\{<, \leq, =, \neq, \geq, >\}$.

For example, the hypothesis `[color = red][size` $>$ `20] or [size` $\leq$ `5]` covers the instance ⟨50, red⟩ since red = red and $50 > 20$. The hypothesis does not cover ⟨25, green⟩, since the instance is satisfied by neither term.

5.1.3 The Search Algorithm

The AQ-11 algorithm searches the hypothesis space for a hypothesis that is consistent with all of the examples, and locally maximizes the LEF. The algorithm is shown in two parts in Figure 5.2 and Figure 5.3.

AQ-11 is essentially a separate-and-conquer [Pagallo and Haussler, 1990] algorithm. The positive examples are placed in a "pot." A term is found that covers at least one of the positive examples in the pot and none of the negative examples. The positive examples covered by the term are removed from the pot, and the process repeats on the remaining examples in the pot. Eventually, every positive example is covered by at least one of the terms, and no negative example is covered by any of the terms. The terms are disjoined, and returned as the induced hypothesis. This main loop of the algorithm is shown in Figure 5.2.

The second part of the algorithm is an inner loop that searches the space of terms for a term that covers a positive example called the *seed*, but none of the negative examples. The seed is selected in each iteration of the main loop from the pot of uncovered positive examples. There are many terms that satisfy these criteria. The term returned to the main loop by the search is ideally one that maximizes the LEF. In practice, the space of terms is so huge that searching it for the best term would be intractable. Instead, a beam search with width b is used to find a locally optimal term. The width of the beam is determined by the user.

The beam initially contains the term `true`. On each iteration, each term in the beam is replaced with its children. Child terms are generated from the parent term by conjoining a selector to the end of the parent. This selector must cover the positive seed, and fail to cover at least one new negative example that was covered by the parent. This ensures progress towards a term that covers the seed but none of the negative examples. The best b terms in the beam are retained, as determined by the LEF. When every term in the beam covers the seed but none of the negative examples, the best term in the beam is returned to the main loop. The beam search is shown in Figure 5.3.

```
Algorithm AQ-11(Pos, Neg, LEF, b) returns h
    PosList is a list of positive examples
    NegList is a list of negative examples
    LEF is a term evaluation function
    b is the beam width
BEGIN
    h = false
    LOOP
        Select seed from PosList
        term = LearnTerm(seed, NegList, LEF, b)
        h = h ∨ term
        Remove from PosList every example covered by Term
    UNTIL PosList is empty
    RETURN h
END AQ-11
```

Figure 5.2: The Outer Loop of the AQ-11 Algorithm.

5.2 Knowledge Sources and Translators

The knowledge used by AQ-11 [Michalski, 1978] consists of positive and negative examples, a user-defined lexicographic evaluation function (LEF), a list of features, and a set of values for each feature. The features and values parameterize the VL_1 concept description language, and the remaining knowledge drives the search.

In order for RS-KII to use this knowledge, it must be expressed in terms of constraints and preferences. This is accomplished by translators, which are described below. In addition to the knowledge used by AQ-11, RS-KII can utilize any other knowledge for which a translator can be constructed. A translator for one such knowledge source—a domain theory—is described below. Translators for other knowledge sources are also possible.

5.2.1 Examples

AQ-11 induces hypotheses that are strictly consistent with the examples. That is, the induced hypothesis must cover all of the positive examples and none of the

```
Algorithm LearnTerm(seed, NegList, LEF, b) returns term
    seed is a positive example
    NegList is a list of negative examples
    LEF is a term evaluation function
    b is the beam width
BEGIN
    Terms = {true}
    LOOP
    (1)  Select neg-seed from NegList
    (2)  S = set of selectors covering seed but not neg-seed.
    (3)  Children = {t · selector | t ∈ Terms and selector ∈ S}
    (4)  Terms = best b terms in Children according to LEF
    (5)  Remove from NegList examples covered by none of the terms in Terms
    UNTIL NegList is empty
    RETURN the best element of Terms according to LEF
END LearnTerm
```

Figure 5.3: The inner Loop of the AQ11 Algorithm.

negative examples. A positive example is translated into a constraint satisfied only by hypotheses that cover the example. A negative example is translated into a constraint satisfied only by hypotheses that do not cover the example.

The implementation of this example translator is shown in Figure 5.4. It takes as input a hypothesis space and an example. The hypothesis space is an instantiation of VL_1 with appropriate features and values. The function $Covers(H, i)$ returns the set of hypotheses in H that cover instance i, and the function $Excludes(H, i)$ returns the set of hypotheses in H that do not cover instance i. Both sets are represented as regular expressions (i.e., regular sets). The regular expression returned by $Covers(H, i)$ is shown in Figure 5.5. $Excludes(H, i)$ returns the complement of $Covers(H, i)$.

The regular grammar returned by $Covers(H, i)$ recognizes all hypotheses in H that are satisfied by instance i. H is an instantiation of VL_1, and i is an ordered vector of feature values, $\langle v_1, v_2, \ldots, v_k \rangle$. A hypothesis covers an instance if the hypothesis contains at least one term in which every selector is satisfied by the instance. In general, selector $[f_i \# v]$ covers $\langle v_1, v_2, \ldots, v_k \rangle$ if $v_i \# v$, where "#" is

$$TranAQExample(\, VL_1(\langle f_1, V_1\rangle, \langle f_2, V_2\rangle, \ldots, \langle f_k, V_k\rangle),$$
$$\langle\langle v_1, v_2, \ldots, v_k\rangle, class\rangle) \rightarrow \langle C, \{\}\rangle$$

where

$$C = \begin{cases} Covers(\; VL_1(\langle f_1, V_1\rangle, \langle f_2, V_2\rangle, \ldots, \langle f_k, V_k\rangle), \\ \quad \langle v_1, v_2, \ldots, v_k\rangle) & \text{if } class = \texttt{positive} \\ Excludes(\; VL_1(\langle f_1, V_1\rangle, \langle f_2, V_2\rangle, \ldots, \langle f_k, V_k\rangle), \\ \quad \langle v_1, v_2, \ldots, v_k\rangle) & \text{if } class = \texttt{negative} \end{cases}$$

Figure 5.4: Example Translator for VL_1.

a relation in $\{<, \leq, =, \neq, \geq, >\}$. For instance, the selector $[f_3 < 12]$ covers instance $\langle red, 18, 9\rangle$ since $9 < 12$.

Figure 5.5 shows the regular expression returned by $Covers(H, i)$. At the top level, the expression is (ANY-TERM or)* COVERING-TERM (or ANY-TERM)*. This says that at least one term in the hypothesis must cover the instance, namely the term recognized by COVERING-TERM.

A term covers the instance if every selector in it covers the instance. COVERING-TERM expands to COVERING-SELECTOR$^+$, which is a conjunction of covering selectors. COVERING-SELECTOR is the set of selectors that are satisfied by the instance. The instance is an ordered vector of feature values, $\langle v_1, v_2, \ldots, v_k\rangle$. A selector covers $\langle v_1, v_2, \ldots, v_k\rangle$ if it is of the form $[f_i \ \# \ v]$, where $v_i \ \# \ v$ and $\#$ is a relation in $\{<, \leq, =, \neq, \geq, >\}$. This set of selectors is recognized by the regular expression COVERING-SELECTOR.

The definition of COVERING-SELECTOR is a little obscure. An instance is covered by a selector of the form $[f_i < v]$ if the value of the instance on feature f_i is less than v. That is, the instance is covered by all selectors in the set $\{[f_i < v] \mid v > v_i\}$, where v_i is the value of the instance on feature f_i. A similar analysis holds for the other relations, $\leq, =, \neq, \geq$, and $>$. COVERING-SELECTOR expands into a union of regular expressions for each of these sets. Each of these sets can be written as regular expressions. For example, the regular expression for $\{[f_2 < v] \mid v > 10\}$ is $[f_2 < (1-9)(0-9)^+]$.

The function $Excludes(H, i)$ returns the complement of the grammar returned by $Covers(H, i)$.

$Covers(VL_1(\langle f_1, V_1\rangle, \langle f_2, V_2\rangle, \ldots, \langle f_k, V_k\rangle), \langle v_1, v_2, \ldots, v_k\rangle) \rightarrow G$ where

G = (ANY-TERM or)* COVERING-TERM (or ANY-TERM)*

ANY-TERM = SELECTOR$^+$

COVERING-TERM = COVERING-SELECTOR$^+$

SELECTOR = "[" f_1 ($<$ | $\leq$ | $=$ | $\neq$ | $\geq$ | $>$) V_1 "]" |
"[" f_2 ($<$ | $\leq$ | $=$ | $\neq$ | $\geq$ | $>$) V_2 "]" |
$\vdots$
"[" f_k ($<$ | $\leq$ | $=$ | $\neq$ | $\geq$ | $>$) V_k "]"

COVERING-SELECTOR = $\{[f_i \# v] \mid v_i \# v$ and $\# \in \{<, \leq, =, \neq, \geq, >\}\}$ =

"[" $f_1 < \{v \in V_1 \mid f_1 \geq v_1\}$ "]"	...	"[" $f_k < \{v \in V_k \mid f_k \geq v_k\}$ "]"
"[" $f_1 \leq \{v \in V_1 \mid f_1 > v_1\}$ "]"	...	"[" $f_k \leq \{v \in V_k \mid f_k > v_k\}$ "]"
"[" $f_1 = v_1$ "]"	...	"[" $f_k = v_k$ "]"
"[" $f_1 \neq (V_1 - \{v_1\})$ "]"	...	"[" $f_k \neq (V_k - \{v_k\})$ "]"
"[" $f_1 \geq \{v \in V_1 \mid f_1 < v_1\}$ "]"	...	"[" $f_k \geq \{v \in V_k \mid f_k < v_k\}$ "]"
"[" $f_1 > \{v \in V_1 \mid f_1 \leq v_1\}$ "]"	...	"[" $f_k > \{v \in V_k \mid f_1 \leq v_k\}$ "]"

Figure 5.5: Regular Expression for the Set of VL_1 Hypotheses Covering an Instance.

5.2.2 Lexicographic Evaluation Function

AQ-11 searches the hypothesis space in a particular order, and returns the first hypothesis in the order that is consistent with the examples. The search order therefore expresses a preference for hypotheses that are visited earlier in the search to those that are visited later. The search order is partly determined by the search algorithm, and partly determined by the lexicographic evaluation function (LEF), which guides the search. The LEF and the search algorithm are translated conjointly into a preference for hypotheses that occur earlier in the search.

Expressing this preference as a regular grammar can be somewhat difficult. Given two hypotheses, h_1 and h_2, determining which of them is visited first by the search is often difficult to extract just from the information available in h_1 and h_2. Often, the only way to determine their relative ordering is to run the search and see which one is visited first. This is clearly impractical. However, for some searches, the ordering can be determined directly from h_1 and h_2. For example, in best first search, h_1 is generally visited before h_2 if h_1 is preferred by the evaluation function, although the topology of the search tree also affects the visitation order.

The beam search used by AQ-11 falls into the class of search orderings that are difficult to express. However, when the beam width is one, beam search devolves into hill climbing search, which can be expressed as a regular grammar, though somewhat awkwardly. Although the biases imposed by some search algorithms are difficult to represent, it should be remembered that these biases are themselves approximations of intractable biases. For example, AQ-11 uses a beam search to find a locally optimal approximation to the globally optimal term. Although the beam search is difficult to encode in RS-KII, it may be possible to find some other approximation of the intractable bias that can be expressed more naturally in RS-KII. This is an area for future work.

The remainder of this subsection describes the search ordering used by AQ-11, and how a restriction on this ordering can be expressed as a regular grammar in RS-KII. AQ-11 uses a two-tiered search. At the bottom level, it conducts a beam search through the space of terms for a term that covers a specified positive example and none of the negative examples. This search is guided by the LEF. Let the search order over the terms be specified by $\prec_t$ where $t_1 \prec_t t_2$ if t_1 and t_2 are terms, and t_1 is visited before t_2. The LEF may assign equivalent evaluations to some terms, in which case it does not matter which term is visited first. In this case, there is no ordering between the two terms.

The top level is a search through the space of hypotheses. In each iteration, the term found by the beam search is disjoined to the end of the current hypothesis. This can be viewed as a non-backtracking hill climbing search, where children are generated by disjoining a term to the parent hypothesis, and the terms are ordered according to $\prec_t$, with terms occurring earlier in $\prec_t$ being selected first.

Let $\prec_h$ be the order in which AQ-11 visits hypotheses, where $h_1 \prec_h h_2$ if h_1 and h_2 are both hypotheses, and h_1 is visited before h_2. The LEF and the fixed search algorithm are translated into a preference defined in terms of $\prec_h$. Namely, the preference set P is $\{\langle x, y\rangle \in H \times H \mid y \prec_h x\}$, which says that hypothesis x is less preferred than y if x is visited after y.

P is essentially the preference set for $\prec_h$, $\{\langle a, b\rangle \in H \times H \mid a \prec_h b\}$, except that the order of the tuples is reversed. If $\prec_h$ can be expressed as a regular grammar, it is likely that P can also be expressed as a regular grammar. To express $\prec_h$ as a regular grammar, a decision procedure is first defined that determines whether

$h_1 \prec_h h_2$ for arbitrary hypotheses h_1 and h_2. This procedure is then expressed as a regular grammar.

A hill climbing search first visits the root of the search tree, followed by the best child of that root. The subtree rooted at the child is searched recursively in the same fashion until a goal node is found. If no goal is found after searching to the bottom of the tree, the search would backtrack from the leaf node, and search the next best child of the leaf's parent. This is a pre-order traversal of the search tree. The root is visited first, and then the subtrees rooted at each child node are searched recursively from best child to worst child, as determined by the evaluation function.

The top-level search of AQ-11 is a hill-climbing search. Let hypothesis h_1 be $t_{1,1} \vee t_{1,2} \vee \ldots \vee t_{1,n}$, and let hypothesis h_2 be $t_{2,1} \vee t_{2,2} \vee \ldots \vee t_{2,m}$, where $t_{i,j}$ is a term. The top-level search visits h_1 before h_2 if h_1 comes before h_2 in pre-order. This can be determined by comparing the terms of each hypothesis from left to right. If $t_{1,1} \prec_t t_{2,1}$, then h_1 comes first, and if $t_{2,1} \prec_t t_{1,1}$, then h_2 comes first. If the first terms of both hypotheses are the same, then the next pair of terms, $t_{1,2}$ and $t_{2,2}$, are compared in the same fashion. This continues until the terms differ, or until one hypothesis runs out of terms. In this case, the longer hypothesis is an extension of the shorter one, and therefore a descendent of the shorter one. Hill climbing visits descendents after visiting the parent, so the shorter one is visited first.

This decision procedure is based on $\prec_t$, the order in which the beam search visits the terms, as guided by the LEF. In a beam search, the terms in the beam can be visited in any order. There is one beam for each iteration of the search, and the terms in each beam are visited after the terms in all of the beams for prior iterations. Terms that participate in none of the beams are never visited, and therefore least preferred. Determining whether term t_1 is visited before t_2 is difficult, since it requires knowing which beams the terms are in, which in turn requires knowing the beams for each of the previous iterations. This information is difficult to compute from t_1 and t_2, other than by running a beam search to determine whether t_1 or t_2 is visited first.

However, when the beam width is set to one, a beam search becomes a hill climbing search, for which it is easier to specify the search order. A hill climbing search visits the search tree in pre-order, as described above for the top-level search. Nodes in this search tree are terms. A term is a conjunction of selectors, $s_1 s_2 \ldots s_n$. To determine which of two terms is visited first, compare their selectors from left

to right. Let $t_1 = s_{1,1}s_{1,2}\ldots s_{1,n}$ and let $t_2 = s_{2,1}s_{2,2}\ldots s_{2,m}$. Term t_1 is visited first if the LEF assigns a better evaluation to $s_{1,1}$ than to $s_{2,1}$. If $s_{2,1}$ is better, then t_2 comes first. If both $s_{1,1}$ and $s_{2,1}$ are equally good, then compare $s_{1,2}$ and $s_{2,2}$ in the same fashion. Compare from left to right until one term has a better selector. If one term has no more selectors, then the shorter term is preferred.

One caveat for this search is that the *LEF* is an evaluation for terms, not for selectors. For a term $s_1s_2\ldots s_n$, the evaluation of selector s_n is $LEF(s_1s_2\ldots s_n)$, not $LEF(s_n)$.

To summarize, the preference order between two hypotheses, h_1 and h_2 is determined as follows. Compare the terms of the two hypotheses from left to right according to $\prec_t$. The terms are compared by $\prec_t$ in the same fashion. The selectors of each term are compared from left to right until one term runs out of selectors, or the selector of one term has a lower evaluation than the corresponding selector of the other term. Recall that the evaluation of the i^{th} selector in a term is $LEF(s_1s_2\ldots s_i)$, where s_1 through s_{i-1} are the preceding selectors in the term. This is the procedure for determining whether $h_1 \prec_h h_2$.

The ordering $\prec_h$ is essentially a *lexicographic ordering.* A lexicographic ordering is an alphabetical ordering over strings in some alphabet, where the symbols in the alphabet are ranked according to some total ordering. For example, a lexicographic ordering over strings in $(a-z)^*$ is the standard dictionary order.

The $\prec_h$ ordering can be mapped onto a lexicographic ordering over strings in $(0-9)^*$, where the symbols in the alphabet $(0-9)$ have the usual ordering for digits. This ordering can be easily expressed as a regular grammar, as can the mapping from hypotheses onto digit strings. These two grammars can be composed into a single regular grammar according to a construction that will be explained later. This is the regular grammar for $\prec_h$.

A hypothesis is mapped onto a digit string as follows. Each selector is replaced by the evaluation assigned to it by the LEF. Evaluations are assumed to be fixed-length integers of length l, where lower valued integers correspond to better evaluations. The $\vee$ symbols between terms are replaced by a zero. This mapping is shown in Figure 5.6. Hypothesis h_1 is preferred to hypothesis h_2 if $M(h_1) \prec_{lex} M(h_2)$, where $\prec_{lex}$ is the standard lexicographic ordering over strings of digits.

$$
\begin{aligned}
M(t_1 \vee t_2 \vee \ldots \vee t_n) &= M_t(t_1) \cdot 0 \cdot M_t(t_2) \cdot 0 \cdot \ldots \cdot 0 \cdot M_t(t_n) \\
M_t(s_1 s_2 \ldots s_m) &= LEF(s_1) LEF(s_1 s_2) \ldots LEF(s_1 s_2 \ldots s_n)
\end{aligned}
$$

Figure 5.6: Mapping Function from Hypotheses onto Strings.

$TranLEF(H, LEF, examples) \rightarrow \langle C, P \rangle$ where
$C = H$ *(i.e., no constraints)*
$P = \{\langle x, y \rangle \in H \times H \mid M(y) \prec_{lex} M(x)\}$
(M has access to H, the LEF, and the examples)

Figure 5.7: Translator for the LEF.

The reason for placing a zero between terms can be seen in the following example. Let h_1 be $t_1 \vee t_2 \vee s_1$ and let h_2 be $t_1 \vee t_2 s_2 \vee t_3$, where $t_2 s_2$ is the conjunction of term t_2 and selector s_2. Without the zeros between terms, $M(h_1)$ is $M_t(t_1) \cdot M_t(t_2) \cdot LEF(s_1)$, and $M(h_2)$ is $M_t(t_1) \cdot M_t(t_2) LEF(t_2 s_2)$. If $LEF(t_2 s_2)$ is less than $LEF(s_1)$, then h_1 will be preferred to h_2, which is not correct. The evaluation of $t_2 s_2$ is being improperly compared to the evaluation of $t_2 \vee s_2$ instead of t_2 because the presence of the disjunction ($\vee$) is not being taken into account. Appending a zero to the end of a term's evaluation assigns a better evaluation to a terminated term than to any extension of that term, thereby producing the correct comparison.

Given this mapping, the preference ordering over the hypotheses imposed by the LEF can be expressed as a preference set P. A hypothesis is preferred if its mapped string comes earlier in the lexicographic ordering, and less preferred if it comes later. The preference set P is therefore $\{\langle x, y \rangle \in H \times H \mid M(y) \prec_{lex} M(x)\}$, where H is the hypothesis space and $\prec_{lex}$ is the standard lexicographic ordering over strings of digits. This leads to the translators for the LEF shown in Figure 5.7. This translator takes as input the hypothesis space (H), and the LEF. Since the LEF often depends on additional knowledge, this information is also provided to the translator. In most cases, this information is the examples covered and not covered by the term being evaluated, so the translator takes as input the list of examples.

The next step is to express C and P as regular sets. C is an instantiation of the VL_1 hypothesis space, all of which can be expressed as regular grammars as shown in Figure 5.1. Representing P as a regular set is a little more difficult. This is discussed in Section 5.2.2.1, below.

5.2.2.1 Constructing P

A pair of hypotheses, $\langle x, y \rangle$, is a member of P if $M(x)$ comes after $M(y)$ lexicographically. This can be expressed as a composition of two simpler DFA, one for the lexicographic ordering, and one for the mapping, M.

The Lexicographic Ordering. The DFA for the standard lexicographic ordering over strings of digits, $\prec_{lex}$, is fairly simple. The DFA takes as input a pair of shuffled strings, $w = \mathit{shuffle}(w_1, w_2)$, where w_1 and w_2 are strings of digits from $(0-9)^*$. The symbols in w alternate between w_1 and w_2, so that the first two digits in w are the first digit from w_1 and the first digit from w_2. These two digits are compared, and if the digit from w_1 is lower, then w is accepted ($w_1 \prec_{lex} w_2$). If the digit from w_1 is higher, then w_1 comes after w_2, and w is rejected. Otherwise, the next pair of symbols are compared in the same manner. If one string runs out of symbols, and the two strings are equivalent up to this point, the shorter string is preferred. If w_1 is shorter, then w is accepted, and if w_1 is longer, w is rejected. This DFA is shown in Figure 5.8.

The Mapping. The mapping, M, is expressed as a kind of DFA known as a Moore machine [Hopcroft and Ullman, 1979]. A Moore machine is a DFA that has an output string associated with each state. The output string for each state is composed of symbols from an output alphabet, Δ. The output string for a state is emitted by the machine whenever it enters that state. The Moore machine for M takes as input a hypothesis, h, and emits a string w such that $w = M(h)$. Formally, a Moore machine is specified by the tuple $\langle Q, s, \delta : Q \times \Sigma \rightarrow Q, \mathit{out} : Q \rightarrow \Delta^l, \Sigma, \Delta \rangle$, where l is a (usually small) fixed integer.

