REASONING WITH TAXONOMIES

by

Andrew Fall
B.Sc. Simon Fraser University 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Andrew Fall 1996
SIMON FRASER UNIVERSITY
December 1996

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Andrew Fall

Degree: Doctor of Philosophy

Title of thesis: Reasoning with Taxonomies

Examining Committee: Dr. David Fracchia
Chair

____
Dr. Veronica Dahl, Senior Supervisor

____
Dr. Ken Lertzman, Supervisor

____
Dr. Fred Popowich, Supervisor

____
Dr. Hassan Aït-Kaci, SFU Examiner

____
Dr. Nick Cercone, External Examiner

Date Approved: _______________________________________

Dedicated to Mom and Dad
Elizabeth Anne Fall and Stewart Temple Fall

Abstract

"We journey to learn, yet in travelling grow each day further and further from where we began"

– Wade Davis

Taxonomies are prevalent in a multitude of fields, including ecology, linguistics, programming languages, databases, and artificial intelligence. In this thesis, we focus on several aspects of reasoning with taxonomies, including the management of taxonomies in computers, extensions of partial orders to enhance the taxonomic information that can be represented, and novel uses of taxonomies in several applications.

The first part of the thesis deals with theoretical and implementational aspects of representing, or *encoding*, taxonomies. Our contributions include (i) a formal abstraction of encoding that encompasses all current techniques; (ii) a generalization of the technique of modulation that enhances the efficiency of this strategy for encoding and reduces its brittleness for dynamic taxonomies; (iii) the development of *sparse logical terms* as a universal implementation for encoding that is supported by a theoretical and empirical analysis demonstrating their efficiency and flexibility.

The second part explores our contributions to the application and extension of taxonomic reasoning in knowledge representation, logic programming, conceptual structures and ecological modeling. We formalize extensions to partial orders that increase the ability of systems to express taxonomic knowledge. We develop a generalization of equality constraints among logic variables that induces a partial order among equivalence classes of variables. For graphic knowledge representation formalisms, we develop techniques for organizing the derived hierarchy among graphs in the knowledge base. Finally, we organize abstract models of landscapes in a taxonomy that provides a framework for systematically cataloging and analyzing landscape patterns.

Acknowledgements

"No matter how much we seek, we never find anything but ourselves"
– Anatole France

My first thanks are to Marie-Ange, for tolerating my incessant drive to achieve my goals, for patiently listening to my explanations in various dialects of martian, and for enduring many lonely times while I was away at conferences. Our kitten Ash and my long-time companion lovebird Milk kept her company during my absences. One of my dreams has been to make my parents proud of my achievements. Even if they are no longer here, they share their love in my heart. They also live on in my brother Joseph, with whom I have been fortunate to have worked with on some of my research.

I wish to express gratitude to my supervisor, Veronica Dahl, without whom I would not have had the courage to let my ideas see the light of day. She has provided inspiration both professionally and personally during the course of my degree. I would also like to thank my committee, Hassan Aït-Kaci, Nick Cercone, Ken Lertzman and Fred Popowich, as well as Paul Tarau, for many enlightening conversations, and for encouraging me to pursue some of the routes I explored during my research. Thanks to all my friends and family, and to Mother Nature, who walked beside me along my path, diverting my attention to other important aspects of life.

Support for this research was initially funded by NSERC PGS-A and PGS-B Postgraduate Scholarships, and later by an ECO-Research Doctoral Fellowship. Additional support was made by Veronica's NSERC Research Grant 31-611024 and NSERC Infrastructure and Equipment Grant given to the Logic and Functional Programming Lab, where this work was primarily developed. Thanks for the use of facilities are also due to the School of Computing Science, and to the Forest Ecology Lab in the School of Resource and Environmental Management, at Simon Fraser University.

"The human race is challenged more than ever before to demonstrate our mastery - not over nature - but of ourselves"

– Rachel Carson

It is the author's wish that no military benefit be derived from any results in this thesis.

Notation

"Once you miss the buttonhole you'll never manage to button up"

– Goethe

Below are descriptions of the intended meaning of some of the symbols used in the thesis.