The Moore machine for M is constructed as follows. Since the mapping depends on the LEF, the Moore machine is is parameterized by the LEF. The machine takes a hypothesis as input. When it recognizes a selector, it emits the string

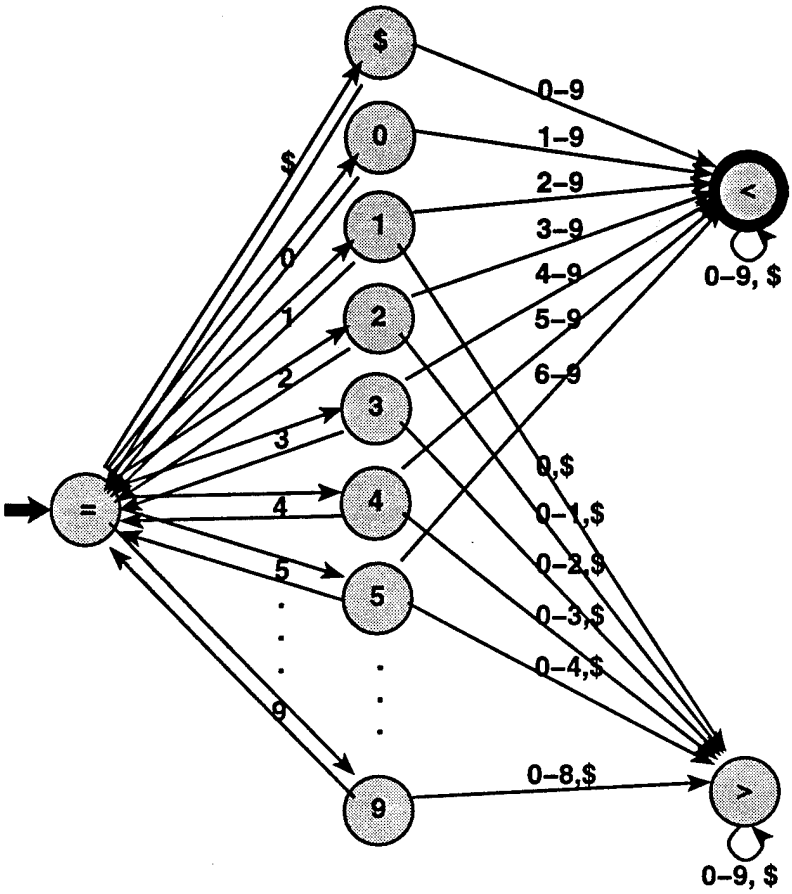

Figure 5.8: DFA Recognizing $\{shuffle(x, y) \in (0|1)^{2k} \mid x < y\}$.

$LEF(s_1 s_2 \ldots s_n)$, where s_n is the selector it has just recognized, and $s_1 s_2 \ldots s_{n-1}$ are the preceding selectors in the term.

Since each state in the machine can have only one output string, there must be at least one state in the machine for each possible term. This is clearly impossible. However, most evaluation functions evaluate a term based on the number of positive and negative examples covered by the term. Instead of having one state for each term, the machine has one state for each way of partitioning the examples into covered and uncovered subsets. Each partitioning corresponds to a set of terms that are all assigned the same evaluation by the LEF. The list of covered and uncovered examples is passed to a modified version of the LEF, which returns an evaluation of the term based on this information rather than by looking at the term itself. The evaluation returned by the LEF is the output string for this state.

There are 2^n distinct ways to partition a list of n examples into sets of covered and uncovered examples, so the machine can have at least 2^n states. Fortunately, the DFA is represented as an implicit DFA, so all of these states do not have to be kept in memory.

Even though this Moore machine has $O(2^n)$ states, the cost of induction can be considerably less than $O(2^n)$ (it is in fact polynomial, when using only the knowledge used by AQ-11, as is demonstrated in Section 5.4). Enumerating a hypothesis from the deductive closure of $\langle C, P \rangle$ involves visiting some of the states in C and P. Only in the worst case (e.g., when the deductive closure is empty) are all of the states visited. Since the DFAs for C and P are represented implicitly, the cost of enumeration is proportional to the number of states visited, not to the total number of states in each DFA. The Moore machine for the mapping function is specified in Figure 5.9.

Behavior of the Moore Machine. The machine takes as input a hypothesis, h, and outputs $M(h)$, where M is the mapping described previously in Figure 5.6. Recall that $M(t_1 \vee t_2 \vee \ldots \vee t_k)$, where t_1 through t_k are terms, maps onto the string $M_t(t_1) \cdot 0 \cdot M_t(t_2) \cdot 0 \ldots 0 \cdot M_t(t_k)$. The mapping M_t maps a term, $s_1 s_2 \ldots s_m$, where s_1 through s_m are selectors, onto the string $LEF(s_1) \cdot LEF(s_1 s_2) \cdot \ldots \cdot LEF(s_1 s_2 \ldots s_m)$. The LEF assigns an l-digit integer to each term. As was discussed previously, this value depends on which examples are covered by the term, and on which positive examples are covered by the previous terms in the hypothesis. Recall that a term is any sequence of selectors, so that s_1, $s_1 s_2$, and so on are all terms, not just $s_1 s_2 \ldots s_m$.

The machine keeps track of which examples are covered by the current term, and which examples are covered by the previous terms in the hypotheses. This is done by maintaining two binary vectors, $\vec{E_t}$ and $\vec{E_h}$, of length n, where n is the number of examples. The vector $\vec{E_t}$ indicates which examples are covered by the current term, and $\vec{E_h}$ indicates which examples are covered by the current hypothesis (all terms but the current one). A one in position i of the vector indicates that example i is covered, and a zero indicates that example i is not covered. The vector for the hypothesis is initially all zeros, since the initial hypothesis is `false`. The vector for the current term is all ones, since the term is initially `true`.

Parameters:

E an ordered list of n examples

$LEF : (\vec{E}_h, \vec{E}_t) \rightarrow (1-9)^l$ where

$\vec{E}_h$ is a binary vector of length n indicating which examples are covered by the hypothesis.

$\vec{E}_t$ is a binary vector of length n indicating which examples are covered by the current term.

l is a small, fixed integer.

SEL is a DFA that recognizes selectors

f_1 through f_k are the features

A Moore machine for M is $\langle Q, s, \delta : Q \times \Sigma \rightarrow Q, out : Q \rightarrow \Delta^l, \Sigma, \Delta \rangle$ where:

$$
\begin{aligned}
Q &= \{\langle \vec{E}_h, \vec{E}_t, end \rangle \mid \vec{E}_h, \vec{E}_t \in (0|1)^n, end \in \{0,1\}\} \cup \{d\} \\
&\quad end \text{ is a flag indicating whether the machine has just seen the end of a term (1), or is in the middle of a term (0). } d \text{ is the terminal state.} \\
s &= \langle \vec{0}^n, \vec{1}^n, 0 \rangle \\
\delta^*(\langle \vec{E}_h, \vec{E}_n, end \rangle, w) &= \begin{cases} \text{if } w \in \text{SEL} & \text{Clear bit } i \text{ of } \vec{E}_t \text{ if example } i \text{ not covered by selector } w. \text{ Set } end \text{ to 0. Goto state } \langle \vec{E}_h, \vec{E}_t, 0 \rangle. \\ \text{if } w = \text{“}\vee\text{”} & \vee \text{ marks the end of a term. Set } end \text{ to 1. Set bit } i \text{ of } \vec{E}_h \text{ to 1 if bit } i \text{ of } \vec{E}_t \text{ is 1, else leave it unchanged. Set every bit in } \vec{E}_t \text{ to 1. Goto state } \langle \vec{E}_h, \vec{E}_t, e \rangle \\ \text{if } w = \text{“\$”} & \text{Goto state } d, \text{ the terminal state.} \end{cases} \\
\delta^*(d, w) &= d \\
out(\langle \vec{E}_h, \vec{E}_t, end \rangle) &= \begin{cases} LEF(\vec{E}_h, \vec{E}_t) & \text{if } end = 0 \\ 0^l & \text{if } end = 1 \end{cases} \\
out(d) &= \epsilon, \text{ the empty string.} \\
\Sigma &= \{[, f_1, f_2, \ldots, f_k, <, \leq, =, \geq, >, 0-9,], \vee, \$\} \\
\Delta &= \{0,1,2,3,4,5,6,7,8,9\}
\end{aligned}
$$

Figure 5.9: Moore Machine for the Mapping.

When a selector is seen, $\vec{E}_t$ is updated by clearing bit i of the vector if the selector does not cover example i. Since a term is a conjunction of selectors, if any selector in the term fails to cover an example, there is no way to cover the example by adding more selectors to the term. Therefore, seeing a selector never causes a bit in $\vec{E}_t$ to be turned on. After seeing the selector, the machine outputs the value assigned by the LEF to the current term. This value depends on the examples covered by the current hypothesis and the current term. Specifically, the machine outputs $LEF(\vec{E}_h, \vec{E}_t)$.

The machine continues in this fashion until the symbol $\vee$ is seen. This symbol marks the completion of the current term and the beginning of a new term. The completed term is now considered part of the current hypothesis, and a new current term is started. The new current hypothesis covers every example covered by the completed term, since the the term is a disjunct of the hypothesis. Bit i of $\vec{E}_h$ is set to one if bit i of $\vec{E}_t$ is one, and is left unchanged otherwise. This is equivalent to computing the logical or of $\vec{E}_h$ and $\vec{E}_t$. Since a new current term has started, the vector $\vec{E}_t$ is set to all ones. The machine outputs a 0 to mark the end of the term. More specifically, it outputs a string of l zeros, since the output strings of the machine must all be of the same length, and the strings output by the LEF are of length l.

The machine continues to process symbols as described above until the \$ symbol is seen, signifying the end of the hypothesis, at which point the machine halts. Specifically, it moves into a state that has an edge to itself on every input symbol, and has no output string associated with it.

The only memory available to a Moore machine are its states. Therefore, the vectors $\vec{E}_h$ and $\vec{E}_t$ correspond to states in the machine. The vectors are of finite length, so there are finite number of states in the machine. When a selector is seen, the vectors are recomputed as described above, and the machine moves to this state. Each state has an output string associated with it, which is emitted when the machine moves to that state. Specifically, the machine emits the string $LEF(\vec{E}_h, \vec{E}_t)$ after seeing each selector, and the string 0^l after seeing each $\vee$ symbol. Since the tuple $\langle \vec{E}_h, \vec{E}_t \rangle$ does not indicate whether a selector or a $\vee$ symbol was just seen, an additional bit, labeled *end*, is added to the state. A state in the machine is therefore of the form $\langle \vec{E}_h, \vec{E}_t, end \rangle$. When *end* is one, indicating that a $\vee$ symbol has just

been seen, the output for the state is 0^l. When *end* is zero, the output for the state is $LEF(\vec{E}_h, \vec{E}_t)$.

Composing the Grammars. The DFA for $P = \{\langle x, y\rangle \in H \times H \mid M(y) \prec M(x)\}$ is computed from M (the Moore machine for the mapping), and from the DFA for $\prec$ (the standard lexicographic ordering over strings of digits). The idea is to use two copies of M, and pass x to one of them, and y to the other. The output from the copies of M, is passed to the DFA for $\prec$, which determines whether $M(x) \prec M(y)$.

Input strings to P are of the form $shuffle(a, b) = a_1 b_1 a_2 b_2 \ldots a_n b_n$ where a and b are hypotheses, and a_i and b_i are symbols in a and b. P maintains two copies of M, M_a and M_b. On odd numbered input symbols (i.e, symbols from a), P simulates a move on M_a, and on even numbered symbols (from b) P simulates a move on M_b. When one move has been made in both M_a and M_b, the output strings, w_a and w_b, from each Moore machine are shuffled together. Moves are then simulated in the DFA for $\prec$ on the shuffled input string, $shuffle(w_b, w_a)$—that is, the DFA tests whether $w_b \prec w_a$. P is in an accept state iff the DFA for $\prec$ is in an accept state. P recognizes the set $\{\langle a, b\rangle \in H \times H \mid M(b) \prec M(a)\}$.

The above construction for P is essentially a substitution of the regular language generated by the Moore machine into the regular language accepted by the DFA for $\prec$. Regular languages are closed under substitution [Hopcroft and Ullman, 1979], and all Moore machines generate regular languages, so this construction always yields a regular grammar for P.

An Information Gain Metric. One common way to compare terms is with an information gain metric (e.g., [Quinlan, 1986]). The information of a term is a measure of how well it distinguishes between the positive and negative example. Terms with higher information are preferred.

Let p_0 be the number of positive examples covered by the term, let n_0 be the number of negative examples by the term, and let p_1 and n_1 be the number of positive and negative examples that are not covered by the term. Let p and n be

the total number of positive and negative examples, respectively. The information of the term is shown in Equation 5.1.

$$-\frac{p_0+n_0}{p+n}\left[\frac{p_0}{p_0+n_0}\log_2\frac{p_0}{p_0+n_0}+\frac{n_0}{p_0+n_0}\log_2\frac{n_0}{p_0+n_0}\right]+$$
$$\frac{p_1+n_1}{p+n}\left[\frac{p_1}{p_1+n_1}\log_2\frac{p_1}{p_1+n_1}+\frac{n_1}{p_1+n_1}\log_2\frac{n_1}{p_1+n_1}\right] \quad (5.1)$$

When this information metric is used to evaluate terms, the LEF function is defined as follows. The LEF takes as input the vectors $\vec{E_t}$ and $\vec{E_h}$), which indicate the examples covered by the current term and the previous terms in the hypothesis. The evaluation is the information of the current term with respect to the negative examples and the positive examples that have not yet been covered by the previous terms. If the positive examples covered by the previous terms are not excluded, the same term gets selected in each call to *LearnTerm*.

Bit i of $\vec{E_t}$ is on if the term covers example i, and is off if the examples is not covered. The LEF is assumed to know which bits in each vector correspond to positive and negative examples. The total number of negative examples covered and not covered by the term (n_0 and n_1) can be computed from $\vec{E_h}$ by counting the number of ones and zeros in the appropriate positions.

To compute the number of positive examples in the pot, a bit-wise complement of $\vec{E_h}$ is first performed. A bit in the resulting vector is on if example i is not covered by any of the previous terms. The number of ones in this vector corresponding to positive examples are computed. This is p, the total number of positive examples in the *pot*—the set of positive examples not yet covered by the current hypothesis.

The number of positive examples in the pot that are covered by the term (p_0) can be determined by computing the logical *and* of $\vec{E_t}$ with the complement of $\vec{E_h}$. Bit i in this vector is on if example i is covered by the term, but not by any of the previous terms. To compute p_1, the number of positive examples in the pot that are not covered by the term, subtract p_0 from p.

The values for n_0, n_1, p_0, p_1, p and n are then substituted into Equation 5.1, which returns a real number between -1 and 1. The first l digits of this number constitute the digit string returned by $LEF(\vec{E_h}, \vec{E_t})$.

5.2.3 Domain Theory

A domain theory encodes background knowledge about the target concept as a collection of horn-clause inference rules that explain why an instance is a member of the target concept. The way in which this knowledge biases induction depends on assumptions about the correctness and completeness of the theory. Each of these assumptions requires a different translator, since the bias maps onto different constraints and preferences.

A complete and correct theory can explain why every instance covered by the target concept is a member of the concept. The theory exactly describes the target concept. This is a very strong form of bias, since the theory identifies the target concept uniquely and accurately. No other knowledge is necessary. The bias can be expressed as a constraint that is satisfied only by the target concept. This bias is used in algorithms such as Explanation Based Learning [DeJong and Mooney, 1986, Mitchell *et al.*, 1986].

Domain theories are not always complete nor correct. An incomplete theory can explain some instances, but not all of them. This occurs, for example, when the theory is overspecific. In an overspecific theory, the target concept is some generalization of the theory. The bias imposed by an overspecific theory can be translated as a constraint that is satisfied only by generalizations of the theory.

In addition to being incomplete, a theory can also be incorrect. An incorrect theory misclassifies some instances. One common kinds of incorrectness is overgenerality. An overgeneral theory explains all of the examples covered by the target concept, and some that are not. An overgeneral theory can be expressed as a constraint that is satisfied only by hypotheses that are specializations of the theory. Overgeneral theories are used by algorithms such as IOE [Flann and Dietterich, 1989] and OCCAM [Pazzani, 1988].

A translator for a particular overgeneral domain theory is described below. The theory being translated is derived from the classic "cup" theory [Mitchell *et al.*, 1986, Winston *et al.*, 1983], as shown in Figure 5.10. Since the theory is assumed to be overgeneral, the target concept must be a specialization of the theory. This bias is expressed as a constraint, which is satisfied only by specializations of the theory. More precisely, the theory is a disjunction of several sufficient conditions for CUP,

```
cup(X) :- hold_liquid(X), liftable(X), stable(X), drinkfrom(X).
hold_liquid(X) :- plastic(X) | china(X) | metal(X).
liftable(X) :- small(X), graspable(X).
graspable(X) :- small(X), cylindrical(X) | small(X), has_handle(X).
stable(X) :- flat_bottom(X).
drinkfrom(X) :- open_top(X).
```

Figure 5.10: CUP Domain Theory.

and it is assumed that some disjunction of these conditions is the correct definition for CUP.

The domain theory is a union of several sufficient conditions that can be expressed in terms of the *operational predicates.* The target concept is assumed to be equivalent to a disjunction of one or more of these conditions. The operational predicates are those that meet user-defined criteria for ease-of-evaluation and comprehensibility, among others. For this theory, the only operationality criteria for predicates is that they can be evaluated on an instance (e.g., small(X) can be determined for instance X, but cup(X) cannot). For this theory, the leaf predicates are the operational predicates. The sufficient conditions for the theory are as follows:

1. cup(X) :- plastic(X), small(X), cylindrical(X), flat_bottom(X), open_top(X).
2. cup(X) :- china(X), small(X), cylindrical(X), flat_bottom(X), open_top(X).
3. cup(X) :- metal(X), small(X), cylindrical(X), flat_bottom(X), open_top(X).
4. cup(X) :- plastic(X), small(X), has_handle(X), flat_bottom(X), open_top(X).
5. cup(X) :- metal(X), small(X), has_handle(X), flat_bottom(X), open_top(X).
6. cup(X) :- china(X), small(X), has_handle(X), flat_bottom(X), open_top(X).

The target concept is assumed to be a disjunction of one or more of these conditions. This bias is translated as a constraint satisfied by hypotheses that are equivalent to these conditions. The regular grammar for the constraint is computed by mapping each condition onto an equivalent hypothesis (or set of hypotheses), and computing the union of these hypotheses.

In order to map the conditions, the hypothesis space language is assumed to have features corresponding to each of the operational predicates in the theory. The operational predicates are all Boolean valued (e.g., small(X) is either true or false), so the corresponding feature is also Boolean valued. This leads to selectors such as [`small = true`] and [`small = false`]. The features are listed below. Each feature has the same name as its corresponding predicate.

`plastic`	`small`	`flat_bottom`
`china`	`cylindrical`	`open_top`
`metal`	`has_handle`	

The regular grammar for the set of hypotheses corresponding to disjunctions of sufficient conditions (specializations) of the domain theory is expressed as shown in Figure 5.11. This grammar is a little less general than it could be, since it does not allow all permutations of the selectors within each term. However, the more general grammar contains considerably more rules, and permuting the selectors does not change the semantics of a hypothesis.

C	→	TERM | C or TERM
TERM	→	SUFFICIENT-CONDITION
SUFFICIENT-CONDITION	→	(PLASTIC | CHINA | METAL)
		(HAS_HANDLE | CYLINDRICAL)
		SMALL FLAT_BOTTOM OPEN_TOP
PLASTIC	→	`[plastic = true]`
CHINA	→	`[china = true]`
METAL	→	`[metal = true]`
HAS_HANDLE	→	`[has_handle = true]`
CYLINDRICAL	→	`[cylindrical = true]`
SMALL	→	`[small = true]`
FLAT_BOTTOM	→	`[flat_bottom = true]`
OPEN_TOP	→	`[open_top = true]`

Figure 5.11: Grammar for VL_1 Hypotheses Satisfying the CUP Theory Bias.

5.3 An Induction Task

This section provides two concrete examples of AQ-11 and RS-KII solving simple induction tasks. In the first task, both AQ-11 and RS-KII use only the knowledge available to AQ-11. Both algorithms learn the same hypothesis. The second task is a synthetic one designed to show the effects of making a domain theory available. The theory is the CUP theory described in the previous section. AQ-11 cannot use the theory, and so is forced to learn from examples and the LEF alone. RS-KII has access to the domain theory as well, and this improves the accuracy of the hypothesis.

5.3.1 Iris Task

The first induction task is taken from the Iris domain [Fisher, 1936]. The data set contains three classes of fifty instances each. Each class corresponds to a different kind of iris. The three classes are as follows.

1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

The goal is to learn a VL_1 hypothesis that identifies one of the classes as distinct from the other two. For this task, the goal is to learn the first class, Iris Setosa.

Instances are 4-tuples consisting of the values of four objective measurements taken from a given plant. Specifically, these features are as follows (in order):

1. Sepal length in millimeters
2. Sepal width in millimeters
3. Petal length in millimeters
4. Petal width in millimeters

The values for these features are integers between 0 and 300.

5.3.1.1 Translators for Task Knowledge

The VL_1 hypothesis space is parameterized with a list of features and a set of values for each feature. There are four features, each of which can be a positive integer. This leads to the following list of parameters for VL_1 shown in Table 5.1. Instantiating V_1 with these parameters yields the grammar shown in Figure 5.12.

feature	values
f_1	$(1-9)(0-9)^*$
f_2	$(1-9)(0-9)^*$
f_3	$(1-9)(0-9)^*$
f_4	$(1-9)(0-9)^*$

Table 5.1: Parameters for VL_1.

VL1 $\rightarrow$ TERM | TERM or VL1
TERM $\rightarrow$ SELECTOR | SELECTOR TERM
SELECTOR $\rightarrow$ "[" f_1 RELATION $(1-9)(0-9)^*$ "]" |
"[" f_2 RELATION $(1-9)(0-9)^*$ "]" |
"[" f_3 RELATION $(1-9)(0-9)^*$ "]" |
"[" f_4 RELATION $(1-9)(0-9)^*$ "]"
RELATION $\rightarrow$ $< \mid \leq \mid = \mid \neq \mid \geq \mid >$

Figure 5.12: Grammar for Instantiated VL_1 Language.

Translators are needed for the positive and negative examples. An example is positive if it is an instance of the target class (e.g., Iris Setosa), and negative otherwise. Examples are translated by the *TranAQExample* translator shown previously in Figure 5.4.

The LEF is not specified by the Iris task. The information gain translator described earlier is a good general-purpose LEF, and is the one used in this example. The translator for the LEF is the one described in Section 5.2.2.

5.3.1.2 Results of Learning

The concept definition learned by both AQ-11 and RS-KII for Iris Setosa is the one-selector hypothesis [$f_3 < 30$].

5.3.2 The CUP Task

The second task is a synthetic one based on the CUP domain. The knowledge consists of a list of examples, an information-gain metric for evaluating hypotheses, and the CUP domain theory of Figure 5.10. RS-KII can make use of all of this knowledge, but AQ-11 cannot use the domain theory.

The features for this task are the operational predicates of the CUP theory, as shown below. All of these features are Boolean valued.

f_1: `plastic`	f_4: `cylindrical`	f_7: `flat_bottom`
f_2: `china`	f_5: `has_handle`	f_8: `open_top`
f_3: `metal`	f_6: `small`	

There are also two irrelevant features, `color` (f_9) and `percent-full` (f_{10}). The `color` feature can take the values {red, blue, green}, and the `percent-full` feature can take integer values between 0 and 100 indicating the amount of liquid in the cup.

The target concept is "plastic cups with handles", which is a specialization of the CUP domain theory. The concept is represented by the VL_1 hypothesis [f_1 = `true`] [f_5 = `true`] [f_6 = `true`] [f_7 = `true`] [f_8=`true`] . There are four examples of this concept, as shown in Table 5.2.

ID	class	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
e_1	+	t	f	f	f	t	t	t	t	blue	10
e_2	+	t	f	f	f	t	t	t	t	red	50
e_3	−	f	t	f	f	t	t	t	t	green	60
e_4	−	t	f	f	t	t	t	t	t	blue	20

Table 5.2: Examples for the CUP Task.

The other available knowledge sources are the CUP domain theory shown previously in Figure 5.10, and an information gain metric for evaluating hypotheses.

The examples are translated according to *TranAQExample*, as shown previously in Figure 5.4. The domain theory is translated into $\langle C, \{\}\rangle$, where the regular grammar for C is as shown in Figure 5.11. The information-gain metric is translated according to the translator described in Section 5.2.2.

5.3.2.1 Results of Learning

AQ-11 has no access to the domain theory. It learns a hypothesis that is consistent with the examples, and preferred by the information gain metric. It learns the following hypothesis:

$$[f_1 = \texttt{false}] \text{ and } [f_4 = \texttt{false}].$$

This hypothesis recognizes all plastic, cylindrical objects, even if they are not cups. In order to learn a more accurate concept, considerably more examples are necessary.

RS-KII can learn the correct hypothesis from the same number of examples. This is because RS-KII has access to the domain theory, which provides a very strong bias on the hypothesis space. It only considers hypotheses that are specializations of the domain theory. This drastically reduces the search space, increasing the accuracy of the induced concept. RS-KII learns the following concept:

$$[f_1 = \texttt{true}]\,[f_5 = \texttt{true}]\,[f_6 = \texttt{true}]\,[f_7 = \texttt{true}]\,[f_8 = \texttt{true}]$$

This is the only hypothesis that is both consistent with the examples, and a specialization of the domain theory.

5.4 Complexity Analysis

The computational complexity of RS-KII depends on the number of knowledge fragments it utilizes as well as the nature of those fragments. A general analysis of RS-KII's complexity would depend on the nature of the knowledge, which is impossible to quantify meaningfully. However, it is possible to obtain a complexity analysis for particular classes of knowledge. This is the approach taken in this section to compare the complexity of RS-KII to AQ-11.

5.4.1 Complexity of AQ-11

In the following complexity analysis, e denotes the number of examples, k indicates the number of features, and b is the width of the beam search.

5.4.1.1 Cost of AQ-11's Main Loop

The AQ-11 algorithm consists of a main loop that calls *LearnTerm* once per iteration. On each iteration, the term found by *LearnTerm* is disjoined to the end of the current hypothesis, and the positive examples covered by the term are removed from the pot. Removing covered examples from the pot takes time proportional to e. The main loop iterates until the pot is empty—i.e., every positive example is covered by at least one of the terms. Since each term is guaranteed to cover at least one positive example that the other terms do not cover, the main loop makes at most one call to *LearnTerm* for each positive example. The worst-case complexity of the main loop is therefore $O(e(x+e))$ where x is the complexity of *LearnTerm*

5.4.1.2 Cost of LearnTerm

LearnTerm conducts a beam search through the space of terms to find a term that covers a given positive example (the *seed*) and none of the negative examples. The search maintains a "pot" of negative examples that are covered by at least one term in the beam. A set of selectors is computed that covers the positive seed but not the negative examples. Child terms are generated from the terms in the beam by extending them with the selectors in this set. The child terms are evaluated according to the LEF, and the best b terms are retained in the beam.

The cost of *LearnTerm* depends partly on the number of selectors that cover the positive seed but not the negative example. In theory, this set can be quite large, or even infinite, which would make AQ-11 intractable. In practice, the set can be made much smaller with a simple bias. The generation of this set of selectors is discussed below, and an upper bound on its size is derived. The cost of *LearnTerm* is then computed using this bound.

Generating the Set of Selectors. The complete set of selectors that cover the positive seed but not the negative example is generated as follows. Let $v_{p,i}$ be the

value of the positive seed on feature f_i, and let $v_{n,i}$ be the value of the negative example on feature f_i. The union of the following sets of selectors over all k features is the set of selectors that cover the positive seed but not the negative example:

- $[f_i = v_{p,i}]$ if $v_{p,i} \neq v_{n,i}$
- $[f_i \neq v_{n,i}]$ if $v_{p,i} \neq v_{n,i}$
- $\{[f_i < v] \mid v_{p,i} < v \leq v_{n,i}\}$ if $v_{p,i} < v_{n,i}$.
- $\{[f_i \leq v] \mid v_{p,i} \leq v < v_{n,i}\}$ if $v_{p,i} < v_{n,i}$
- $\{[f_i > v] \mid v_n \leq v < v_p\}$ if $v_{p,i} > v_{n,i}$
- $\{[f_i \geq v] \mid v_n < v \leq v_p\}$ if $v_{p,i} > v_{n,i}$

The last four sets in this list can contain a large to infinite number of selectors, depending on whether the feature values map onto the integers or the reals. However, many of these selectors are redundant, in that they can be partitioned into classes of "equivalent" selectors. For any hypothesis, one selector can be substituted for any other in the same class, and the resulting hypothesis will still cover the same examples. Of course, the two hypotheses may assign different classes to new instances. This suggests a way to bias the set of selectors—namely, include only one selector from each class in the set. As will be argued below, the number of classes for a given feature is bounded by the number of examples. If there are k features, the biased set of selectors contains at most $O(ek)$ elements.

Consider the set of selectors, $\{[f_i < v] \mid v_{p,i} < v \leq v_{n,i}\}$ where $v_{p,i} < v_{n,i}$. A selector partitions the examples into a set of covered and uncovered examples. The selectors in this set can induce at most e different partitions on the examples. To see why this is so, take the value of each example on feature f_i, and order the values from smallest to largest. If there are e examples, there are at most e partitions that can be induced by selectors of the form $[f_i < v]$: the first example in the list is covered, the first two are covered, and so on up to all of the examples being covered. For each of these partitions, the characteristic selector for the equivalence class is $[f_i < v]$ where v is one of the values in the list of examples. The set of characteristic selectors is $\{[f_i < v] \mid v_{p,i} < v \leq v_{n,i}$ and v is the value of some example on feature $f_i\}$.

The characteristic selectors for the other sets of selectors are computed similarly. Let E be the set of positive and negative examples, and let $\pi_i(E)$ be the projection of the examples onto their values for feature f_i—that is, $\pi_i(E)$ is the set of values held by one or more of the examples on feature f_i. The set of characteristic selectors that cover the positive seed but not the negative seed is the union of the following sets for i between one and k:

- $\{[f_i = v_{p,i}]\}$ if $v_{p,i} \neq v_{n,i}$
- $\{[f_i \neq v_{n,i}]\}$ if $v_{p,i} \neq v_{n,i}$
- $\{[f_i < v] \mid v_{p,i} < v \leq v_{n,i} \text{ and } v \in \pi_i(E)\}$ if $v_{p,i} < v_{n,i}$
- $\{[f_i \leq v] \mid v_{p,i} \leq v < v_{n,i} \text{ and } v \in \pi_i(E)\}$ if $v_{p,i} < v_{n,i}$
- $\{[f_i > v] \mid v_n \leq v < v_p \text{ and } v \in \pi_i(E)\}$ if $v_{p,i} > v_{n,i}$
- $\{[f_i \geq v] \mid v_n < v \leq v_p \text{ and } v \in \pi_i(E)\}$ if $v_{p,i} > v_{n,i}$

Each of these sets contains at most e elements, since $\pi_i(E)$ contains at most $e = |E|$ values. The union of these sets over all k features contains at most $O(ke)$ selectors. The time needed to generate this set is also $O(ke)$.

Cost of One Iteration. Each iteration of *LearnTerm* consists of six steps:

1. Select a negative example from the pot. This takes time O(1).
2. Compute S, the set of selectors that covers the *seed*, but does not cover the negative example selected in line (1). Computing S takes time $O(ek)$, and contains $O(ek)$ selectors. The derivation of these results is given below.
3. Generate the children of the terms in the beam: $\{t_i \cdot s \mid t_i \text{ is a term in the beam, and } s \in S\}$. Computing the children takes time $O(b|S|) = O(bek)$.
4. Evaluate the children with the LEF. In most cases, the evaluation of a term depends on the number of positive and negative examples the term covers. The time to evaluate all the child terms is therefore bounded by $O(bke^2)$

5. Sort the children according to the LEF, and select the first b children as the new beam. This takes time $O(b|S|\log(b|S|)) = O(bek\log(bek))$. If the beam width is one, the new beam is computed by selecting the best child. This only takes time $O(b|S|) = O(bek)$.

6. Remove from the pot of negative examples every example covered by none of the terms in the beam. This takes time $O(eb)$.

The total time for one iteration is bounded by the sum of these costs, or $O(|S| + b|S| + b|S|e + b|S|\log(b|S|) + eb)$, where S is the number of selectors that cover the seed but not the negative examples. Since there are at most ek such selectors, substituting ek for $|S|$ yields $O(ek + bek + bke^2 + bek\log(bek) + eb)$, or $O(bke^2 + bek\log(bek))$.

When the beam width is one, the term $bek\log(bek)$ can be replaced by bek, in which case the time cost for one iteration is bounded by $O(bke^2 + bek) = O(bke^2)$. Since the beam width ($b$) is one, this can be rewritten as $O(ke^2)$.

Total Cost of LearnTerm. *LearnTerm* makes at most one iteration per negative example. The number of negative examples are bounded by e, so the time complexity of one call to *LearnTerm* is is bounded by $O(bke^2\log(bke) + bke^3)$ when $b > 1$, or by $O(ke^3)$ when $b = 1$. These costs are summarized in Equation 5.2 and Equation 5.3.

$$O(ke^3) \quad \text{if} \quad b = 1 \tag{5.2}$$

$$O(bke^2\log(bke) + bke^3) \quad \text{if} \quad b > 1 \tag{5.3}$$

5.4.1.3 Total Time Complexity for AQ-11.

The total cost of AQ-11 is $O(e(x+e))$ where x is the time complexity of *LearnTerm*. This yields the following computational complexity for AQ-11:

$$O(bke^3\log(bke) + bke^4) \quad \text{if} \quad b > 1 \tag{5.4}$$

$$O(bke^4) \quad \text{if} \quad b = 1 \tag{5.5}$$

The $bke^3\log(bke)$ term is the cost of expanding and pruning the beam, and the bke^4 term is the cost of evaluating the terms with the LEF.

5.4.2 Complexity of RS-KII when Emulating AQ-11

When emulating AQ-11, RS-KII induces a hypothesis by enumerating a single hypothesis from the solution set of $\langle C, P \rangle$. This is accomplished by the branch-and-bound algorithm described in Figure 4.14 of Chapter 4. The parameters to the algorithm are shown in Figure 4.15 of the same chapter.

The branch-and-bound algorithm maintains L, a collection of subsets of C. On each iteration, one of these subsets, X_s, is selected. X_s is compared to all of the subsets in L. If X_s is not dominated by any of the subsets, then it contains only solutions, and an element of X_s is enumerated. Otherwise, X_s is replaced by $Split(X_s)$ in the collection, and dominated subsets are pruned from the collection. This continues until a hypothesis is enumerated, or L is empty.

The cost of one iteration depends on the number of subsets in L. An iteration consists of four steps, the costs of which are analyzed below.

1. Select X_s from L. This can be done in constant time if L is maintained according to the efficient schemes mentioned in Section 4.2.2.4.

2. Determine whether X_s is undominated by all of the subsets in L. This takes time $O(|L|t_{\not\prec})$, where $t_{\not\prec}$ is the cost of comparing two subsets with the not-dominated relation, $D_{\not\prec}$.

3. Split X_s. A subset of C is of the form $w_i \cdot q_i$, where w_i is some string and $q_i = \delta_C(start_C, w_i)$, the state reached by w_i in C. $Split(w_i \cdot q_i)$ generates a child subset of $w_i \cdot q_i$ by extending w_i with a symbol σ from Σ_C and changing q_i to be the state reached by the new string, $w_i\sigma$. There is at most one child subset for each symbol, for a total cost of $O(|\Sigma_C|)$.

4. Replace X_s with $Split(X_s)$ in L, and prune dominated subsets from L. None of the existing subsets in L dominate each other. The child subsets need to be compared to each other, and to each of the existing subsets in L.

   There are at most $|\Sigma_C|$ child subsets, and $a-1$ existing subsets, where a is the number of subset in L prior to replacing X_s with its children. The children are added to the collection as follows. Each child is compared to the $a-1$ existing subsets, and each of the $a-1$ existing subsets is compared to the child. If an

existing subset is dominated, it is removed from the collection. If the child is not dominated by any of the existing subsets, the child is added to the list. Otherwise the child is discarded.

For the first child, there are $2(a-1)$ comparisons. If the child is added to the collection, the collection contains a subsets. Determining whether the second child should be added requires $2a$ comparisons in this case. If each child is added to the collection, the total number of comparisons required is $\sum_{i=0}^{|\Sigma_C|-1} 2((a-1)+i)$, for a total cost of $O((|\Sigma_C|^2 + |\Sigma_C||L|)t_{\prec})$ where $t_{\prec}$ is the cost of comparing two subsets with the domination relation $D'_{\prec}$.

The total cost of one iteration is the sum of these costs. The costs of the two dominance relations, $t_{\prec}$ and $t_{\not\prec}$, are of the same order, so these values can be replaced by a single variable, t. The resulting computational complexity for one iteration of *BranchAndBound-3* is shown in Equation 5.6.

$$O((|\Sigma_C|^2 + |\Sigma_C||L|)t) \tag{5.6}$$

The Size of L. L initially contains a single subset that is equivalent to C. In each iteration, a subset is selected and split into at most Σ_C subsets. L can grow geometrically unless subsets are pruned aggressively.

The pattern of growth for L when RS-KII emulates AQ-11 is a cyclic one. It grows geometrically for a few iterations, and is then pruned back to a single subset. Call the initial single subset $X_1 = w_1 \cdot q_1$. This is split into subsets by adding symbols from Σ_C to the end of w_1, the prefix string for X_1. Symbols in Σ_C are not selectors, but rather components of selectors (i.e., feature names, the six relations, and digits from which to construct the value field). These subsets cannot be meaningfully compared since their prefix strings end in partially constructed selectors. Additional symbols are needed to form a complete selector, at which point the subsets can be compared.

Selectors are strings of up to l symbols, where l depends on the number of digits that can be in the value field of a selector. Subsets with the shortest prefix strings are selected first, so the subsets are expanded breadth first until of the subsets have complete selectors. L grows geometrically for these iterations.

At the end of this growth cycle, all of the subsets in L have complete selectors. The subsets can be compared with the dominance relation $\tilde{D}_{<}$, and the dominated

subsets pruned from L. The relation prunes a subset $w_i \cdot q_i$ if there is another subset in the collection, $w_j \cdot q_j$, such that $\mathit{shuffle}(w_i, w_j)$ leads to an accept-all state in P and $w_j \cdot q_j$ is not empty (i.e., q_j is not a dead state in C). Omitting the empty test yields the relation $\tilde{D}'_<$, which is used to order select the a subset from the collection.

Since P is a lexicographic ordering, if w_i comes before w_j, then every extension of w_i also comes before w_j and its extensions. This means that $\mathit{shuffle}(w_i, w_j)$ leads to an accept-all state in P. Since a lexicographic ordering is a total ordering, for every pair of subsets, $w_i \cdot q_i$ and $w_j \cdot q_j$ in the collection, either w_i comes before w_j, or w_j comes before w_i, or w_i and w_j are equivalent. In all but the last case, one of the two subsets is dominated, assuming the other is not empty. In a total ordering there can be at most one maximal element, so all but one subset can potentially be pruned, assuming the dominant subset can be shown to be non-empty.

However, it is not usually possible to tell whether a subset is empty, since C's *Dead* function returns `unknown` for most states. Therefore, L is not pruned. However, the way in which subsets are selected mitigates against this. The subsets are ordered according to $\tilde{D}'_<$, and the best subset is selected as X_s in the next iteration. The $\tilde{D}'_<$ relation is exactly $\tilde{D}_<$ without the empty test. The Branch-and-Bound algorithm sorts the subsets in L once in each iteration according to $\tilde{D}'_<$. L is stored as a lattice reflecting the known dominance relations among the subsets. The elements in the top of the order are the undominated elements according to $\tilde{D}'_<$. After splitting a subset sufficiently, it may be identified as either empty or non-empty instead of `unknown`. If it is non-empty, all the subsets dominated by it in L are pruned. Otherwise, the empty subset is removed from L.

The representation for L effectively allows the subsets to be compared according to $\tilde{D}'_<$ instead of $\tilde{D}_<$. The subsets at the top of the lattice are the undominated elements according to $\tilde{D}'_<$. Since $\tilde{D}'_<$ does not check for emptiness, it allows a subset to be dominated by an empty subset. However, dominated subsets are not pruned. They are only moved lower in the lattice. If the dominating subset later turns out to be empty, the dominated subsets may become new top elements. This allows incorrectly "pruned" subsets to be reinstated in L. Since $\tilde{D}'_<$ is a total ordering, and defined over all subsets that have prefix strings with complete selectors, there is at most one subset at the top of the lattice after the growth stage. When children are added, they are only compared to the subsets at the top of the lattice, since if

they dominate these elements, they certainly dominate those below it. In effect, $|L|$ is always one after each growth stage.

The size of L grows by $|\Sigma_C|$-fold until $|L|$ is the number of selectors (i.e., L contains all one-selector extensions of the initial single hypothesis, X_1). There is one iteration where $|L| = 1$, $|\Sigma_C|$ iterations where $|L| = O(|\Sigma_C|)$, $|\Sigma_C|^2$ iterations where $|L| = O(|\Sigma_C|^2)$, and so on up to $|\Sigma_C|^l$ iterations where $|L| = O(|\Sigma_C|^l)$. The cost of the last tier dominates the cost of all the other tiers. The cost of this tier is $|\Sigma_C|^l$ times the cost of an iteration in which $|L| = |\Sigma_C|^l$. The cost of the growth stage is bounded by this cost, which is $O(|\Sigma_C|^{2l}t)$.

The maximum size to which L can grow is the number of selectors with which the initial single subset, X_1, can be extended. There are at most $O(ek)$ such selectors, as was shown in Section 5.4.1. Since the size of L is $|\Sigma_C|^l$, this term can be replaced with ek in $O(|\Sigma_C|^{2l}t)$. This yields the cost complexity bound for the growth stage shown in Equation 5.7. This cost includes pruning L back to a single element.

$$O((ek)^2t) \tag{5.7}$$

A Special Case. The dominance relation $D'_{\prec}$ is not entirely accurate, in that it allows a subset $w_1 \cdot q_1$ to be dominated by $w_2 \cdot q_2$ if q_2 is a dead state. This means a subset is dominated by an empty subset, which is not correct. This approximation is necessitated by the expense of determining whether q_2 is dead. Dominated subsets are removed from L, but since $D'_{\prec}$ is not entirely correct, some subsets may be removed because $D'_{\prec}$ thinks they are dominated by subsets that are in fact empty. This is mitigated by maintaining L in such a way that a dominated subset can be reinstated to L if the subsets that supposedly dominate it later turn out to be empty.

The selected subset, X_s, is empty if for each of its child subsets, $w_i \cdot q_i$, q_i is a recognizable dead state. In this case, X_s is marked as empty, and the subsets that X_s was thought to dominate are reinstated to L if they are not otherwise dominated. The emptiness of X_s is propagated to the supersets of X_s, since some of these may also be empty. If these are in fact empty, then the subsets they dominated may also be reinstated to L. All of the reinstated subsets are then compared to the other subsets of L as the children of X_s have been. The complexity of this step depends on the number of subsets dominated by X_s and its empty parent subsets.

However, X_s is never empty when RS-KII is emulating AQ-11. This is because all of the dead states in C are immediately recognizable. C is an intersection of constraint sets from each example. A constraint set for an example contains the hypotheses that are consistent with the example, and an intersection of two of these sets, $C_1 \cap C_2$, contains hypotheses consistent with both examples. A partial hypothesis, w, leads to a state $w \cdot \langle q_1, q_2 \rangle$ in the intersection. The state $\langle q_1, q_2 \rangle$ is recognizably dead if either q_1 or q_2 are dead—that is, there is no extension of hypothesis w that is consistent with one of the examples. The state is not recognizably dead if and only if there is one extension of w that is consistent with the first example, and a second extension of w that is consistent with the second example, but no extension that is consistent with both examples. This never happens, since any VL_1 hypothesis can be extended to be consistent with any set of mutually consistent examples. If the examples are not consistent, then C is immediately recognizable as empty.

Since X_s is never empty unless C is empty, none of the dominated subsets ever need to be reinstated. The cost of the reinstatement operation is therefore omitted from the computational complexity of RS-KII when emulating AQ-11.

Number of Iterations. The search algorithm makes one iteration for every selector in the induced hypothesis. The hypothesis induced by RS-KII is consistent with the positive and negative examples. Furthermore, each selector in the hypothesis makes progress towards consistency with the examples, and no selector or term is added unnecessarily (e.g., once a term is consistent with the negative examples, additional selectors are not appended to it). These latter conditions are guaranteed by the LEF. By the same arguments used in the analysis of AQ-11, the induced hypothesis contains at most one term per positive example, and each term contains at most one selector per negative example. If there are p positive examples, and n negative examples, the hypothesis contains at most pn selectors.

Total Computational Complexity. The search algorithm makes at most pn iterations. The cost to induce a hypothesis is therefore pn times the cost of the growth stage, in which all of the selectors are generated (Equation 5.7. This yields

the cost complexity for RS-KII shown in Equation 5.8 when RS-KII uses only the knowledge used by AQ-11.

$$O(pn(ek)^2t) \tag{5.8}$$

The cost of comparison, t, is proportional to the cost of computing the next-state function in P. A state in P is a vector of length e, so the cost of computing the next-state is $O(e)$. Substituting e for t, and e^2 for pn in Equation 5.8 yields the following computational complexity for RS-KII when emulating AQ-11:

$$O(e^5k^2) \tag{5.9}$$

The complexity of AQ-11 with a beam-size of one is $O(e^4k)$. The complexity of RS-KII is worse by a factor of ek. This comes from the comparisons between incomplete selectors performed during the growth stage. If the *Split* function is defined so that the string w in subset $\langle w, q \rangle$ is extended by a full selector instead of by a single symbol, then the complexity is $O(ke^4)$, the same as for AQ-11.

5.5 Summary

RS-KII can utilize all of the knowledge used by AQ-11 when the beam size is one, and thus functionally subsumes AQ-11 for this beam width. The complexity of RS-KII when utilizing only the knowledge of AQ-11 is a little worse than that of AQ-11, but still polynomial in the number of examples and features. RS-KII can also utilize a domain theory, which AQ-11 can not. RS-KII can utilize a domain theory at the same time it is using AQ-11 knowledge, effectively integrating this knowledge into AQ-11.