Partial order theory:

$\sqcap$	meet (greatest lower bound) and meet crest
$\sqcup$	join (least upper bound) and join base
$\leq, \sqsubseteq, \preceq$	partial order relations

Set theory:

$\cap, \bigcap$	intersection
$\cup, \bigcup$	union
$\subseteq$	subset (which is also a partial order relation)
$\in$	set membership

Predicate logic:

$\wedge$	conjunction
$\vee$	disjunction
$\neg$	negation
$\rightarrow$	implication
$\Leftrightarrow$	logical equivalence

Contents

Abstract **v**

Acknowledgements **vi**

Notation **viii**

1 Introduction **1**
1.1 Motivation and Summary of Thesis Results 2
1.2 Organization of Thesis 5

2 Background and Mathematical Preliminaries **6**
2.1 Partial Order Theory 7
2.1.1 Properties of ordered sets 7
2.1.2 Lattices 9
2.1.3 Order mappings and lattice completions 9
2.1.4 Lattice completions 10

Part I: Taxonomic Encoding **12**

3 The Evolution of Taxonomic Encoding **13**
3.1 Introduction 13
3.2 Encoding tree-shaped hierarchies 14
3.3 Extending trees to graphs 15
3.4 Characterizing term encodable hierarchies 15
3.5 Bit-vector encodings 17
3.6 Discussion 20

4 The Foundations of Taxonomic Encoding **21**
4.1 Setting the Stage 22
4.2 Spanning Sets 23
4.2.1 Taxonomic operations using spanning sets 24
4.2.2 Representation theory and encoding 25
4.3 Efficient Implementations of Component Mappings 26
4.3.1 Unordered implementations 26
4.3.2 Tree representations and code sharing 26
4.3.3 Logical terms 28
4.3.4 Sparse logical terms 28
4.3.5 Integer vectors 28
4.4 Infinite Suborders 30
4.5 Spanning Sets of Principal Down-sets and Up-sets 31

4.5.1 All principal down-sets . . . 31
4.5.2 Principal down-sets of meet irreducible elements . . . 32
4.6 Spanning Sets of Prime Down-sets and Up-sets . . . 33
4.7 Spanning Sets of Compound Down-sets and Up-sets . . . 34
4.7.1 Finding a minimal subsumption preserving spanning set is NP-Hard . . . 36
4.7.2 Multiple occurrences of factors . . . 37
4.8 Spanning Set Decomposition . . . 40
4.8.1 Chain decomposition . . . 41
4.8.2 Meet incompatible decomposition . . . 42
4.8.3 Meet homogeneous decomposition . . . 45
4.9 Constraints and Coreference . . . 46
4.9.1 Types of constraints . . . 46
4.9.2 Augmented spanning sets . . . 47
4.9.3 Integrating spanning sets and constraints . . . 48
4.9.4 Guarded constraints . . . 49
4.9.5 Coreference . . . 50
4.9.6 Coreference, decomposition and meet incompatibility constraints . . . 51
4.9.7 Encoding algorithms . . . 52
4.9.8 Variations . . . 53
4.10 Discussion and Conclusion . . . 53

5 Modulated Encoding **56**
5.1 Order Intervals and Modules . . . 56
5.2 Order partitions . . . 58
5.3 Modulation . . . 58
5.4 Extending modulation . . . 60
5.4.1 Lower and Upper Semi-Modules . . . 60
5.4.2 Generalized Modules . . . 61
5.4.3 Non-overlapping Modulation . . . 62
5.4.4 Overlapping Modulation . . . 63
5.4.5 Extending Modulation Algorithms . . . 64
5.5 Conclusion . . . 65

6 Encoding with Sparse Logical Terms **66**
6.1 Introduction . . . 66
6.2 Basic Sparse Terms . . . 67
6.2.1 Space requirements . . . 68
6.2.2 Unification and Implementation . . . 68
6.2.3 Variations . . . 68
6.3 Generalizing Sparse Terms for Encoding . . . 70
6.3.1 Explicit and canonical forms for sparse terms . . . 71
6.3.2 Sparse term subsumption . . . 72
6.4 Encoding with Sparse Terms . . . 72
6.5 Sparse Term Encoding . . . 73
6.6 Theoretical Justification . . . 75
6.7 Empirical Evidence . . . 77
6.8 Conclusion . . . 78

Part II: Applications and Extensions of Reasoning with Taxonomies **79**

7 Extending Partial Orders for Sort Reasoning **80**
7.1 Introduction . . . 80
7.2 Background . . . 81
7.3 Sort Reasoning . . . 81
7.3.1 Generalizing sort reasoning . . . 82
7.3.2 Clausal taxonomic specification . . . 83
7.3.3 Definitional specifications . . . 83
7.4 Sort Logic . . . 84
7.4.1 Complexity of Sort Reasoning . . . 85
7.5 Tractable subcases . . . 86
7.5.1 Containing sort reasoning complexity . . . 87
7.6 Implementing Conjunctive Sorts . . . 88
7.7 Conclusion . . . 88