Chapter 6

RS-KII and IVSM

Incremental version space merging (IVSM) [Hirsh, 1990] is an extension of Mitchell's candidate elimination algorithm (CEA) [Mitchell, 1982] that can utilize noisy examples, domain theories, and other knowledge sources in addition to the the noise-free examples to which CEA is limited. IVSM strictly subsumes CEA. Each fragment of knowledge is translated into a version space of hypotheses that are consistent with the knowledge. The version spaces for each knowledge fragment are intersected, yielding a new version space consistent with all of the knowledge.

IVSM is exactly an instantiation of KII in which constraints are represented as version spaces, or *convex sets*, and preferences are represented by the null representation, which can express only the empty set (i.e., no preference information can be utilized). This instantiation is called CS-KII, for convex-set KII. The IVSM and CEA algorithms are described further in Section 6.1, and the equivalence of IVSM and CS-KII is demonstrated.

RS-KII can utilize the same knowledge as IVSM, and thereby subsume both IVSM and CEA, but only for some hypothesis spaces. IVSM expresses knowledge as a set of hypotheses that are consistent with the knowledge. For some hypothesis spaces, every subset of the hypothesis space that can be expressed as a convex set can also be expressed as a regular set. In these spaces, every knowledge fragment that can be expressed as a convex set in IVSM can be expressed as an equivalent regular set in RS-KII. These are the hypothesis spaces for which RS-KII subsumes IVSM. For other spaces, there are at least some subsets that can be expressed as a convex set but not as a regular set. For these spaces, RS-KII and IVSM overlap in the knowledge they can use, or the knowledge utilizable by IVSM and RS-KII is

disjoint. There are no non-trivial spaces for which IVSM subsumes RS-KII, since for every non-trivial hypothesis space there is at least one regular set that cannot be expressed as a convex set. The expressiveness of RS-KII and IVSM is investigated further in Section 6.2.1, and sufficient conditions are identified for hypotheses spaces for which RS-KII subsumes IVSM.

One class of hypothesis spaces for which RS-KII subsumes IVSM is the class of spaces described by conjunctive feature languages. These are the hypothesis spaces commonly used by CEA and IVSM. This class is defined more precisely in Section 6.2.2.1. The subsumption of IVSM by RS-KII for this class of spaces is investigated in this section as well. This includes both a general proof, and empirical support in the form of translators for a number of common knowledge sources that are also used by IVSM.

RS-KII also subsumes IVSM in terms of computational complexity for this class of hypothesis spaces. Section 6.3 compares the complexity of set operations in the regular and convex set representations for subsets of these hypothesis spaces. For these spaces, the complexity of RS-KII is up to a squared factor less than that of IVSM, and in all cases RS-KII's worst case complexity is bounded by IVSM's worst case complexity. For at least one space, there is a set of examples for which the complexity of IVSM is exponential in the number of examples, but for which the complexity of RS-KII is only polynomial. These examples are the well known examples demonstrated by Haussler [Haussler, 1988] to cause exponential behavior in CEA. The behavior of RS-KII for these examples is investigated in Section 6.4.

Concluding remarks are made in Section 6.5

6.1 The IVSM and CEA Algorithms

6.1.1 The Candidate Elimination Algorithm

The candidate elimination algorithm (CEA) [Mitchell, 1982] induces a hypothesis from positive and negative examples of the target concept. The implicit assumption of the algorithm is that the target concept is both a member of the hypothesis space and consistent with all of the examples. CEA maintains a set of hypotheses, called a *version space*, that consists of the hypotheses consistent with all of the examples

seen so far. Examples are processed incrementally. Each example is processed by removing hypotheses from the version space that are inconsistent with the example. If the example is positive, then all hypotheses not covering the example are removed. Conversely, a negative example is processed by removing the hypotheses that do cover the example.

If after processing an example the version space is empty, then the algorithm halts. No hypothesis in the hypothesis space is consistent with all of the examples, and the version space is said to have *collapsed.* If the version space contains only a single hypothesis, then the version space has *converged.* This single hypothesis is uniquely identified by all of the examples seen so far. If the implicit assumptions of CEA are correct, then this hypothesis is the target concept.

If there are multiple hypotheses in the version space, then it is assumed that one of them is the target concept. However, the target concept can not be discriminated from the other candidates without observing additional examples. If more examples are available, they can be processed in the hopes that the version space will converge. Otherwise, a hypothesis can be selected arbitrarily from the version space as the induced hypothesis, or the entire version space can be used to classify unseen instances as follows. If the instance is covered by all of the hypotheses in the version space, then the instance is also covered by the target concept and is classified as a positive instance. Likewise, if none of the hypotheses in the version space cover the example, neither does the target concept. The instance is classified as negative. Otherwise, the instance may or may not be covered by the target concept. In this case, the instance is classified as *unknown.*

6.1.2 Incremental Version Space Merging

Incremental version-space merging (IVSM) [Hirsh, 1990] is an extension of the candidate elimination algorithm that learns from non-example knowledge as well as from examples. Each knowledge fragment is translated into a constraint on the hypothesis space, and this constraint is represented as a version space of hypotheses that satisfy the constraint. The version spaces for each fragment are intersected to produce a single version space consistent with all of the knowledge. Every hypothesis in this

set satisfies all of the constraints, and is therefore equally preferred by the knowledge for being the target concept.

When the knowledge consists solely of noise-free examples, IVSM computes exactly the same version space as CEA, and with essentially the same time and space complexities [Hirsh, 1990]. That is, IVSM subsumes CEA.

A hypothesis can be selected arbitrarily from the final version space as the target concept, or instances can be classified against all of the hypotheses in the version space, as was described for CEA. Other queries about the version space are also possible. The queries that have generally proven useful are membership, emptiness (collapse), uniqueness (convergence), and subset [Hirsh, 1992].

6.1.3 Convex Set Representation

In both IVSM and CEA, the version space is represented as a convex set. A convex set consists of all hypotheses "between" two *boundary sets*, S and G, where S is the set of most specific hypotheses in the version space and G is the set of most general hypotheses in the version space. A hypothesis is in the version space if it is more general than or equivalent to some hypothesis in S, and is more specific than or equivalent to some hypothesis in G.

Formally, a convex set is specified by the tuple $\langle H, \prec, S, G \rangle$, where H is the hypothesis space, $\prec$ is a partial ordering of generality over H, and S and G are the boundary sets. When the values of H and $\prec$ are obvious, a convex set can be written as just $\langle S, G \rangle$. The convex set $\langle H, \prec, S, G \rangle$ consists of all hypotheses h such that $s \preceq h \preceq g$ for some s in S and g in G. The relation $\preceq$ is derived from $\prec$, such that $x \preceq y$ iff $x \prec y$ or $x = y$.

$$\langle H, \prec, S, G \rangle = \{h \in H \mid \exists s \in S \; \exists g \in G \; (s \preceq h \preceq g)\} \tag{6.1}$$

6.1.4 Equivalence of CS-KII and IVSM

IVSM essentially provides a collection of operations on convex sets: translation of knowledge into constraints expressed as convex sets, intersection of convex sets, enumeration of hypotheses from a convex set, and queries on convex sets such as

membership, emptiness, uniqueness and subset. These correspond to the operations provided by KII.

IVSM is equivalent to CS-KII, an instantiation of KII in which constraints are represented as convex sets, and preferences are represented by the null representation, which can represent only the empty set (i.e., preference information is not allowed). Both CS-KII and IVSM represent knowledge as convex sets, and provide the same operations on convex sets.

In IVSM, each knowledge fragment is translated into a a convex set containing all of the hypotheses that are consistent with the knowledge. The fragments are integrated by intersecting their corresponding convex sets into a single set containing the hypotheses consistent with all of the fragments. This set corresponds to the version space in CEA. A hypothesis can be selected from this set as the target concept, or instances can be classified against the entire set, in the same way that CEA classifies instances against the version space. IVSM also supports other operations on the convex set that represents all of the integrated knowledge, namely membership, emptiness (collapse), and uniqueness (convergence).

CS-KII represents knowledge in essentially the same way, and provides the same operations. In CS-KII, a knowledge fragment is translated into an $\langle H, C, P \rangle$ tuple, where H and C are convex sets, and P is the empty set. The representation for P can represent *only* the empty set, so P is always empty, regardless of the knowledge. The C set contains all of the hypotheses that are consistent with the knowledge fragment. This is exactly the same convex set that IVSM would use to represent the knowledge fragment. IVSM and CS-KII both represent a given knowledge fragment in terms of constraints on the hypothesis space, which is represented as a convex set of the hypotheses satisfying the constraints. Neither IVSM nor CS-KII can represent knowledge in terms of preference information. In CS-KII, this lack of knowledge is expressed explicitly by an empty P set. In IVSM, the lack of preference knowledge is implicit, so the P set can be omitted. The two representations are equivalent.

Knowledge is integrated in CS-KII by intersecting $\langle H, C, P \rangle$ tuples. The C sets of the pairs are intersected, and the P sets are unioned. Since the P sets are always empty, the union of two P sets is also empty. The result of integrating several knowledge fragments is $\langle H, C, \emptyset \rangle$, where C is the convex set containing the hypotheses consistent with all of the integrated knowledge fragments. This C set is

the the same convex set that IVSM would use to represent the same collection of knowledge fragments. Each fragment is represented by the same convex set in both IVSM and CS-KII, and both IVSM and CS-KII integrate knowledge by intersecting these convex sets.

Finally, CS-KII provides the same operations on convex sets as IVSM. In IVSM the queries are applied to the version space, and in CS-KII they are applied to the solution set of $\langle C, P \rangle$. Since P is always empty, the solution set of $\langle C, P \rangle$ is just C. This is the same convex set computed by IVSM. The queries in IVSM and CS-KII therefore both apply to C, the convex set of hypotheses consistent with all of the knowledge. The provided queries are membership, emptiness, and uniqueness. Both IVSM and CS-KII can also test whether C is a subset of another convex set. CS-KII also provides an explicit operator for enumerating hypotheses from C. IVSM does not provide an explicit enumeration operator, but its existence is implied by IVSM's ability to select an arbitrary hypothesis from the version space as the target concept. In both IVSM and CS-KII, hypotheses are enumerated from a convex set, so they can both use the same implementation of the enumeration operator.

IVSM provides one additional operation that CS-KII does not provide directly. This is the classification of instances against the entire version space (see Section 6.1.1). CS-KII does not provide this classification operation directly, but it can be implemented in terms of two operations that CS-KII already does provide: translation and the subset query. The implementation is shown in Figure 6.1. The idea is to translate the instance as if it were a positive example. This yields the set of hypotheses that cover the instance. If the version space (solution set) is a subset of this set, then every hypothesis in the version space covers the instance. The instance is assigned the `positive` classification. A similar procedure determines whether every hypothesis fails to cover the instance: translate the instance as if it were negative, and determine whether the solution set is a subset of the resulting set. If the version space is a subset of neither translation, then the instance is assigned the `unknown` classification.

CS-KII and IVSM are equivalent. Both represent knowledge as convex sets of hypotheses consistent with the knowledge, and both provide the same operations on this representation. These operations are implemented the same way in both

```
Algorithm Classify(i, ⟨C, P⟩)
    i: unclassified instance
    ⟨C, P⟩: a COP
BEGIN
    ⟨C+, P+⟩ ← TranExample(H, ⪯, ⟨i, positive⟩)
    ⟨C−, P−⟩ ← TranExample(H, ⪯, ⟨i, negative⟩)
    IF Subset(⟨C, P⟩, ⟨C+, P+⟩) THEN
        RETURN positive
    ELSE IF (Subset(⟨C, P⟩, ⟨C−, P−⟩)THEN
        RETURN negative
    ELSE
        RETURN unknown
    END IF
END Classify
```

Figure 6.1: Classify Instances Against the Version Space.

IVSM and CS-KII, except that CS-KII represents the lack of preference knowledge explicitly with an empty P set.

The representation for P can express *only* the empty set. One could imagine an instantiation of KII in which both C and P were represented as convex sets. However, convex sets are not closed under union [Hirsh, 1990], so integration would not be defined in this instantiation of KII.

The candidate elimination algorithm is subsumed by both IVSM and CS-KII. This follows from the equivalence of CS-KII and IVSM, and the fact that IVSM subsumes CEA [Hirsh, 1990]. One might ask whether AQ-11 was also subsumed by CS-KII and IVSM, at least to the extent that AQ-11 is subsumed by RS-KII (see Chapter 5). The answer is a qualified *no*. AQ-11 finds a VL_1 hypothesis that is strictly consistent with the examples and preferred by the LEF. The ability to express preference information is required in order to utilize the LEF, but CS-KII cannot utilize preference information. The only P set that can be expressed in CS-KII is the empty set. Extending the P representation to convex sets does not help much either, since convex sets are not closed under union, which makes it impossible to integrate $\langle C, P \rangle$ tuples.

However, CS-KII as it stands can at least find a VL_1 hypothesis consistent with the examples. The generality ordering over VL_1 hypotheses is well defined, so convex subsets of VL_1 can certainly be expressed. Each example is translated as a convex set of hypotheses consistent with the example, the convex sets are intersected, and a hypothesis is selected arbitrarily from the intersection as the target concept.

6.2 Subsumption of IVSM by RS-KII

RS-KII subsumes IVSM, but only for hypothesis spaces for which every convex subset of the space is also expressible as a regular set. In general, the convex set representation and the regular set representation overlap in expressiveness, but neither subsumes the other.

Section 6.2.1 discusses the relative expressiveness of the two representations in general terms, and identifies conditions for which every convex subset of hypothesis space can also be expressed as a regular set. These are the hypothesis spaces for which RS-KII subsumes IVSM.

A specific class of hypothesis spaces for which RS-KII subsumes IVSM is identified in Section 6.2.2.1. Since RS-KII subsumes IVSM for these spaces, it should be possible to construct RS-KII translators for the knowledge used by IVSM. Section 6.2.3 provides such translators as additional empirical support for the subsumption of IVSM by RS-KII.

6.2.1 Expressiveness of Regular and Convex Sets

Convex sets and regular sets overlap in expressiveness. There is some knowledge that can be expressed in both representations, and some that can be expressed in one representation but not the other.

The convex set representation places no restrictions on either the hypothesis space or the partial ordering of generality. The hypothesis space and the generality relation could require arbitrarily powerful Turing machines to represent them. This means that convex sets can express many sets that regular sets cannot, at least in theory. In practice, operations on convex sets—such as finding the most specific

common generalization of two hypotheses—may be computationally infeasible for sufficiently expressive hypothesis spaces or generality orderings.

Although the convex set representation is very expressive, it does not subsume the regular set representation. For every partial ordering having transitive chains of length three or more (e.g., $w \prec x \prec y$), there are regular subsets of the hypothesis space that cannot be expressed as convex sets with that ordering. For example, consider a set containing two elements, g and s, where g is a very general hypothesis, and s is a very specific hypothesis. If there is at least one hypothesis between these two hypotheses in the partial ordering, such that $s \prec x \prec g$, then the set $\{s, g\}$ cannot be represented by a convex set since every convex set containing s and g must also contain x. However, the set $\{s, g\}$ can be represented by a regular set. There are only two elements, and any finite set can be expressed as a regular set.

Another way of stating this phenomenon is that convex sets cannot have "holes", but regular sets can. A convex set must contain every hypothesis between the boundary sets. It cannot exclude hypotheses unless all of the remaining hypotheses are between some other pair of boundary sets. Regular sets, on the other hand, can represent sets with holes.

Whether a given subset of the hypothesis space has a "hole" depends on the generality ordering. Under one ordering, $\prec$, a given set may have a hole, but in ordering $\prec'$, the same set may not have a hole. However, there will be other subsets of the hypothesis space that do have holes in the $\prec'$ ordering. At least some of these can be expressed as regular sets.

6.2.2 Spaces for which RS-KII Subsumes CS-KII

For some hypothesis spaces, every convex subset of the space can be expressed as a regular set. These are the hypothesis spaces for which RS-KII subsumes CS-KII. A convex set, $\langle S, G, \prec, H \rangle$, contains all hypotheses that are more specific than or equal to some element of G, and more general than or equivalent to some element of S. S is the set of maximally specific elements in the set, and G is the set of maximally general elements, where every element of S is more specific than one or

more elements of G. A convex set $\langle S, G, \prec, H \rangle$ is therefore equivalent to the following set:

$$S \cup G \cup \bigcup_{s \in S, g \in G} (\{h \in H \mid s \prec h\} \cap \{h \in H \mid h \prec g\} \tag{6.2}$$

Every convex subset of H can be expressed as a regular grammar if for every hypothesis x in the hypothesis space, the sets $\{h \in H \mid x \prec h\}$ and $\{h \in H \mid h \prec x\}$ can be expressed as regular sets. This follows from closure of regular sets under intersection and union, and the finiteness of the S and G sets.

It is difficult to determine whether an arbitrary hypothesis space satisfies this condition, and is therefore a hypothesis space for which RS-KII subsumes CS-KII. However, there is a generalization that is easier to test, namely that both the hypothesis space (H) and the generalization ordering ($\prec$) can be expressed as regular sets. The set $\{h \in H \mid h \prec x\}$ is equivalent to $\mathit{first}((H \times \{x\}) \cap R_{\prec})$, where $R_{\prec}$ is a grammar encoding the generality ordering $\prec$. That is, $R_{\prec} = \{\langle x, y \rangle \in H \times H \mid x \prec y\}$. Since regular grammars are closed under intersection, Cartesian product, and projection, $\{h \in H \mid h \prec x\}$ is expressible as a regular set for every hypothesis x. Similarly, the set $\{h \in H \mid x \prec h\}$ is equivalent to $\mathit{second}((\{x\} \times H) \cap R_{\prec})$, and therefore expressible as a regular grammar as long as both H and $R_{\prec}$ are expressible as regular grammars.

6.2.2.1 Conjunctive Feature Languages

One class of languages for which RS-KII subsumes CS-KII are a subset of the conjunctive feature languages. A hypothesis in a conjunctive feature language is a conjunct of feature values, with one value for each of the features. For example, in a language with k features, hypotheses are of the form $\langle V_1, V_2, \ldots, V_k \rangle$ where V_i is an *abstract value* corresponding to a set of *ground values*. An instance is a vector of ground values, $\langle v_1, v_2, \ldots, v_k \rangle$. An instance is covered by a hypothesis if $v_i \in V_i$ for every i between one and k. If the "single representation trick" is used (e.g., [Dietterich *et al.*, 1982]), there is an abstract value, v, for each ground value, v.

Feature values consist of both ground values and abstract values. The ground values are the specific values that the feature can take, and the abstract values correspond to sets of ground values. For example, the feature `color` may have ground values such as *green* and *red*, and abstract values such as *dark-color* and

light-color. The abstract values correspond to sets of ground values. Features may be continuously valued or discrete. If the ground values are continuously valued (e.g., the real numbers) then the abstract values are sets of ground values, such as the set of real numbers between one and five, inclusive.

The abstract values for each feature can be partially ordered by generality. Value V_1 is more general than value V_2 if V_2 is a subset of V_1. If neither value is a subset of the other, then there is no generality ordering between them. The generality ordering over the abstract features is called a *generalization tree*. A value in the tree is more general than its descendents. Typically, the leaves of the tree are abstract values that cover only a single ground value.

The conjunctive feature languages are a family of languages. The family is parameterized by the number of features, and a generalization tree for each of the features. A generalization tree is a tuple $\langle F, \prec \rangle$, where F is a set of abstract values, and $x \prec y$ means that abstract value x is more specific than abstract value y.

The grammar for the language is a concatenation of the grammars for the values of each feature. $G(F_i)$ is the grammar for the abstract values in feature F_i.

$$\mathit{ConjunctiveFeatureLanguage}(\langle F_1, \prec_1 \rangle, \ldots, \langle F_k, \prec_k \rangle) \rightarrow G(F_1) \cdot G(F_2) \cdot \ldots \cdot G(F_k) \tag{6.3}$$

The generality relation among hypotheses is derived from the relations on individual features. A hypothesis, $\langle x_1, x_2, \ldots, x_k \rangle$, is more general than or equal to $\langle y_1, y_2, \ldots, y_k \rangle$ if $y_i \preceq_i x_i$ for every i between one and k. Hypothesis $\langle x_1, x_2, \ldots, x_k \rangle$ is strictly more general than $\langle y_1, y_2, \ldots, y_k \rangle$ if there is at least one feature, f_i, such that $y_i \prec_i x_i$, and for the remaining features, $f_{j \neq i}$, $y_j \preceq_j x_j$. The strict relation is stated formally in Equation 6.4, and $\preceq$ is defined formally in Equation 6.5.

$$\langle x_1, x_2, \ldots, x_k \rangle \prec \langle y_1, y_2, \ldots, y_k \rangle \text{ iff}$$
$$(\exists i \in \{1, \ldots, k\}\ x_i \prec_i y_i) \text{ and } (\forall j \in \{1, \ldots, k\}\ x_i \preceq_i y_i) \tag{6.4}$$

$$\langle x_1, x_2, \ldots, x_k \rangle \preceq \langle y_1, y_2, \ldots, y_k \rangle \text{ iff } \forall j \in \{1, \ldots, k\}\ x_i \preceq_i y_i \tag{6.5}$$

6.2.2.2 Conjunctive Languages where RS-KII Subsumes CS-KII

RS-KII subsumes CS-KII for hypothesis spaces in which both the hypothesis space and the generality ordering over the hypotheses can be expressed as regular grammars. This is true of conjunctive feature languages in which the values for each feature can be expressed as a regular grammar, and the generalization tree for each feature can be expressed as a regular grammar. The grammar for the hypothesis space is the concatenation of the grammars for each feature.

The grammar for the generality ordering, $\prec$, over hypotheses is defined as follows. Let $R_{\prec,i}$ be the regular set $\{\langle x, y\rangle \in F_i \times F_i \mid x \prec_i y\}$, which represents $\prec_i$, the generalization hierarchy for feature f_i. Let $R_{\preceq}, i$ be the regular set $\{\langle x, y\rangle \in F_i \times F_i \mid x \preceq_i y\}$, where $x \preceq_i y$ if $x \prec_i y$ or $x = y$. This is the union of $R_{\prec,i}$ and $R_{=,i}$, where $R_{=,i} = \{ww \mid w \in F_i\}$. The generalization hierarchy, $\prec$, over hypotheses is constructed from the regular sets for the individual feature hierarchies as follows:

$$R_{\prec} = \bigcup_{i=1}^{k} R_{\preceq,1} \cdot \ldots \cdot R_{\preceq,i-1} \cdot R_{\prec,i} \cdot R_{\preceq,i+1} \cdot \ldots \cdot R_{\preceq,k} \tag{6.6}$$

If both the hypothesis space and the generality orderings for each feature, $R_{\prec,i}$, and $R_{\preceq,i}$, can be expressed as regular grammars, then every convex subset of the hypothesis space is expressible as a regular set. When this is true, any knowledge that can be expressed by CS-KII can also be expressed by RS-KII.

What generality orderings over features can be expressed as regular grammars? A generality ordering, $R_{\prec,i}$ or $R_{\preceq,i}$, is a subset of $F_i \times F_i$. If F_i can be expressed as a regular grammar, then so can $F_i \times F_i$, since regular grammars are closed under Cartesian product. However, not every subset of $F_i \times F_i$ can be expressed as a regular grammar (see Section 4.1.2.4), so not all generality orderings over F_i are expressible as regular grammars.

It is difficult to specify necessary and sufficient conditions for which a given subset of $F_i \times F_i$ can be expressed as a regular grammar. However, it is possible to identify some sufficient conditions. For example, any finite set can be expressed as a regular grammar, so any generality ordering over F_i can be expressed as a regular grammar if F_i is finite. Likewise, the equality relation, $R_{=,i}$, can also be expressed as a regular grammar if F_i is finite.

When F_i is infinite, some but not all orderings can be expressed. $R_{=,i}$ can always be expressed as a regular set under the shuffle mapping if F_i is a regular set (see Section 4.1.2.4). However, only some generality orderings can be expressed as regular grammars. Among the subsets of $F_i \times F_i$ that can be expressed as regular grammars under the shuffle mapping are those of the form $\{\langle x, y\rangle \in F_i \times F_i \mid x < y\}$ where $<$ is a lexicographic (dictionary) ordering. This provides a way to represent total orderings over F.

In one common generality ordering over features with ordinal values (e.g., integers), each abstract value is a range of values, and one abstract value is more general than another if its range properly contains the other. For example, a range might be of the form $[x..y]$, where $x < y$, and one range would be more general than another if the first range contained the second (e.g., $[5..10] \prec [0..20]$).

To represent the set of abstract values, F_i, as a regular grammar, the range $[x..y]$ is represented as the string $(x \circ y)$ where and $\circ$ is a special symbol that indicates the end of x and the beginning of y. F_i is the set of all possible ranges, $\{x \circ y \mid x, y \in \text{INT}$ and $\leq y\}$ where INT is a regular expression for the set of integers, $(0-9)|1(0-9)^+$. F_i can be expressed as the regular set $\{\mathit{shuffle}(x, y) \mid x, y \in \text{INT}\}$.

The generality ordering over F_i is $\{\langle [x_1..y_1], [x_2, y_2]\rangle \in F_i \times F_i \mid x_2 \leq x_1$ and $y_1 \leq y_2\}$. This is the set of all pairs of ranges such that the first range is contained in the second. The regular set for this ordering takes as input strings of the form $\mathit{shuffle}(w_1, w_2)$, where w_1 and w_2 are strings encoding ranges in F_i. A range in F_i is encoded as a string of the form $\mathit{shuffle}(x, y)$ where x and y are ground values for the feature, and $x \leq y$. An input to the grammar is therefore an interleaving of four strings, a, b, c and d, where the input is accepted if the range $[a..b]$ is contained within the range $[c..d]$, and both of these are valid ranges (i.e, $a \leq b$ and $c \leq d$).

The grammar for the generality ordering, $\preceq$, over F_i is constructed from four simpler grammars Create four copies of a regular grammar that recognizes the set $\{\mathit{shuffle}(x, y) \mid x, y \in \text{INT}$ and $x \leq y\}$. This is a simple modification of the DFA described in Section 4.1.2.4 of Chapter 4. Call these grammars M_1 through M_4. For each quartet of symbols in the input string, $a_i b_i c_i d_i$, pass $a_i b_i$ to M_1, $c_i d_i$ to M_2, $c_i a_i$ to M_3, and $b_i d_i$ to M_4. The machines M_1 and M_2 ensure that the ranges $[a..b]$ and $[c..d]$ are valid, in that $a \leq b$ and $c \leq d$. The machines M_3 and M_4 ensure that $[a..b]$

is contained within $[c..d]$—that is, $c \leq a$ and $b \leq d$. If all four machines accept their input, the input string is accepted by the composite regular grammar.

6.2.3 RS-KII Translators for IVSM Knowledge

RS-KII and CS-KII (IVSM) overlap in expressiveness. Some knowledge can be utilized by both RS-KII and CS-KII, and some knowledge can be used by one but not the other. For some hypothesis spaces, RS-KII subsumes CS-KII—that is, all knowledge that can be utilized by CS-KII can also be utilized by RS-KII. This is true of hypothesis spaces described by conjunctive feature languages, where the generality ordering for each feature can be expressed as a regular grammar.

As empirical support for this fact, RS-KII translators will be demonstrated for a number of knowledge sources used by IVSM. The hypothesis space for all of these translators is assumed to consist of conjuncts of feature values, where the generalization ordering for each feature can be expressed as a regular grammar.

These translators generally take as input both the hypothesis space (H) and the generality ordering ($R_{\preceq}$). $R_{\preceq}$ is the union of $R_{\prec}$ and $R_{=}$, and therefore a regular set whenever $R_{\prec}$ and $R_{=}$ are regular. The ordering $R_{\preceq}$ is used in place of $R_{\prec}$ in the translators because it is easier to construct the necessary C sets from operations on $R_{\preceq}$ than it is to construct them from set operations on $R_{\prec}$. The P sets produced by the translators are always empty, since the set representation that CS-KII uses for P can only express the empty set.

A translator for noise-free examples is described in Section 6.2.3.1, a translator for noisy examples with bounded inconsistency [Hirsh, 1990] is described in Section 6.2.3.2, and a translator for overgeneral domain theories is described in Section 6.2.3.3.

6.2.3.1 Noise-free Examples

The target concept is consistent with all of the noise-free examples, by definition. That is, the target concept covers all of the positive examples and none of the negative examples. A hypothesis that is not consistent with the noise-free examples cannot be the target concept.

A noise-free example is translated as a constraint that is satisfied by all of the hypotheses consistent with the example. A positive example is translated as $\langle C, \emptyset \rangle$, where C is the set of hypotheses that cover the example. Similarly, a negative example is translated as $\langle C, \emptyset \rangle$, where C is the set of hypotheses that do not cover the example.

For the conjunctive feature languages, an instance $\langle x_1, x_2, \ldots, x_k \rangle$ is covered by a hypothesis, $\langle v_1, v_2, \ldots, v_k \rangle$, if and only if for every feature f_i from one to k, $x_i \preceq_i v_i$.

The set of hypotheses covering an instance can be computed as follows. Let F_i be the set of abstract values that feature i can take. Let $R_{\preceq,i}$ be a regular grammar encoding the generality ordering over F_i: $R_{\preceq,i} = \{\langle x, y \rangle \in H \times H \mid x \preceq_i y\}$. The set of features values covering x_i is given by the following formula:

$$\{v \in F_i \mid x_i \preceq_i v\} = \mathit{first}(R_{\preceq,i} \cap (\{x_i\} \times F_i)) \tag{6.7}$$

This set is expressible as a regular grammar if x_i, $R_{\preceq}$, and F_i are regular sets, since regular sets are closed under union, intersection, and projection.

The set of hypotheses covering the instance is the Cartesian product of these sets. The regular grammar for this set is the concatenation of the regular grammars for the set of values covering the instance on each feature. A translator for noise-free examples based on this construction is given in Figure 6.2.3.1. This translator takes as input the hypothesis space, expressed as a list of value sets for each feature, F_1 through F_k; the generality ordering $R_{\preceq}$, expressed as a list of generality orderings for each feature, $R_{\preceq,1}$ through $R_{\preceq,k}$; and an example of the form $\langle v_1, \ldots, v_k, c \rangle$, where v_i is a value for feature f_i and c is a classification (e.g., positive or negative).

6.2.3.2 Noisy Examples with Bounded Inconsistency

Bounded inconsistency [Hirsh, 1990] is a kind of noise in which each feature of the example can be wrong by at most a fixed amount. For example, if the width value for each instance is measured by an instrument with a maximum error of ± 0.3mm, then the width values for these instances have bounded inconsistency.

The idea for translating examples with bounded inconsistency is to use the error margin to work backwards from the noisy example to compute the set of possible noise-free examples. One of these examples is the correct noise-free version of the

$$
\begin{aligned}
&\mathit{TranExample}(F_1, F_2, \ldots, F_k, \\
&\qquad R_{\preceq,1}, R_{\preceq,2}, \ldots, R_{\preceq,k} \\
&\qquad \langle\langle x_1, x_2, \ldots, x_k\rangle, \text{class}\rangle) \rightarrow \langle C, \emptyset\rangle
\end{aligned}
$$

where

$F_1 \times F_2 \times \ldots \times F_k$ is the hypothesis space

$R_{\preceq,1} \cdot R_{\preceq,2} \cdot \ldots \cdot R_{\preceq,k}$ is $R_{\preceq}$

$\langle\langle x_1, x_2, \ldots, x_k\rangle, \text{class}\rangle$ is an example

$$
C = \begin{cases} \overline{V_1 \times V_2 \times \ldots \times V_k} & \text{if class} = \texttt{negative} \\ V_1 \times V_2 \times \ldots \times V_k & \text{if class} = \texttt{positive} \end{cases}
$$

$$
V_i = \{v \in F_i \mid v \preceq x_i\} = \mathit{first}(R_{\preceq,i} \cap (\{x_i\} \times F_i))
$$

Figure 6.2: RS-KII Translator for Noise-free Examples.

observed example, into which noise was introduced to produce the observed noisy example. The target concept is strictly consistent with this noise-free example.

Let e be the noisy observed example, E be the set of noise-free examples from which e could have been generated, and let e' be the correct noise-free example from which e was in fact generated. Ideally, e is translated into $\langle C, \emptyset\rangle$, where C is the set of hypotheses consistent with e'. However, it is unknown which example in E is e'. Because of this uncertainty, a noisy example is translated as $\langle C, \emptyset\rangle$, where C is the set of hypotheses that are strictly consistent with at least one of the examples in E. Hypotheses that are consistent with none of the examples in E are also not consistent with e', and therefore not the target concept. This is the approach used by Hirsh [Hirsh, 1990] in IVSM to translate noisy examples with bounded inconsistency.

This suggests the following RS-KII translator for examples with bounded inconsistency. The set of possible noise-free examples, E, is computed from the noisy examples and the error margins for each feature. Each example, e_i, in this set is translated using the RS-KII translator for noise-free examples, $\mathit{TranExample}(H, R_{\preceq}, e_i)$ (Figure 6.2.3.1), which translates example e_i into $\langle C_i, \emptyset\rangle$. C_i is the set of hypotheses that are strictly consistent with e_i. The translator for the bounded inconsistent example returns $\langle C = \bigcup_{i=1}^{|E|} C_i, \emptyset\rangle$. C is the set of hypotheses consistent with at least one of the examples in E.

$$TranExampleBI(H, R_{\preceq}, \langle \delta_1, \delta_2, \ldots, \delta_k \rangle, \langle \langle x_1, x_2, \ldots, x_k \rangle, \text{class} \rangle) \rightarrow \langle C, P \rangle$$

where

$$\begin{aligned} E &= [x_1, \pm\delta_1] \times [x_2, \pm\delta_2] \times \ldots [x_k, \pm\delta_k] \times \{\text{class}\} \\ &\quad \text{where } [x_i, \pm\delta_i] = \{v \mid x_i - \delta_i \leq v \leq x_i + \delta_i\} \\ C &= \bigcup_{e_i \in E} C_i \text{ s.t. } \langle C_i, \emptyset \rangle = TranExample(H, R_{\preceq}, e_i) \end{aligned}$$

Figure 6.3: RS-KII Translator for Examples with Bounded Inconsistency.

The set E is computed from the observed example, $\langle x_1, x_2, \ldots, x_k, \text{class} \rangle$, and the error margins for each feature, $\pm\delta_1$ through $\pm\delta_k$, as follows. If the observed value for feature i is x_i, and the error margin is $\pm\delta_i$, then the correct value for feature i is in $\{v \mid x_i - \delta_i \leq v \leq x_i + \delta_i\}$. Call this set $[x_i, \pm\delta_i]$ for short. Since examples are ordered lists of feature values and a class, E is $[x_1, \pm\delta_1] \times [x_2, \pm\delta_2] \times \ldots \times [x_k, \pm\delta_k] \times \{\text{class}\}$.

A translator for examples with bounded inconsistency based on this approach is shown in Figure 6.2.3.2. It takes as input the hypothesis space (H), a regular grammar encoding the generality ordering ($R_{\preceq}$), the error margin for each feature ($\pm\delta_1$ through $\pm\delta_k$), and an example. The hypothesis space input is represented in factored form as an ordered list of regular sets, F_1 through F_k, where F_i is the set of abstract values for feature f_i. The input $R_{\preceq}$ is also represented in factored form as an ordered list of regular sets, $R_{\preceq,1}$ through $R_{\preceq,k}$, where $R_{\preceq,i}$ represents the generality ordering for feature f_i. The factored representation is used since the translator makes a call to $TranExample(H, R_{\preceq}, e)$, which expects H and $R_{\preceq}$ to be represented in this fashion.

6.2.3.3 Domain Theory

A domain theory is a set of rules that explain why an instance is a member of the target concept. For instance, a particular object is a cup because it is liftable, and has a stable bottom and an open top. There are many ways to use a domain theory, depending on assumptions about the completeness and correctness of the theory.

For example, if the theory is overspecial, then there are instances which cannot be explained as members of the target concept. The target concept is a generalization of the domain theory. The theory provides an initial guess at the target concept, and additional knowledge indicates which generalization of the theory has the most support for being the target concept. Other assumptions about the correctness and completeness of the theory are also possible (see Section 6.2.3.3). For the sake of brevity, only a translator for overspecial theories will be described.

If the hypothesis is overspecial, then the target concept is some generalization of the domain theory. The theory could therefore be translated as a constraint satisfied only by generalizations of the theory. Determining which hypotheses are generalizations of the theory is somewhat difficult. One approach is to map the theory onto an equivalent hypothesis, and then use the generalization ordering to determine the generalizations. However, it is not clear how to perform this mapping.

A second approach is to translate both the theory and a positive example at the same time. An explanation of the example by the theory corresponds to a sufficient condition for the concept described by the theory. The target concept is a generalization of the explanation, and an an explanation can be easily mapped onto the hypothesis space. The explanation is a conjunction of operational predicates, or features, and exactly corresponds to a member of the hypothesis space. Once the mapping has been done, it is a simple matter to compute the set of hypotheses more general than the explanation.

Only a positive example can be used for this translation, since the theory can explain why an instance is a member of the target concept, and therefore a positive example, but cannot explain why an instance is not a member of the target concept. To utilize negative examples in this way, a theory is needed for the complement of the target concept. Complementary theories for positive and negative examples are also used in IVSM [Hirsh, 1990].

A translator for an overspecific domain theory is shown in Figure 6.2.3.3. In this translator, H is the hypothesis space, $R_{\preceq}$ is a regular grammar for the generalization ordering, T is a domain theory, and e is a positive example. The explanation of an example by the theory is essentially a generalized example. This example is passed to the translator for noise-free examples described in Section 6.2.3.1.

$$\mathit{TranOverSpecialDT}(H, R_{\preceq}, T, e = \langle \text{inst,positive} \rangle) \rightarrow \langle C, P \rangle \text{ where}$$
$$\langle C, P \rangle = \mathit{TranExample}(H, R_{\preceq}, \langle \mathit{explain}(\text{inst}, T), \text{positive} \rangle)$$

Figure 6.4: RS-KII Translator for an Overspecial Domain Theory.

6.3 Complexity of Set Operations

The computational complexity of IVSM (CS-KII) and RS-KII are determined by the complexity of integrating knowledge, and the complexity of enumerating hypotheses from the solution set. The complexity of these operations is determined in turn by the computational complexity of the set operations that define them.

When the P set is empty, the only set operation that determines the computational complexity of integration is intersection. Integration involves intersecting the C sets and computing the union of the P sets, but since the P sets are empty, their union is always empty. The union operation can be computed in constant time, and is dominated by the complexity of intersecting the C sets. Since P is empty, the deductive closure of $\langle C, \emptyset \rangle$ is just C. The complexity of enumerating a hypothesis from the deductive closure of $\langle C, P \rangle$ therefore depends on the size of the DFA for C.

In CS-KII, the P set is always empty since only the empty set is expressible in the representation for P. Therefore, the costs of integration and enumeration in CS-KII depend on the cost of intersecting convex sets and the cost of enumerating hypotheses from an intersection of convex sets, respectively. In RS-KII, the P set can be non-empty. However, when only the knowledge expressible in CS-KII is utilized, the P set is always empty. The cost of integration and enumeration in CS-KII are determined by the costs of intersecting regular sets and the cost of enumerating hypotheses from an intersection of regular sets.

The computational complexity of intersection and enumeration for regular sets is derived in Section 6.3.1 and Section 6.3.2, respectively. The computational complexity of intersection and enumeration for convex sets is derived in Section 6.3.3 and Section 6.3.4. A comparison between the complexity equations for regular and convex sets is made in Section 6.3.5.

6.3.1 Complexity of Regular Set Intersection

Recall that the intersection of two DFAs is implemented in RS-KII by constructing an intentional DFA as shown in Figure 6.5. Constructing this DFA only requires pointers to the two original DFAs, and definitions of the DFAs components in terms of calls to similar components in the the two original DFAs. The intentional DFA for the intersection of two DFAs can be constructed in constant time.

$$\langle s_1, \delta_1, F_1, Dead_1, AccAll_1, \Sigma_1 \rangle \cap \langle s_2, \delta_2, F_2, Dead_2, AccAll_2, \Sigma_2 \rangle = \langle s, \delta, F, Dead, AcceptAll, \Sigma \rangle \text{ where}$$

$$\begin{aligned}
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F(\langle q_1, q_2 \rangle) &= F_1(q_1) \wedge F_2(q_2) \\
Dead(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true if} & Dead_1(q_1) = \texttt{true or} \\ & Dead_2(q_2) = \texttt{true} \\ \texttt{else unknown} & \end{cases} \\
AcceptAll(\langle q_1, q_2 \rangle) &= \begin{cases} \texttt{true if} & AccAll_1(q_1) = \texttt{true and} \\ & AccAll_2(q_2) = \texttt{true} \\ \texttt{false if} & AccAll_1(q_1) = \texttt{false or} \\ & AccAll_2(q_2) = \texttt{false} \\ \texttt{else unknown} & \end{cases} \\
\Sigma &= \begin{cases} \Sigma_1 \cap \Sigma_2 & \text{if } \Sigma_1 \neq \Sigma_2 \\ \text{Pointer to } \Sigma_1 & \text{if } \Sigma_1 = \Sigma_2 \end{cases}
\end{aligned}$$

Figure 6.5: Intersection Implementation.

6.3.2 Complexity of Regular Set Enumeration

Although constructing an implicit DFA for the intersection of two DFAs is a constant time operation, enumerating hypotheses from this DFA is not. A hypothesis is enumerated by finding a path from the start state to an accept state. This can take both time and space proportional to the number of states in the DFA.

The implicit DFA for the intersection represents the explicit DFA shown in Figure 6.6. The implicit DFA generates states and edges in the explicit DFA as they

are needed by the search. Thus the space cost is bounded by the number of states actually searched instead of by the total number of states in the explicit DFA. This saves space on average, but the worst case complexities are still the same.

$$
\begin{aligned}
\langle Q, s, \delta, F, \Sigma \rangle &= \langle Q_1, s_1, \delta_1, F_1, \Sigma \rangle \cap \langle Q_2, s_2, \delta_2, F_2, \Sigma \rangle \\
\text{where} & \\
Q &= Q_1 \times Q_2 \\
s &= \langle s_1, s_2 \rangle \\
\delta(\langle q_1, q_2 \rangle, \sigma) &= \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle \\
F &= F_1 \times F_2
\end{aligned}
$$

Figure 6.6: Explicit DFA for the Intersection of Two DFAs.

The cost of enumerating a hypothesis from this DFA is proportional to the number of states in the DFA, and the cost of computing the next-state function, δ. The explicit DFA has $|Q_1||Q_2|$ states, and the cost of computing δ is $cost(\delta_1) + cost(\delta_2)$. The time and space needed to enumerate a hypothesis from the intersection of two DFAs is bounded by Equation 6.8.

$$O(|Q_1||Q_2|[cost(\delta_1) + cost(\delta_2)]) \tag{6.8}$$

6.3.3 Complexity of Convex Set Intersection

The intersection of two convex sets is computed in two phases. In the first phase, a convex set is computed that represents the intersection of the two sets, but whose boundary sets contain extraneous hypotheses. In the second phase, these hypotheses are eliminated from the boundary sets, yielding a minimal convex set. This analysis follows [Hirsh, 1990].

Phase One. The non-minimal intersection of two convex sets, $\langle H, \prec, S_1, G_1 \rangle$ and $\langle H, \prec, S_2, G_2 \rangle$, is defined in Figure 6.7. In this definition, $LUB(a, b, \prec)$ returns the least upper bounds of a and b in the partial order $\prec$, and $GLB(a, b, \prec)$ returns the

$$\langle S, G, \prec, H\rangle = \langle S_1, G_1, \prec, H\rangle \cap \langle S_2, G_2, H, \prec\rangle \text{ where}$$
$$S = \{s \mid \exists_{s_1 \in S_1, s_2 \in S_2} \text{ s.t. } s \in LUB(s1, s2, \prec)\}$$
$$G = \{g \mid \exists_{g_1 \in G_1, g_2 \in G_2} \text{ s.t. } g \in GLB(g1, g2, \prec)\}$$

Figure 6.7: Intersection of Convex Sets, Phase One.

greatest lower bounds of a and b in $\prec$. That is, $LUB(a, b, \prec)$ returns the most specific common generalizations of a and b, and $GLB(a, b, \prec)$ returns the most general common specializations of a and b.

Computing the non-minimal intersection involves finding $LUB(a, b, \prec)$ for every pair of hypotheses $\langle a, b\rangle$ in $S_1 \times S_2$, and $GLB(x, y, \prec)$ for every pair of hypotheses $\langle x, y\rangle$ in $G_1 \times G_2$. There are $|S_1||S_2| + |G_1||G_2|$ such pairs. The cost of computing the non-minimal intersection is proportional to the number of pairs and the cost of computing the GLBs and LUBs. Assuming that the cost of computing the LUB of two hypotheses is t_{lub} and the cost of computing the GLB is t_{glb}, the cost of computing the non-minimal intersection is given by the Equation 6.9.

$$|S_1||S_2|t_{lub} + |G_1||G_2|t_{glb} \tag{6.9}$$

Phase Two. In the second phase, the boundary sets are minimized by removing from S hypotheses that are more general than other elements of S, or that are not more specific than some element of G. The first test requires $|S|^2$ comparisons. Removing elements of S that are not more specific than or equivalent to some element of G requires $|S|(|G_1| + |G_2|)$ comparisons. This makes use of the observation that since G contains the GLBs of every pair of elements in G_1 and G_2, an element $s \in S$ is more specific than some element of G if and only if s is more specific than an element of G_1 and an element of G_2. The cost of minimizing S is therefore $[|S|^2 + |S|(|G_1| + |G_2|)]t_<$, where $t_<$ is the cost of comparing two hypotheses to determine which is more general.

S contains the LUBs of all pairs of elements from S_1 and S_2, so $|S| \leq |S_1||S_2|$. The cost of minimizing the S set is therefore bounded by $[(|S_1||S_2|)^2 + |S_1||S_2|(|G_1| +$

$|G_2|)]t_<$. A symmetric analysis holds for minimizing G. The total cost of minimizing the intersection of two convex sets is given by Equation 6.10.

$$[(|S_1||S_2|)^2 + |S_1||S_2|(|G_1| + |G_2|) + (|G_1||G_2|)^2 + |G_1||G_2|(|S_1| + |S_2|)]t_< \quad (6.10)$$

Total Cost. The cost of intersecting two convex sets, is the sum of Equation 6.9 and Equation 6.10. The values of $t_<$, t_{lub}, and t_{glb}, depends on both the hypothesis space and the generality ordering ($\prec$). The time cost of computing the intersection of two convex sets, $\langle S_1, G_1\rangle$ and $\langle S_2, G_2\rangle$, is bounded by Equation 6.11:

$$\begin{aligned} &|S_1||S_2|t_{lub} + |G_1||G_2|t_{glb} + \\ &\quad [(|S_1||S_2|)^2 + |S_1||S_2|(|G_1| + |G_2|) + \\ &\quad (|G_1||G_2|)^2 + |G_1||G_2|(|S_1| + |S_2|)]t_< \end{aligned} \quad (6.11)$$

6.3.4 Complexity of Enumerating Convex Sets

Enumerating a hypothesis from an intersection of convex sets is inexpensive. If only one or two hypotheses are needed, the first hypotheses in the S and G sets can be returned. These are both constant time operations. This is sufficient to induce a hypothesis, and to answer the emptiness and uniqueness queries.

If additional hypotheses are needed, they can be enumerated by starting with an element of the S set and climbing the generalization tree. This is done by generating the immediate parents of the hypothesis in the generalization tree, and enumerating only those parents that are also covered by at least one G set element. The complexity of generating an immediate parent depends on the representation of the generality ordering. For conjunctive languages with tree-structured and lattice-structure features, it is a constant time operation. Verifying that a hypothesis is covered by at least one G set element takes time proportional to $|G|t_<$. The time to generate each element is therefore $O(|G|t_<)$, at least for conjunctive languages with tree or lattice-structured features.

6.3.5 Complexity Comparison

The intersection and enumeration costs for convex sets and regular sets are in terms of entirely different quantities. The costs for convex sets are in terms of the S and G set sizes, whereas the costs for regular sets are in terms of the number of states in the DFA. In order to compare these costs we need to know the relationship between these different quantities for equivalent sets.

It is difficult to find a general equation relating these quantities, but it is possible to relate them when the sets are assumed to be subsets of a specific hypothesis space. The idea is to devise an algorithm for translating an arbitrary convex subset of the hypothesis space to an equivalent regular set, such that the size of the resulting regular set can be expressed in terms of $|S|$ and $|G|$. This provides the necessary relation between the sizes of equivalent regular and convex subset of the hypothesis space. This relation is only valid for the given hypothesis space, or class of hypothesis spaces, and the approach requires that every convex subset of the hypothesis space can be expressed as an equivalent regular set.

The computational complexities of CS-KII and RS-KII will be compared using this approach for the class of hypothesis spaces described by a conjunctive language with tree-structured features, where the generalization tree has a fixed depth bound. A similar comparison will be made for conjunctive languages in which the features are lattice structured, where the generalization tree for each feature has a fixed depth bound. These hypothesis spaces are commonly used with IVSM and CEA, and every convex subset of these hypothesis spaces can be expressed as regular set (see Section 6.2.2.1).

6.3.5.1 Equating Regular and Convex sets

A hypothesis in a conjunctive language can be written as a vector of feature values, $\langle v_1, v_2, \ldots, v_k \rangle$, where v_i is one of the values in the generalization hierarchy for feature f_i. A hypothesis is generalized by replacing at least one feature value, v_i, with a more general value. A hypothesis is specialized by replacing at least one feature value with a more specific value. The generality ordering over the hypotheses has a maximally general hypothesis, U, that classifies every instance as positive, and a maximally specific element, $\emptyset$, that classifies every instance as negative.

A convex set in this language can be written as shown in Equation 6.12, where $x \preceq y$ if x and y are hypotheses in the language and x is either more specific than y, or $x = y$.

$$\begin{aligned} \langle S, G \rangle &= \{h \mid \exists s \in S \, \exists g \in G \; s \preceq h \preceq g\} \\ &= \bigcup_{s \in S} \{h \mid s \preceq h \preceq U\} \cap \bigcup_{g \in G} \{h \mid \emptyset \preceq h \preceq g\} \end{aligned} \tag{6.12}$$

In a conjunctive language, a hypothesis can be written as a vector of feature values, $\langle v_1, v_2, \ldots, v_k \rangle$. Equation 6.12 can therefore be rewritten as shown in Equation 6.13. In this definition, $f_i(h)$ is the value of hypothesis h on feature f_i.

$$\begin{aligned} \langle S, G \rangle &= \bigcup_{s \in S} \{\langle v_1, v_2, \ldots, v_k \rangle \mid f_1(s) \preceq v_1 \preceq f_1(U) \wedge \ldots \wedge f_k(s) \preceq v_k \preceq f_k(U)\} \cap \\ &\quad \bigcup_{g \in G} \{\langle v_1, v_2, \ldots, v_k \rangle \mid f_1(\emptyset) \preceq v_1 \preceq f_1(g) \wedge \ldots \wedge f_k(\emptyset) \preceq v_k \preceq f_k(g)\} \\ &= \bigcup_{s \in S} \{v_1 \mid f_1(s) \preceq v_1 \preceq f_1(U)\} \times \ldots \times \{v_k \mid f_k(s) \preceq v_k \preceq f_k(U)\} \cap \\ &\quad \bigcup_{g \in G} \{v_1 \mid f_1(\emptyset) \preceq v_1 \preceq f_1(g)\} \times \ldots \times \{v_k \mid f_k(\emptyset) \preceq v_k \preceq f_k(g)\}. \end{aligned} \tag{6.13}$$

Let $A_i(s)$ be the regular set $\{v_i \mid f_i(s) \preceq v_i \preceq f_i(U)\}$, and let $B_i(g)$ be the regular set $\{v_i \mid f_i(\emptyset) \preceq v_i \preceq f_i(g)\}$. The convex set described in Equation 6.13 can be expressed as the regular set shown in Equation 6.14. Regular grammars are closed under concatenation, intersection, and finite union, so the resulting set is a regular grammar.

$$\langle S, G \rangle = \bigcup_{s \in S} A_1(s) A_2(s) \ldots A_k(s) \cap \bigcup_{g \in G} B_1(g) B_2(g) \ldots B_k(g) \tag{6.14}$$

The number of states in the regular set described in Equation 6.14 depends on the number of states in the $A_i(s)$ and $B_i(g)$ grammars for each s and g. The sizes of these grammars depend on the depth of the generalization hierarchies (trees) for each feature, and whether these hierarchies are tree structured or lattice structured.

6.3.5.2 Tree Structured Hierarchies

If the generalization hierarchy for each feature is tree structured, then there is at most one hypothesis in the S set [Bundy *et al.*, 1985]. Equation 6.14 reduces to Equation 6.15.

$$\langle\{s\}, G\rangle = \bigcup_{g \in G} (A_1(s) \cap B_1(g))(A_2(s) \cap B_2(g)) \ldots (A_k(s) \cap B_k(g)) \quad (6.15)$$

For brevity, the set $A_i(s) \cap B_i(g)$ will also be referred to as $X_i(s, g)$. The set $X_i(s, g)$ is equivalent to $\{v_i \in F_i \mid f_i(s) \preceq v_i \preceq f_i(g)\}$, where F_i is the set of values for feature f_i. It contains all hypotheses between $f_i(s)$ and $f_i(g)$ in the generalization hierarchy for f_i. In a tree structured generalization hierarchy with depth d, $X_i(s, g)$ contains at most d hypotheses. This is because s and g must be on the same branch of the tree in order for s to be less general than g, and no branch of the tree has more than d nodes. The DFA for $X_i(s, g)$ therefore has at most $O(d)$ states.

The DFA for $X_1(s, g)X_2(s, g) \ldots X_k(s, g)$ has at most $O(kd)$ states. Call this DFA $X(s, g)$ for short. The DFA for $\bigcup_{g \in G} X(s, g)$ has at most $O(|G|kd)$ states.

For hypothesis spaces described by conjunctive languages in which the generalization hierarchies for each feature are tree structured and have a maximum depth of d, every convex subset of the space can be expressed as a regular set as shown in Equation 6.14. The DFA for this set has at most $O(|G|kd)$ states.

This relation between the sizes of equivalent convex and regular sets allows the time complexity of intersecting two regular sets and enumerating a hypothesis from their intersection (Equation 6.8) to be rewritten in terms of the boundary set sizes of two equivalent convex sets, as shown in Equation 6.16. In this equation, it is assumed that the regular set $\langle Q_1, s_1, \delta_1, F_1, \Sigma\rangle$ represents the same set of hypotheses as the convex set $\langle S_1, G_1\rangle$, and that $\langle Q_1, s_1, \delta_1, F_1, \Sigma\rangle$ represents the same set of hypotheses as $\langle S_2, G_2\rangle$. Equation 6.16 is derived by substituting $|G_1|kd$ for Q_1 and $|G_2|kd$ for Q_2 in Equation 6.8.

$$O(|Q_1||Q_2|[cost(\delta_1) + cost(\delta_2)]) = O(|G_1||G_2|(kd)^2[cost(\delta_1) + cost(\delta_2)]) \quad (6.16)$$

For most grammars, $cost(\delta_1)$ and $cost(\delta_2)$ are $O(1)$.

The cost of enumerating a hypothesis from the intersection of two regular sets has been expressed in terms of the sizes of S and G. This cost can now be compared to the cost of enumerating a hypothesis from the intersection of two equivalent convex sets, $\langle S_1, G_1 \rangle$ and $\langle S_2, G_2 \rangle$. The cost of intersecting two convex sets is shown in Equation 6.11. For conjunctive languages with tree structured features, the S set never has more than one element, and the cost of determining the generality relation between two hypotheses, $t_<$, is $O(kd)$ [Hirsh, 1990]. The cost of finding the least upper bound of two hypotheses (t_{lub}) is also $O(kd)$ when the generalization hierarchies for each feature are tree structured. The cost of finding the greatest lower bound of two hypotheses (t_{glb}), is also $O(kd)$. Substituting these values into Equation 6.11 yields Equation 6.17.

$$kd + |G_1||G_2|kd + (|G_1| + |G_2| + (|G_1||G_2|)^2 + 2|G_1||G_2|)kd. \tag{6.17}$$

This cost is bounded by Equation 6.18:

$$O((|G_1||G_2|)^2 kd) \tag{6.18}$$

The cost of enumerating a hypothesis from the intersection of two regular sets is $O(|G_1||G_2|(kd)^2)$, and the cost of enumerating a hypothesis from two equivalent convex sets $O((|G_1||G_2|)^2 kd)$. If $kd \ll |G_1||G_2|$, then the cost for regular sets is significantly less than the cost for convex sets. Since the size of the G sets can grow exponentially in the number of examples, this is often a valid assumption. When a convex set is represented as an equivalent regular set, and the generalization hierarchies are tree structured, the regular set is effectively a factored representation of the convex set. This allows the set to be represented more compactly, and improves the computational complexity. Similar complexity improvements can be obtained for convex sets by maintaining them in factored form [Subramanian and Feigenbaum, 1986]. However, there are still expressive differences between convex and regular sets. In particular, regular sets can represent sets with "holes", whereas convex sets cannot. Also, convex sets are not closed under union [Hirsh, 1990], but regular sets are [Hopcroft and Ullman, 1979].

The computational complexity of enumerating a hypothesis from an intersection of n convex sets in IVSM is $n(|G^*|^4)kd$, where $|G^*|$ is the maximum size attained by the G set. The size of G^* can be as high as 2^n [Haussler, 1988]. The computational complexity of enumerating a hypothesis in RS-KII from the same n sets represented as regular sets is is bounded by $O(|G|^n(kd)^n)$, where $|G|$ is the size of the largest G set among the convex sets being intersected. The worst case complexities of IVSM and RS-KII are equivalent. Although RS-KII can be exponential, this is a very loose upper bound. As will be seen in Section 6.4, the complexity of RS-KII can be polynomial when the complexity of IVSM is exponential.

6.3.5.3 Lattice Structured Features

When the generalization hierarchies for the feature are lattice structured rather than tree structured, the relation between the sizes of convex and regular sets changes somewhat.

The S set is not guaranteed to be singleton in this language [Bundy *et al.*, 1985]. Convex subsets are therefore expressed as shown in Equation 6.19.

$$\begin{aligned}\langle S, G\rangle \;=\; & \bigcup_{s\in S}\{v_1 \mid f_1(s) \preceq v_1 \preceq f_1(U)\}\times\ldots\times\{v_k \mid f_k(s) \preceq v_k \preceq f_k(U)\} \cap \\ & \bigcup_{g\in G}\{v_1 \mid f_1(\emptyset) \preceq v_1 \preceq f_1(g)\}\times\ldots\times\{v_k \mid f_k(\emptyset) \preceq v_k \preceq f_k(g)\} \qquad (6.19)\end{aligned}$$

Let $A_i(s)$ be the regular set $\{v_i \mid f_i(s) \preceq v_i \preceq f_i(U)\}$, and let $B_i(g)$ be the regular set $\{v_i \mid f_i(\emptyset) \preceq v_i \preceq f_i(g)\}$. The regular set equivalent to the one in Equation 6.19 is shown in Equation 6.21, where $X_i(s,g) = A_i(s) \cap B_i(g)$.

$$\langle S, G\rangle \;=\; \bigcup_{s\in S} A_1(s)A_2(s)\ldots A_k(s) \cap \bigcup_{g\in G} B_1(g)B_2(g)\ldots B_k(g) \qquad (6.20)$$

$$\;=\; \bigcup_{\langle s,g\rangle\in S\times G} X_1(s,g)X_2(s,g)\ldots X_k(s,g) \qquad (6.21)$$

$X_i(s,g)$ is the set $\{v_i \in F_i \mid f_i(s) \preceq v_i \preceq f_i(g)\}$, where F_i is the set of values in the generalization hierarchy for feature f_i, and $x \preceq y$ means that either value x is less general than value y in the generalization hierarchy for F_i, or that $x = y$. When the hierarchy for F_i is lattice structured, and has a maximum depth of d, there can

be $O(w^d)$ hypotheses between s and g, where w is the width (branching factor) of the lattice. The DFA for $X_i(s,g)$ therefore has $O(w^d)$ states.

Let $X(s,g)$ stand for $X_1(s,g)X_2(s,g)\ldots X_k(s,g)$. The DFA for $X(s,g)$ is a concatenation of the DFAs for each of the features, $X_i(s,g)$, and has at most $O(kw^d)$ states. The DFA for $\bigcup_{\langle s,g\rangle \in S\times G} X(s,g)$ has at most $O(|S||G|kw^d)$ states, by arguments similar to the ones used for tree structured features.

This relation between the number of states in a regular set and the boundary sets sizes of an equivalent convex set can be used to compare the complexities of intersecting and enumerating hypotheses from both regular and convex sets. Let $\langle Q_1, s_1, \delta_1, F_1, \Sigma_1\rangle$ be a regular set that represents the same set of hypotheses as the convex set $\langle S_1, G_1\rangle$, and let $\langle Q_1, s_1, \delta_1, F_1, \Sigma_1\rangle$ be a regular set that represents the same set of hypotheses as the convex set $\langle S_2, G_2\rangle$. The cost of intersecting the two regular sets and enumerating a hypothesis from the intersection is bounded by $O(|Q_1||Q_2|(cost(\delta_1) + cost(\delta_2)$ (Equation 6.8), where $cost(\delta_1)$ and $cost(\delta_2)$ are the costs of computing the next-state functions, δ_1 and δ_2. These costs can generally be assumed to be constant unless they are proportional to a relevant scale-up variable. Substituting $O(|S_1||G_1|kw^d)$ for the size of Q_1, and $O(|S_2||G_2|kw^d)$ for the size of Q_2, yields the cost for intersecting and enumerating a hypothesis from the two regular grammars in terms of the boundary set sizes of the equivalent convex sets. This cost is shown in Equation 6.22.

$$O(|S_1||S_2||G_1||G_2|(kw^d)^2) \tag{6.22}$$

This cost of intersecting two regular sets and enumerating a hypothesis from their intersection can now be compared to the cost of performing the same operation on two convex sets. Let $\langle S_1, G_1\rangle$ and $\langle S_2, G_2\rangle$ be two convex sets that represent, respectively, the same sets of hypotheses as the two regular sets $\langle Q_1, s_1, \delta_1, F_1, \Sigma_1\rangle$ and $\langle Q_2, s_2, \delta_2, F_2, \Sigma_2\rangle$. Since the cost of enumerating a hypothesis from a convex set is $O(1)$ (see Section 6.3.4), the cost of intersection and enumeration is just the cost of intersecting two convex sets, as given in Equation 6.11. This equation depends on the boundary set sizes of both convex sets, the cost of determining the generality relation between two hypotheses ($t_<$), and the costs of finding the least upper bound (t_{lub}) and greatest lower bound (t_{glb}) of two hypotheses in the generalization hierarchy.

For lattice structured feature hierarchies, the cost of determining whether one hypothesis is more general than another is $O(kw^d)$ [Hirsh, 1990]. The costs of finding the least upper bound or greatest lower bound of two hypotheses in the generalization hierarchy are also bounded by $O(kw^d)$. The intuition behind this bound is that all three of these operations are essentially searches in the generalization hierarchy, starting from the hypotheses in question. The hierarchy for each feature can be searched independently. There are k features, each with a generalization hierarchy of width w and depth d. This leads to k independent searches, each visiting at most w^d nodes in the hierarchy. Substituting $O(kw^d)$ for t_{lub}, t_{glb} and $t_<$ in Equation 6.11 yields the cost bound shown in Equation 6.23.

$$\begin{aligned} &|S_1||S_2|kw^d + |G_1||G_2|kw^d + \\ &\quad [(|S_1||S_2|)^2 + |S_1||S_2|(|G_1| + |G_2|) + \\ &\quad (|G_1||G_2|)^2 + |G_1||G_2|(|S_1| + |S_2|)]kw^d \end{aligned} \tag{6.23}$$

This is in turn bounded by Equation 6.24.

$$O([(|S_1||S_2|)^2 + |S_1||S_2|(|G_1| + |G_2|) + (|G_1||G_2|)^2 + |G_1||G_2|(|S_1| + |S_2|)]kw^d) \tag{6.24}$$

When the feature generalization hierarchies are lattice structured, the cost of intersecting two convex sets and enumerating a hypothesis from their intersection (Equation 6.22) is roughly equivalent to the cost for performing the same operation on convex sets (Equation 6.24), depending on the relative sizes of the boundary sets and the feature hierarchies.

If the boundary sets are all about the same size, x, then the cost for regular sets becomes $O(x^4(kw^d)^2)$ and the cost for convex sets becomes $O(x^4kw^d)$. The convex sets are a little more efficient, by a factor of kw^d, the cost of comparing two hypotheses. The regular sets are less efficient, since the comparison is encoded in the states of the DFA, and these extra states combine multiplicatively when the DFAs are intersected. If the size of the boundary sets is significantly larger than kw^d, then this additional factor does not have much of an impact, and the complexities of regular and convex sets are about the same with respect to the combined intersection and enumeration operation.

When the boundary set sizes differ, so that the S sets are much smaller or larger than the G sets, regular sets are more efficient. Let the sizes of S_1 and S_2 each be s, and the sizes of G_1 and G_2 each be g. The complexity of the intersection and enumeration operation for regular sets is $O(s^2g^2(kw^d)^2)$, and the complexity of the same operation on convex sets is $O([s^4+g^4+s^2g+g^2s]kw^d)$. This difference is most likely attributable to the fact that convex set intersection involves a minimization step that requires $O(s^4+g^4)$ comparisons among the hypotheses of the boundary sets, whereas regular set intersection has no corresponding minimization step. The complexity for convex sets is a squared factor greater than the complexity for regular sets if $(s+g)^2 \ll kw^d$. If kw^d dominates $(s+g)^2$, then convex sets are more efficient than regular sets.

6.4 Exponential Behavior in CS-KII and RS-KII

For both RS-KII and CS-KII, the worst case complexity for inducing a hypothesis from n examples is $O(2^n)$. This exponential behavior occurs in CS-KII and RS-KII for different reasons. The computational cost of inducing a hypothesis in RS-KII is proportional to the size of the DFA for the solution set. When P is empty, the solution set is just C. The C set is the intersection of n DFAs, one for each knowledge fragment. The intersection of two DFAs, A and B, results in a new DFA with $|Q_A||Q_B|$ states, where $|Q_A|$ is number of states in A, and $|Q_B|$ is the number of states for B. Intersecting n DFA's, each of size r, can result in a final DFA with up to r^n states.

For CS-KII, the cost of integrating an example is proportional to the size of the boundary sets for the current version space. The worst case complexity occurs when integrating an example causes the boundary sets to grow geometrically. When a new negative example is processed, there may be more than one way to specialize each hypothesis in the G set in order to exclude the example. If each hypothesis has two specializations, and none of these hypotheses need to be pruned from G, then the size of the G set doubles. Recall that a hypothesis is pruned from G if it is more specific than some other element of G, or if it is not more general than any element of S. The S set grows in the same fashion when there is more than one way to generalize the hypotheses in S in order to cover a new positive example. This kind of growth can

also occur when non-example knowledge fragments are integrated. This geometric increase in the size of a boundary set is known as *fragmentation.* After processing n fragmenting examples or knowledge fragments, the affected boundary set contains 2^n hypotheses. In a geometric series, the cost of processing the last example dominates the cost of processing all of the previous examples, so the computational complexity is $O(2^n)$.

Although both RS-KII and CS-KII can be exponential in the worst case, the complexity of RS-KII is bounded by that of CS-KII for some hypothesis space languages, such as the conjunctive languages described in Section 6.2.2.1. In one case, the complexity of RS-KII is considerably less than that of CS-KII. This is the hypothesis space and set of examples described by Haussler [Haussler, 1988], which induces exponential behavior in CEA (and in IVSM and CS-KII by extension). For this same task, the complexity of RS-KII is only polynomial in the number of examples.

6.4.1 Haussler's Task

Haussler [Haussler, 1988] describes a hypothesis space and sequence of examples that cause the boundary sets of CEA to grow geometrically. Both the space complexity and time complexity of CEA are exponential in the number of examples.

Hypotheses in Haussler's task are conjuncts of feature values, with one value for each of k features. The possible values for each feature are `true`, `false`, and `don't-care`. These values are abbreviated as `t`, `f`, and `d`, respectively. For example, a hypothesis for $k = 4$ might be ⟨`t,f,d,t`⟩. The `don't-care` value is more general than the `true` and `false` values, and neither `true` nor `false` is more general than the other.

The examples for this task consist of $k/2$ negative examples and a single positive example. The positive example is ⟨`t,t,t, ... t`⟩. The negative examples have two adjacent features with the value `false`, and the remaining features have the value `true`. For the i^{th} negative example, features $2i$ and $2i+1$ are `false`. See Figure 6.8. The positive example is presented first, followed by the $k/2$ negative examples in order.

$$
\begin{array}{l}
\langle\ \mathtt{f},\ \mathtt{f},\ \mathtt{t},\ \mathtt{t},\ \ldots,\ \mathtt{t},\ \mathtt{t}\ \rangle \\
\langle\ \mathtt{t},\ \mathtt{t},\ \mathtt{f},\ \mathtt{f},\ \ldots,\ \mathtt{t},\ \mathtt{t}\ \rangle \\
\qquad\qquad\qquad\quad \ddots \\
\langle\ \mathtt{t},\ \mathtt{t},\ \mathtt{t},\ \mathtt{t},\ \ldots,\ \mathtt{f},\ \mathtt{f}\ \rangle
\end{array}
$$

Figure 6.8: Haussler's Negative Examples.

6.4.2 Performance of CS-KII and RS-KII

For each negative example in Haussler's task there are two ways to specialize each element of the G set in order to exclude the negative example, one for for each of the example's `false` values. None of the specializations can be pruned from G, so the size of the G set doubles after processing each negative example. After processing all $k/2$ examples, the G set contains $2^{k/2}$ hypotheses. The cost of induction in CS-KII is proportional to the sizes of the S and G sets, so the cost of inducing a hypothesis from $n = k/2$ of these examples is $O(2^n)$.

The cost of inducing a hypotheses from these same examples in RS-KII is only $O(n^2)$. This dramatic change is due to the representational differences between convex sets and DFAs. Processing one of Haussler's negative examples doubles the number of most general hypotheses in the version space. This doubles the size of the convex set representation of the version space, but does not double the size of the equivalent DFA representation, at least not in this hypothesis space. The size of the DFA may grow geometrically in other hypothesis spaces, or for other (non-example) knowledge sources. In this hypothesis space, processing one of Haussler's negative examples only adds a single state to the DFA. Thus, the size of the DFA grows only linearly in the number of examples. The initial DFA has k states, so the cost of processing $k/2$ negative examples is $\sum_{i=1}^{k/2}(k+i) = O(k^2)$. Substituting n for $k/2$ yields $O(n^2)$.

An example of RS-KII solving Haussler's task for six features ($k = 6$) is given below, followed by a more detailed complexity analysis motivated by this example. The detailed analysis includes some additional costs that do not appear in the simplified analysis above, but the overall complexity is still $O(n^2)$.

6.4.2.1 RS-KII's Performance on Haussler's Task

For this instantiation of Haussler's task there are six features. The examples therefore consist of one positive example, and three ($k/2$) negative examples:

- $p_0 = \langle \texttt{t,t,t,t,t,t} \rangle$
- $n_1 = \langle \texttt{f,f,t,t,t,t} \rangle$
- $n_2 = \langle \texttt{t,t,f,f,t,t} \rangle$
- $n_3 = \langle \texttt{t,t,t,t,f,f} \rangle$

Each example is translated into a $\langle C, P \rangle$ pair, where C is the set of hypotheses consistent with the example, and P is the empty set. C corresponds to the version space of hypotheses consistent with the example. Translations of the four examples are shown in Table 6.1. The corresponding DFAs for the C sets are shown in Figure 6.9 for the positive example, and in Figure 6.10, Figure 6.11, and Figure 6.12 for the three negative examples. The unshaded nodes are dead states.

Example	C	P
$p_0 = \langle \texttt{t,t,t,t,t,t} \rangle$	$C_0 = \texttt{(d\|t)(d\|t)(d\|t)(d\|t)(d\|t)(d\|t)}$	$P_0 = \{\}$
$n_1 = \langle \texttt{f,f,t,t,t,t} \rangle$	$C_1 = \overline{\texttt{(d\|f)(d\|f)(d\|t)(d\|t)(d\|t)(d\|t)}}$	$P_1 = \{\}$
$n_2 = \langle \texttt{t,t,f,f,t,t} \rangle$	$C_2 = \overline{\texttt{(d\|t)(d\|t)(d\|f)(d\|f)(d\|t)(d\|t)}}$	$P_2 = \{\}$
$n_3 = \langle \texttt{f,f,t,t,f,f} \rangle$	$C_3 = \overline{\texttt{(d\|t)(d\|t)(d\|t)(d\|t)(d\|f)(d\|f)}}$	$P_3 = \{\}$

Table 6.1: Translations of Haussler's Examples.

The four examples are integrated into RS-KII one at a time. After integrating each example, the solution set of the resulting $\langle C, P \rangle$ pair is tested for emptiness. If the solution set to $\langle C, P \rangle$ is empty, then there are no hypotheses consistent with the examples seen so far, much less with all of the examples, so no more examples are processed. These actions emulates the behavior of CEA, which integrates each example one at a time, and after integrating each example tests the resulting version space for emptiness. If it is empty, then the version space has *collapsed*, and the remaining examples are not integrated.

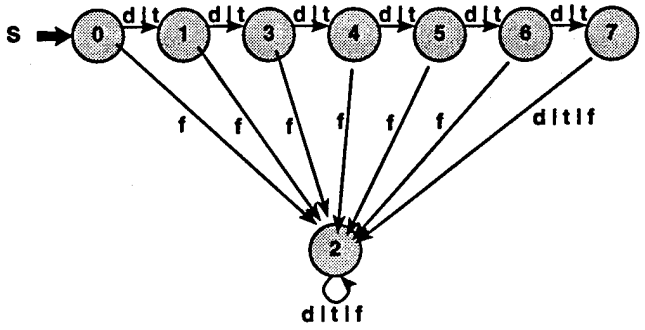

Figure 6.9: DFA for C_0, the Version Space Consistent with p_0.

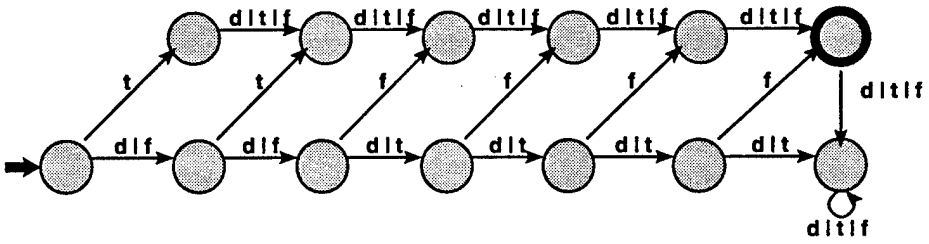

Figure 6.10: DFA for C_1, the Version Space Consistent with n_1.

The test for emptiness is important for the polynomial behavior of RS-KII. The empty test identifies and eliminates dead states from the DFA for C in the process of ascertaining whether C is empty. If these dead states are not removed, then when the C set for the next example is intersected, those dead states combine multiplicatively with the states in C, so that the number of dead states in the intersection is proportional to the product of the states in C and the number of original dead states. This geometric growth means that the DFA for $C_0 \cap C_1 \cap \ldots C_{k/2}$ has $O(2^{k/2})$ states,

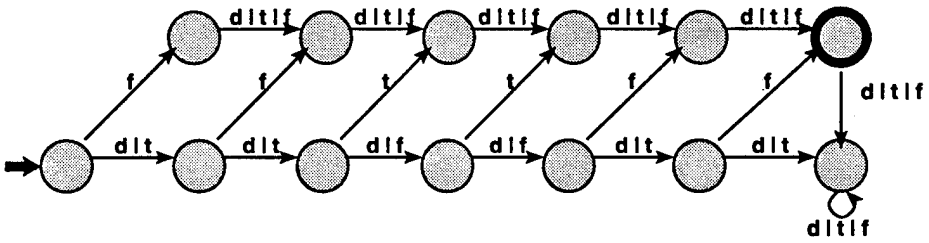

Figure 6.11: DFA for C_2, the Version Space Consistent with n_2.

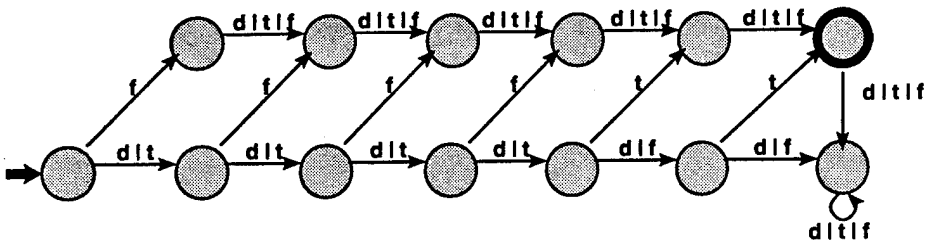

Figure 6.12: DFA for C_3, the Version Space Consistent with n_3.

most of which are dead states. Intersecting these DFAs takes time proportional to $O(k/2)$, since intersection is a constant time operation in RS-KII, but enumerating a hypothesis from the intersection can take time proportional to the number of states in the DFA, or $O(2^{k/2})$. The complexity of CEA on this task is also $O(2^{k/2})$.

However, if the dead states are eliminated after each intersection, then the DFA for $C_0 \cap C_1 \cap \ldots \cap C_i$, has only a few more states than the DFA for $C_0 \cap C_1 \cap \ldots \cap C_{i-1}$. This linear growth leads to a polynomial time complexity for RS-KII on this task, as will be seen below. The final DFA for the intersection of the C sets for all $k/2$ examples has only $O(k)$ states, and the time spent in eliminating dead states is only $O(k^2)$.

The positive example, p_0, is seen first. It is translated into $\langle C_0, P_0 \rangle$ and integrated into RS-KII. This is the first knowledge seen, so $\langle C, P \rangle = \langle C_0, P_0 \rangle$. The negative examples are then processed one at a time, starting with n_1. Example n_1 is translated into $\langle C_1, P_1 \rangle$ and integrated with $\langle C, P \rangle$, yielding $\langle C_0 \cap C_1, P_0 \cup P_1 \rangle$.

Constructing the implicit DFAs for $C_0 \cap C_1$ and $P_0 \cup P_1$ are constant time operations. The solution set for $\langle C_0 \cap C_1, P_0 \cup P_1 \rangle$ is then tested for emptiness. The P sets are empty for all of the examples in this task, so $P_0 \cup P_1 = \emptyset$, and the solution set is just $C_0 \cap C_1$. Testing whether $C_0 \cap C_1$ is empty takes time proportional to the number of states in the DFA representing the intersection. The DFA for $C_0 \cap C_1$ is shown in Figure 6.13. The unshaded states are dead states.

The empty test tries to find a path from the start state of the DFA to an accept state. Along the way, it eliminates any dead states it finds. If all of the states are identified as dead states and eliminated, then the DFA is empty. Removing the dead states from the DFA for $C_0 \cap C_1$ results in the DFA shown in Figure 6.14.

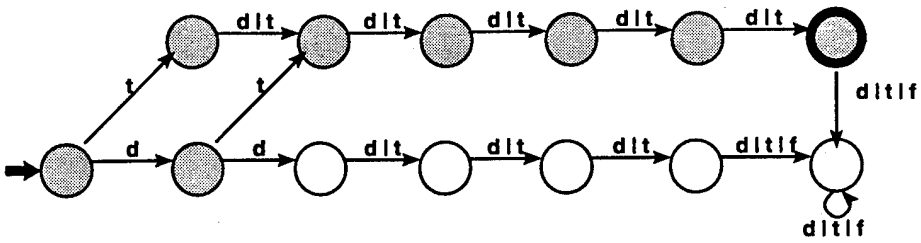

Figure 6.13: DFA for $C_0 \cap C_1$.

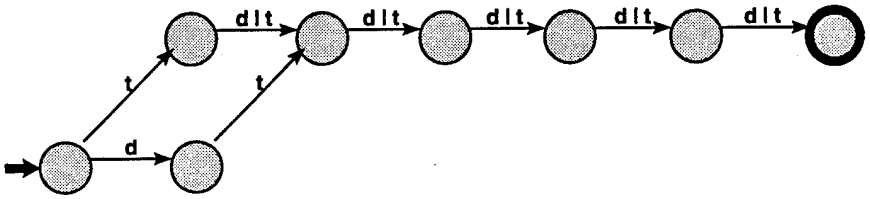

Figure 6.14: DFA for $C_0 \cap C_1$ After Empty Test.

The same process is repeated on the remaining negative examples. Example n_2 is seen next, and translated into $\langle C_2, P_2 \rangle$. Integrating $\langle C_2, P_2 \rangle$ with $\langle C_0 \cap C_1, \emptyset \rangle$ yields $\langle C_0 \cap C_1 \cap C_2, \emptyset \rangle$. The DFA for $C_0 \cap C_1 \cap C_2$ is shown in Figure 6.15. This DFA is tested for emptiness, and its dead states are removed as a side effect. This results in the DFA of Figure 6.16.

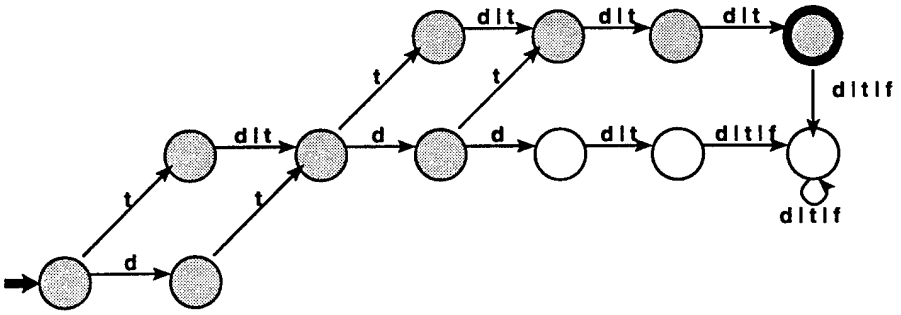

Figure 6.15: DFA for $C_0 \cap C_1 \cap C_2$.

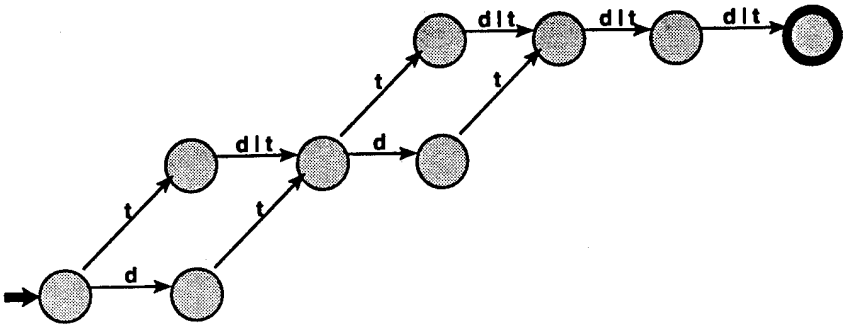

Figure 6.16: DFA for $C_0 \cap C_1 \cap C_2$ After Empty Test.

Finally, n_3 is seen. Translating and integrating n_3 yields $\langle C = C_0 \cap C_1 \cap C_2 \cap C_3, \emptyset \rangle$. The DFA for C is shown in Figure 6.17. After the empty test removes the dead states, the DFA for C is as shown in Figure 6.18.

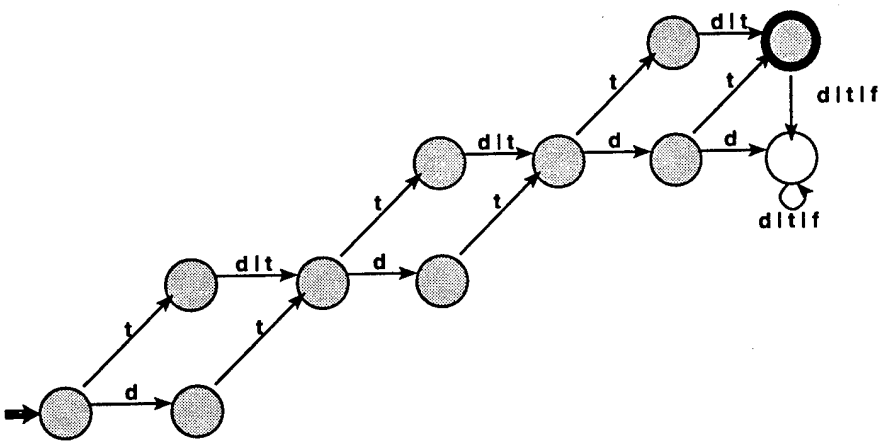

Figure 6.17: DFA for $C_0 \cap C_1 \cap C_2 \cap C_3$.

$C_0 \cap C_1 \cap C_2 \cap C_3$ corresponds to the version space consistent with all four examples. This is the same version space computed by IVSM from these examples, though

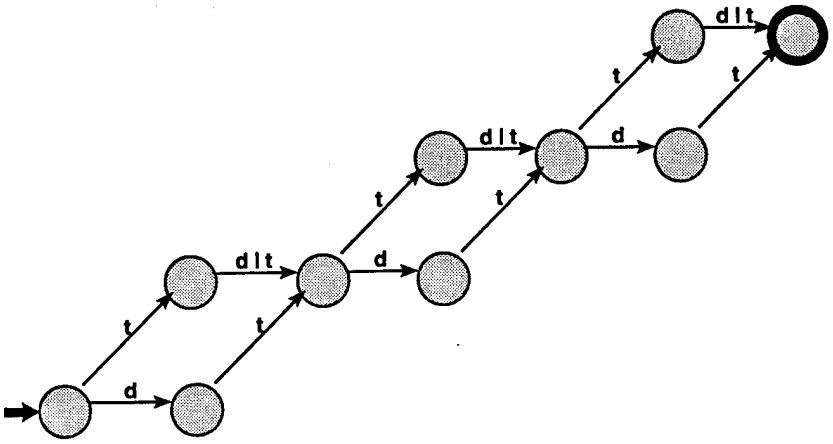

Figure 6.18: DFA for $C_0 \cap C_1 \cap C_2 \cap C_3$ After Empty Test.

represented as a DFA instead of as a convex set. The equivalent convex set is shown below.

$$
\begin{aligned}
S &= \{\langle \mathtt{t},\mathtt{t},\mathtt{t},\mathtt{t},\mathtt{t},\mathtt{t}\rangle\} \\
G &= \{\langle \mathtt{t},\mathtt{d},\mathtt{t},\mathtt{d},\mathtt{t},\mathtt{d}\rangle, \langle \mathtt{t},\mathtt{d},\mathtt{t},\mathtt{d},\mathtt{d},\mathtt{t}\rangle, \langle \mathtt{t},\mathtt{d},\mathtt{d},\mathtt{t},\mathtt{t},\mathtt{d}\rangle, \langle \mathtt{t},\mathtt{d},\mathtt{d},\mathtt{t},\mathtt{d},\mathtt{t}\rangle, \\
&\quad \langle \mathtt{d},\mathtt{t},\mathtt{t},\mathtt{d},\mathtt{t},\mathtt{d}\rangle, \langle \mathtt{d},\mathtt{t},\mathtt{t},\mathtt{d},\mathtt{d},\mathtt{t}\rangle, \langle \mathtt{d},\mathtt{t},\mathtt{d},\mathtt{t},\mathtt{t},\mathtt{d}\rangle, \langle \mathtt{d},\mathtt{t},\mathtt{d},\mathtt{t},\mathtt{d},\mathtt{t}\rangle\}
\end{aligned}
$$

The S set contains one hypotheses, and the G set contains eight hypotheses, each with six features. This consumes 54 units of space. The equivalent regular set has eleven states, with three edges per state, for a total of 33 units of space. Edges that lead directly to a dead state have been omitted from the figures for clarity.

6.4.3 Complexity Analysis

The costs involved in processing the examples are the cost of translating each example, the cost of integrating each example, and the cost of applying the empty test to the solution set of the resulting C set.

The cost of translating example i into $\langle C_i, P_i \rangle$ is the cost of constructing the DFAs for C_i and P_i. The cost of constructing one of these DFA is proportional to the number of states and edges it has. P_i is always empty, so the cost of constructing

P_i is $O(1)$. The DFA for the C set of the positive example has $k+2$ states and $(k+2)|\Sigma|$ edges. The DFA for the C set of each negative example has $(2k+1)$ states and $(2k+1)|\Sigma|$ edges. The cost of translating the examples is the sum of the sizes for each DFA:

$$[(k+2)+(k+2)|\Sigma|]+(k/2)[(2k+1)+(2k+1)|\Sigma|] = (k^2+3k/2+2)(|\Sigma|+1) \quad (6.25)$$

Each example is processed by integrating its translation, $\langle C_i, P_i\rangle$, into the $\langle C, P\rangle$ pair that represents all of the examples integrated so far, and then testing whether the solution set of $\langle C \cap C_i, P \cup P_i\rangle$ is empty. Integration in RS-KII is a constant time operation, so the cost of integrating all $1+k/2$ examples is $O(k)$.

The cost of $Empty(\langle C \cap C_i, P \cup P_i\rangle$ is proportional to the number of states in the DFA for the solution set of $\langle C \cap C_i, P \cup P_i\rangle$. Since P and P_i are always empty in this task, the solution set is just $C \cap C_i$. The cost of the empty test is therefore proportional to the number of states in the DFA for $C \cap C_i$.

The positive example is the first one integrated, so the empty test is applied to $\langle C_0, P_0\rangle$, the translation of p_0. Since P_0 is empty, this is equivalent to testing whether C_0 is empty. The DFA for C_0 is minimal, as are the DFAs for all of the examples. Determining the emptiness of a minimal DFA is a constant time operation, so the emptiness of C_0 can be determined in constant time.

For a negative example, n_i, the empty test is applied to $C \cap C_i$, where $\langle C_i, \emptyset\rangle$ is the translation for n_i, and C is the set of hypotheses consistent with all of the previously observed examples (p_0 through n_{i-1}). Determining the emptiness of $C \cap C_i$ is proportional to the number of states in the DFA for $C \cap C_i$. This number can be expressed as a function of i and the number of features, k. After processing examples p_0 through n_{i-1}, the DFA for C is of the form shown in Figure 6.19.

The DFA for C consists of $i-1$ "diamonds", each with three states, and a chain of $k-2(i-1)$ states. Intersecting the DFA for C with the DFA for C_i, results in a DFA of the form shown in Figure 6.20. This DFA is identical to the DFA for C up to state $i-1$. After that, it has one more "diamond", two chains of length $k-2i$, and a dead state. There are i diamonds, each with three states for a total of $3i$ states.

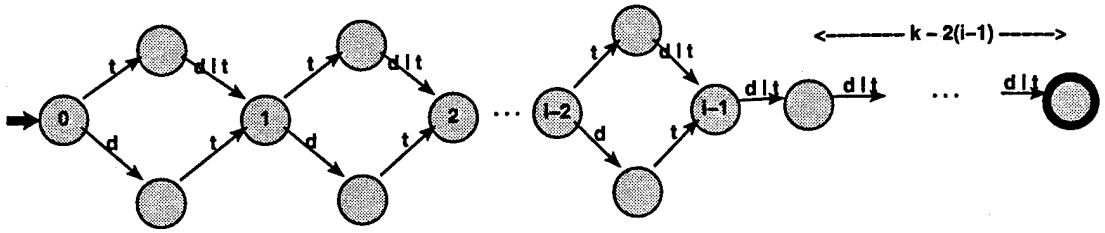

Figure 6.19: DFA for $C_0 \cap C_1 \cap \ldots \cap C_{i-1}$ After Empty Test.

This does not include the state labeled i. There are two branches of $k - 2i$ states, and a dead state. The total number of states in $C \cap C_i$ is therefore:

$$3i + 1 + 2(k - 2i) + 1 = 2k - i + 2 \tag{6.26}$$

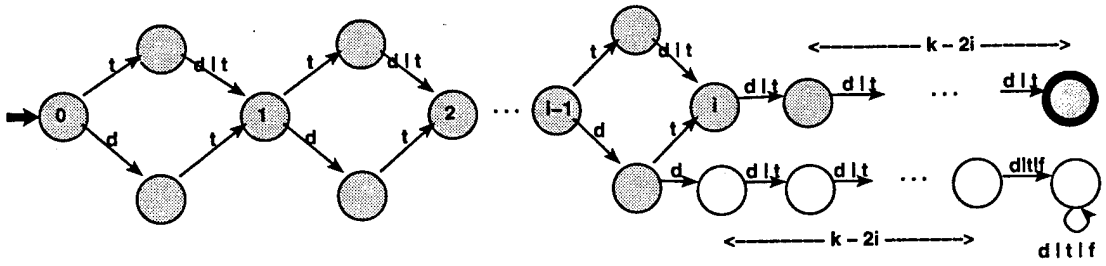

Figure 6.20: DFA for $C \cap C_i$.

The lower branch in Figure 6.20 (unfilled circles) consists entirely of dead states, and is pruned by the empty test. The dead branch has $(k - 2i + 1)$ states, so after applying the empty test, the number of states in $C \cap C_i$ is given by the following equation:

$$(2k - i + 2) - (k - 2i + 1) = k + i + 1 \tag{6.27}$$

The cost of applying the empty test to $C \cap C_i$ for all $k/2$ negative examples is given by Equation 6.28.

$$\begin{aligned}\sum_{i=1}^{k/2} 2k - i + 2 &= k^2 + k - \sum_{i=1}^{k/2} i \\ &= k^2 + k - (k/2)(k/2+1)/2 \\ &= k^2 + k - (k^2 + 2k)/16 \\ &= \frac{15}{16}k^2 + \frac{7}{8}k \end{aligned} \tag{6.28}$$

The cost for the entire induction task is the translation cost (Equation 6.25) plus the integration cost ($O(k)$) plus the cost of the empty test (Equation 6.28). The sum of these costs is given by the following equation:

$$\begin{aligned}&(k^2 + 3k/2 + 2)(|\Sigma| + 1) + k + \frac{15}{16}k^2 + \frac{7}{8}k \\ &= (1\frac{15}{16} + |\Sigma|)k^2 + (2\frac{3}{8} + \frac{3}{2}|\Sigma|)k + 2|\Sigma| + 2 \\ &= O(k^2|\Sigma|)\end{aligned}$$

In this task, Σ is {t,f,d} regardless of the number of features, so $|\Sigma|$ can be treated as a constant. Substituting n for $k/2$, we get a total cost of $O(n^2)$.

6.4.4 Summary

The improvements of RS-KII over CS-KII on this task are due to representational differences between convex and regular sets. A convex set stores the entire G set and S set. In Haussler's task, the size of the G set doubles with each negative example, so that it contains $O(2^n)$ elements after processing n examples. Representing the same version space as a DFA requires only a constant number of states. The DFA effectively stores the version space in factored form. Factored convex-set representations have been shown to have similar complexity improvements [Subramanian and Feigenbaum, 1986].

The regular set representation has similar complexity results for a range of hypothesis spaces, including the conjunctive languages with tree-structured features, and to a lesser degree for conjunctive languages with lattice-structured features. For

the latter hypothesis space, the cross-feature fragmentation is controlled for by the DFA representation, but there is also within-feature fragmentation that increases the complexity. However, the complexity is still less than it would be for a completely unfactored representation.

6.5 Discussion

Regular and convex sets overlap in the knowledge they can represent, but neither representation strictly subsumes the other. There are many knowledge fragments that can be expressed as either a regular set or a convex set, but there are also fragments that can be expressed in only one of the representations and not the other.

The relative computational complexities of RS-KII and CS-KII depend on the knowledge being utilized, but for conjunctive languages with tree and lattice structured features, both RS-KII and CS-KII have the same worst-case complexity. RS-KII effectively represents the version space in factored form [Subramanian and Feigenbaum, 1986], which makes the cost of enumerating a hypothesis from the intersection of regular sets cheaper than the cost of intersecting equivalent convex sets by about a square factor. The factored representation also allows RS-KII to learn from Haussler's examples [Haussler, 1988] in polynomial time, whereas IVSM and CEA both take exponential time.

Chapter 7

Related Work

7.1 IVSM

Incremental Version Space Merging (IVSM) [Hirsh, 1990] was one of the first knowledge integration systems for induction, and provided much of the motivation for KII. IVSM integrates knowledge by translating each knowledge fragment into a version space of hypotheses consistent with the knowledge, and then intersecting the version spaces for the fragments to get a version space consistent with all of the knowledge. This theoretically allows IVSM to utilize any knowledge that can be expressed as a constraint on the hypothesis space. In practice, IVSM represents version spaces as convex sets, and this limits the constraints that can be expressed, and therefore the knowledge that can be used.

In later work, Hirsh suggests using representations other than convex sets for version spaces, and identifies representations that guarantee polynomial time bounds on the cost of induction [Hirsh, 1992]. KII expands on this work by extending the space of set representations for the version space (i.e., C) from the few suggested by Hirsh to the space of all possible set representations. KII also expands on IVSM by allowing knowledge to be expressed in terms of preferences as well as constraints, thereby increasing the kinds of knowledge that can be utilized. Finally, KII facilitates formalization of the space of set representations by mapping them onto the space of grammars, and using results from automata and formal language theory to establish upper bounds on the expressiveness of set representations. KII strictly subsumes IVSM, in that IVSM can be cast as an instantiation of KII with convex sets (CS-KII).

7.2 Grendel

Grendel [Cohen, 1992] is another cognitive ancestor of KII. The motivation behind Grendel is to better understand the effects of inductive biases on learning by expressing the biases explicitly in the form of a grammar. Grendel expresses the biases as hard constraints on the hypothesis space, and as preferences on the order in which the space is searched. The constraints are expressed as an antecedent description grammar[1], and the preferences are represented by marking productions in the grammar as *preferred* or *deferred*. The grammar representing a given collection of biases is constructed by a *translator* that takes as input all of the biases and outputs the grammar. The language of the grammar is the biased hypothesis space. The grammar is then searched for a hypothesis (string) consistent with the examples. The search is guided by an information gain metric, and the preference markings on the grammar productions.

Grendel can utilize a wide range of knowledge (biases), but it cannot integrate knowledge. The integration work is done by the translator, not by Grendel itself. A translator takes as input *all* of the biases, and outputs a single grammar. It is not possible to translate the biases independently and integrate the grammars, as is done in KII, because the grammar is essentially context free and not closed under intersection. In Grendel, each new combination of biases requires a new translator. This is in contrast to KII, where knowledge fragments can be translated and integrated independently, so that it is possible to have a single translator for each knowledge fragment. This allows knowledge to be added or omitted much more flexibly than in Grendel. This independence also means that the knowledge integration effort in KII occurs primarily within KII's integration operator, as opposed to occurring primarily within a single translator, as is the case with Grendel.

KII's greater flexibility in integrating knowledge comes from two sources. First, constraints can be expressed in languages that are closed under intersection, which allows constraints to be specified independently and composed via set intersection. Second, preferences are represented as a separate grammar instead of as orderings on the productions of the constraint grammar, as is the case in Grendel. This removes

[1] An antecedent description grammar is essentially a context free grammar.

a source of dependence between preferences and the biases encoded in the constraint grammar, and provides a potentially more expressive language for the preferences.

KII also removes the somewhat arbitrary distinction between biases and examples. Grendel treats these as separate entities, but KII treats them both equally. This uniformity facilitates certain analyses—such as determining an upper bound on the expressiveness of set representations for which induction is computable—that would be more difficult if examples were treated differently than other forms of knowledge. Grendel treats examples in a fixed way, assuming that they are noise-free and using an information gain metric to select among the strictly consistent hypotheses. KII allows examples to be translated in more than one way, which permits other assumptions about the examples, such as that they have noise conforming to the bounded inconsistency model.

7.3 Bayesian Paradigms

Buntine [Buntine, 1991] describes a knowledge integration system in which knowledge is represented as prior probability distributions over the hypotheses. The prior probability for a hypothesis is its prior probability of being the target concept. The hypothesis space is then searched for a hypothesis with the highest prior probability.

This is similar to KII, except that knowledge is expressed as probability distributions instead of as constrained optimization problems. Each of these representations make different trade-offs between expressiveness, efficiency, and ease of constructing translators. Whether the constraint paradigm or the Bayesian paradigm is more appropriate depends on the available knowledge.

In the Bayesian paradigm, it may be difficult to find priors that adequately express a piece of knowledge. Solving the probability equations to find the most probable concept may also be difficult, depending on the distribution. KII is most appropriate when the knowledge can be easily expressed as constraints and preferences in set representations for which induction is computable, and preferably polynomial.

7.4 Declarative Specification of Biases

Russell and Grosof [Russel and Grosof, 1987] describe a knowledge integration system for induction in which knowledge, in the form of inductive biases, is translated into *determinations*. The determinations and the examples deductively imply the target concept. It is also possible to search for a desired set of determinations, such as those that do not overconstrain the solution, and deduce the target concept from these determinations. This yields the ability to dynamically shift the bias.

This system shares with KII the idea that induction is the process of identifying a hypothesis that is deductively implied by the knowledge. The inductive leaps come from unsupported assumptions made by the biases (knowledge), and from having to select a hypothesis arbitrarily when more than one hypothesis is deductively implied by the knowledge.

A determination expresses a bias—that is, a preference for selecting certain hypotheses over others as the target concept. This can certainly be expressed in terms of constraints and preferences over the hypothesis space. For example, a determination of the form `nationality(?x, ?n) ::> language(?x,?l)` can be translated along with an example, `nationality(Fritz, German)`, `language(Fritz, German)`, into a preference for hypotheses that imply `language(?x, German)` is true whenever `nationality(?x, German)` is true. The exact form of the preference would depend on the hypothesis space and the preference set representation. Determinations are no more expressive than $\langle H, C, P \rangle$ tuples, but it may be more natural to express some knowledge in terms of determinations than in terms of constraints and preferences, and vice versa. As with the Bayesian paradigm, the appropriateness of a given framework depends on how naturally the knowledge at hand can be expressed in that framework.

KII can have instantiations at various trade-offs points between expressiveness and complexity. This is useful for investigating the effects of knowledge on induction, and for generating induction algorithms that guarantee certain time complexities. The framework of Russell and Grosof can not make such trade-offs directly. It may be possible to find restricted determination languages with desirable complexity properties, but this is not supported in any principled fashion by the framework. It can, however, shift the bias by selecting which knowledge to utilize. KII has

no equivalent capability, although the choice of set representation does determine which biases can be utilized, albeit at a much coarser grain than Russell and Grosof's system.

7.5 PAC learning

The PAC learning literature (e.g., [Vapnik and Chervonenkis, 1971], [Valiant, 1984]) investigates, in part, the conditions under which a concept can be learned in polynomial time from a polynomial number of examples. One of the main results from this literature is that all concepts in a family of concepts are polynomially learnable if any given concept in the family can be identified within a given error margin and confidence level by a polynomial number of examples, and the time needed to identify a hypothesis consistent with the examples is polynomial in the number of examples [Valiant, 1984].

The KII framework is concerned with the hypothesis identification half of this result. KII provides operations for identifying hypotheses consistent with a collection of knowledge fragments, not just examples. The set representation determines the complexity of identification, and determines the knowledge that can be expressed. If the representations for C and P are such that a hypothesis can be enumerated from the solution set in polynomial time, then the cost of identification is polynomially bounded for any knowledge expressible in these representations. This complements the PAC results, which deal with the cost of identifying a hypothesis from examples only.

The accuracy of a hypothesis induced from examples depends on the Vapnik-Chervonenkis dimension (VC-dimension) [Vapnik and Chervonenkis, 1971] of the family of concepts to which the target concept belongs. The VC-dimension determines how many examples are needed in order to guarantee that the hypotheses consistent with the examples will have a given level of accuracy. KII has nothing formal to say as yet about the number of knowledge fragments needed to guarantee a certain level of accuracy. This is an area for future research. However, accuracy is generally correlated with the number of correct knowledge fragments, and this depends in turn on the expressiveness of the set representation. More expressive set

representations can represent more of the available knowledge, and thus the accuracy of the learned hypothesis tends to increase with the expressiveness of the set representation.

Chapter 8

Future Work

8.1 Long Term Vision

The ultimate vision for this work is to provide a general framework for integrating knowledge into induction. I envision a library of translators for different knowledge sources, hypothesis spaces, and set representations, and a library of set representations with which to instantiate KII. This would provide the user maximum flexibility in deciding which hypothesis space to use, which of the available knowledge to use, and in what set representation to express the knowledge.

The ability to choose a set representation is particularly important, since it allows the user to make trade-offs between expressiveness and complexity. If it is important to utilize all of the knowledge, a very expressive representation may be most appropriate. If complexity is more of an issue, an inexpressive representation with low complexity bounds may be more desirable. The representation also helps the user evaluate the utility of available knowledge fragments. If a knowledge fragment requires an expressive representation, but the remaining knowledge can be expressed in a more restricted representation, then the benefits of using the more expressive knowledge may not be worth the added complexity.

Instead of omitting overly expensive knowledge altogether, it may be possible to use an approximation of the knowledge that is easier to express. For example, a preference for the globally shortest hypothesis consistent with the knowledge may be difficult to express, but a preference for the locally shortest hypothesis consistent with the knowledge may be easier to express. The set representation provides a way to evaluate the cost of various approximations.

8.2 Immediate Issues

The near term goals are to take steps in this direction by developing RS-KII translators for the knowledge used by additional induction algorithms, and to identify set representations that guarantee polynomial-time identification.

Developing RS-KII translators for additional knowledge sources is needed to understand the full scope of RS-KII's expressiveness, and its practicality as an induction algorithm. The expressiveness of regular sets makes it seems likely that RS-KII can subsume a number of existing algorithms, but this same expressiveness suggests that RS-KII's complexity could be worse than that of the original, more specialized algorithms. RS-KII's complexity with respect to AQ-11 and CEA is close to that of the original algorithms, and in some cases much better. The expressiveness and complexity of RS-KII, and thus its ultimate practicality as an induction algorithm, is an area for future research.

Another key issue is identifying instantiations of KII that can integrate n knowledge fragments and enumerate a hypothesis from the solution set in time polynomial in n. This would provide a tractable induction algorithm that can potentially utilize a range of knowledge other than examples. Additionally, the set representation for the instantiation effectively defines a class of knowledge from which hypotheses can be induced in polynomial time. This would complement the results in the PAC literature, which deal with polynomial-time learning from examples only (e.g.,[Vapnik and Chervonenkis, 1971], [Valiant, 1984], [Blummer *et al.*, 1989]).

Finally, the order in which an induction algorithm searches the hypothesis space is an implicit bias of the algorithm. Hypotheses that occur earlier in the search are preferred over those that come later, since an induction algorithm usually selects the first hypothesis it finds that also satisfies the goal conditions. It is often difficult to express search orderings in terms of binary preferences between hypotheses. In order to determine whether one hypothesis comes before another in the search, it is often necessary to emulate the search. In a few cases, such as best first or hill climbing in certain hypothesis spaces, it is possible to extract the search order from the hypotheses themselves. In a best first search, this is a simple matter of determining which hypothesis has the better evaluation. In a hill climbing search, it is more awkward, as evidenced by the LEF translator for AQ-11 described in Section 5.2.2.

The issue of expressing search order biases needs to be investigated in more detail. One way to circumvent this problem is to replace the search-order bias with a bias that can be more naturally expressed as an $\langle H, C, P \rangle$ tuple. Since the search order is often an approximation of a more restrictive bias, an alternate approximation may be well justified. In the case of AQ-11, the strict bias is to prefer hypotheses that maximize the LEF. The cost of finding such a hypothesis is prohibitive, so AQ-11 uses a beam search to find a locally maximal hypothesis. It may be possible to find some other approximation of the LEF that can be expressed more naturally as an $\langle H, C, P \rangle$ tuple.

Chapter 9

Conclusions

KII is a framework for integrating arbitrary knowledge into induction. This theoretically allows all of the available knowledge to be utilized by induction, thereby increasing the accuracy of the induced hypothesis. It also allows hybrid induction algorithms to be constructed by "mixing and matching" the knowledge and implicit biases of various algorithms.

Knowledge is expressed uniformly in terms of constraints and preferences on the hypothesis space, which are expressed as sets. Theoretically, just about any knowledge can be expressed this way, but in practice the set representation determines what knowledge can be expressed, and the cost of inducing a hypothesis from that knowledge. This reflects what seems to be an inherent trade-off between the computational complexity of induction and the breadth of utilizable knowledge.

Instantiations of KII at various trade-off points between complexity and expressiveness can be generated by selecting an appropriate set representation. The space of possible set representations can be mapped onto the space of grammars. This provides a principled way to investigate the space of possible trade-offs, and to establish the trade-off limits. One such limit is that $(C \times C) \cap P$ can be at most context free, which effectively limits C to the regular languages, and P to the context free languages. If the ability to integrate knowledge is sacrificed, C can be context free and P can be regular. Otherwise, the solution set is not computable, so it is not possible to induce a hypothesis from the knowledge.

This expressiveness bound also applies to search in general, and by extension to other induction algorithms. The C set corresponds to the goal conditions, and the P set to the relative "goodness" of goals. The solution set consists of the best

goals. If C and P are too expressive, then it is not possible to find the best goal. The goal conditions must be relaxed, or the goodness ordering must be altered to allow sub-optimal goals. To the extent that other induction algorithms use search to identify the target concept, they are also limited by these bounds.

The vision motivating KII is the desire to integrate arbitrary knowledge into induction. The reality is that complexity tends to increase with expressiveness, and places an ultimate upper bound on expressiveness. RS-KII, an expressive instantiation of KII, was developed to test the range of knowledge that can be practically expressed and integrated within these bounds.

RS-KII can utilize the knowledge used by two disparate induction algorithms, AQ-11 and (for some hypothesis spaces) CEA. RS-KII can also utilize noisy examples with bounded inconsistency, and can utilize domain theories. This knowledge can be integrated with the knowledge from AQ-11 and/or CEA, thereby forming hybrid induction algorithms. It is likely that RS-KII can utilize the knowledge utilized by many other induction algorithms as well. This would allow RS-KII not only to subsume these algorithms, but also to form hybrid algorithms by "mixing-and-matching" knowledge from different algorithms.

Since RS-KII is expressive, it can also be computationally expensive. However, when RS-KII uses only the knowledge used by AQ-11, it's computational complexity is only slightly worse than that of AQ-11. When using only the knowledge of CEA, the complexity of RS-KII is comparable to that of CEA. For at least one collection of knowledge (Haussler's examples), the complexity of CEA is exponential, but the complexity of RS-KII is only polynomial. Similar results may obtain when using the knowledge of other induction algorithms, but developing translators for additional knowledge sources is an area for future work.

Reference List

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA., 1974.

[Blummer *et al.*, 1989] A. Blummer, A. Ehrenfect, D. Haussler, and M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the Association for Computing Machinery*, 36(4):929–965, 1989.

[Brieman *et al.*, 1984] L. Brieman, Friedman J., R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, 1984.

[Bundy *et al.*, 1985] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27, 1985.

[Buntine, 1991] W. Buntine. Classifiers: A theoretical and empirical study. In *12th International Joint Conference on Artificial Intelligence*, pages 638–644, Sydney, 1991.

[Chomsky, 1959] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2, 1959.

[Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(?):261–283, 1989.

[Cohen, 1992] W. W. Cohen. Compiling prior knowledge into an explicit bias. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Workshop*, pages 102–110, Aberdeen, 1992.

[DeJong and Mooney, 1986] G. F. DeJong and R. Mooney. Explanation based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[Dietterich *et al.*, 1982] T. Dietterich, B. London, K. Clarkson, and G. Dromey. Learning and inductive inference. In P. Cohen and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Volume III.* William Kaufmann, Los Altos, CA, 1982.

[Fisher, 1936] R. Fisher. The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7:179–188, 1936.

[Flann and Dietterich, 1989] N. S. Flann and T. G. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226, 1989.

[Gordon and desJardins, 1995] D. Gordon and M. desJardins. Evaluation and selection of biases in machine learning. *Machine Learning*, 20(1/2):5–22, 1995.

[Haussler, 1988] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[Hirsh, 1990] H. Hirsh. *Incremental Version Space Merging: A General Framework for Concept Learning.* Kluwer Academic Publishers, Boston, MA, 1990.

[Hirsh, 1992] H. Hirsh. Polynomial-time learning with version spaces. In *AAAI-92: Proceedings, Tenth National Conference on Artificial Intelligence*, pages 117–122, San Jose, CA, 1992.

[Hopcroft and Ullman, 1979] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[Kumar and Laveen, 1983] V. Kumar and K. Laveen. A geeneral branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

[Kumar, 1992] V. Kumar. Search, branch and bound. In *Encylopedia of Artificial Intelligence*, pages 1000–1004. John Wiley & Sons, Inc., second edition, 1992.

[Michalski and Chilausky, 1980] R.S. Michalski and R.L. Chilausky. Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems*, 4(3):219–244, September 1980.

[Michalski, 1974] R.S. Michalski. Variable-valued logic: System VL_1. In *Proeedings of the Fourth International Symposium on Multiple-Valued Logic*, Morgantown, West Virginia, May 1974.

[Michalski, 1978] R.S. Michalski. Selection of most representative training examples and incremental generation of VL_1 hypotheses: The underlying methodology and the descriptions of programs ESEL and AQ11. Technical Report 877, Department of Computer Science, University of Illinois, Urbana, Illinois, May 1978.

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

[Mitchell, 1980] T.M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, 1980.

[Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, March 1982.

[Pagallo and Haussler, 1990] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–99, 1990.

[Pazzani and Kibler, 1992] M. J. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[Pazzani, 1988] M. Pazzani. *Learning causal relationsips: An integration of empirical and explanation based learning methods.* PhD thesis, University of California, Los Angeles, CA, 1988.

[Quinlan, 1986] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Rosenbloom *et al.*, 1993] P. S. Rosenbloom, H. Hirsh, W. W. Cohen, and B. D. Smith. Two frameworks for integrating knowledge in induction. In K. Krishen, editor, *Seventh Annual Workshop on Space Operations, Applications, and Research (SOAR '93)*, pages 226–233, Houston, TX, 1993. Space Technology Interdependency Group. NASA Conference Publication 3240.

[Russel and Grosof, 1987] S. Russel and B. Grosof. A declarative approach to bias in concept learning. In *Sixth national conference on artificial intelligence*, pages 505–510, Seattle, WA, 1987. AAAI.

[Subramanian and Feigenbaum, 1986] D. Subramanian and J. Feigenbaum. Factorization in experiment generation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 518–522, Philadelphia, PA, August 1986.

[Valiant, 1984] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Vapnik and Chervonenkis, 1971] V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[Warshall, 1962] S. Warshall. A theorem on Boolean matrices. *Journal of the Association for Computing Machinery*, 9(1):11–12, 1962.

[Winston *et al.*, 1983] P. Winston, T. Binford, B. Katz, and M. Lowry. Learning physical descriptions from functional definitions, examples, and precedents. In *Proceedings of the National Conference on Artificial Intelligence*, pages 433–439, Washington, D.C., August 1983. AAAI, Morgan-Kaufmann.