8 Reference Constraints in Logic Programming **90**
8.1 Introduction . . . 90
8.2 Background . . . 91
8.3 Decoupling Coreference via Reference Constraints . . . 91
8.3.1 Notational considerations . . . 92
8.3.2 Maintaining and satisfying the reference order . . . 92
8.3.3 Example . . . 93
8.3.4 Comparison with sort hierarchies . . . 93
8.3.5 Implementation . . . 94
8.4 Individual Level Inheritance . . . 95
8.5 Conclusion . . . 98

9 Organizing the Hierarchy of Conceptual Graphs **99**
9.1 Background and Motivation . . . 99
9.2 Cardinality Constraints . . . 100
9.3 Normalization . . . 101
9.4 Spanning Tree Normal Form . . . 101
9.4.1 Pivoting . . . 103
9.5 Representing the Generalization Hierarchy . . . 103
9.5.1 Depth-first topological traversals . . . 104
9.6 Conclusion . . . 105

10 A Hierarchical Organization of Landscape Models **107**
10.1 Introduction . . . 107
10.2 Background: Neutral models . . . 109
10.3 Landscape Model Prototypes . . . 110
10.3.1 Pattern constraints . . . 111
10.4 A Hierarchy of Landscape Model Prototypes . . . 114
10.5 Conclusion . . . 116
Chapter Appendix: Formal Basis for Landscape Model Generators that Permit General Richness, LAR and Contagion Constraints . . . 117

11 Conclusion **119**
11.1 Significance of Research . . . 120
11.2 Future Research Directions . . . 121

Bibliography **124**

List of Tables

3.1 Assigning bits to elements from Figure 3.2 17

4.1 Characterization of encoding schemes in terms of spanning set of down-sets 54

6.1 Asymptotic encoding results for theoretical orders 77
6.2 Empirical results (in bits) for chess learning system [16] 78
6.3 Empirical results (in bits) for medical ontology 78

List of Figures

1.1 Research overview . . . 1

2.1 Sample ordered sets . . . 7
2.2 Example ordered set . . . 8
2.3 Example order mappings. The first (centre) mapping is order-preserving and the second (right-hand) mapping is an order-embedding. . . . 10
2.4 Example lattice mappings. Both mappings are {0,1}-homomorphisms and the second (right-hand) mapping is also order-embedding. . . . 10
2.5 Minimal completion of the ordered set in Figure 2.2 . . . 10

3.1 A tree-shaped hierarchy . . . 14
3.2 Taxonomy showing tree prefix . . . 15
3.3 Logical term encoding of a tree-shaped hierarchy . . . 16
3.4 Encoding of type hierarchy in Figure 3.2 . . . 16
3.5 Bottom-up bit-vector encoding of taxonomy in Figure 3.2 . . . 17
3.6 Compact bit-vector encoding of taxonomy in Figure 3.2 . . . 18
3.7 A modulated taxonomy and its encoding . . . 19
3.8 A subsumption only encoding . . . 20

4.1 Diamond lattice and two spanning sets . . . 24
4.2 Tree representation . . . 27
4.3 Chain partition of the ordered set in Figure 2.2 . . . 29
4.4 Meet incompatible anti-chain partition of the ordered set in Figure 2.2 . . . 29
4.5 Principal down-set encoding . . . 31
4.6 Cover tree, preorder numbering and interval encoding for the lattice in Figure 4.5 . . . 31
4.7 Meet irreducible encoding . . . 33
4.8 Principal up-set and prime down-set encodings . . . 34
4.9 Elements that cannot be in the same down-set . . . 36
4.10 Subsumption preserving encoding . . . 36
4.11 Transformation of a graph to a lattice . . . 37
4.12 Subsumption preserving encoding . . . 38
4.13 Violation of subsumption . . . 38
4.14 Example encodings that discriminate non-meet irreducible elements . . . 39
4.15 Distributed virtual time encoding . . . 41
4.16 Meet incompatible decomposition . . . 43
4.17 Logical term implementation of meet incompatible decomposition . . . 43
4.18 Transformation of a graph to a lattice . . . 44
4.19 Meet homogeneous decomposition . . . 45
4.20 Term encoding for diamond and cube lattices . . . 51
4.21 Lattice for which no augmented spanning set of down-sets can preserve meets and joins . . . 51

5.1 Types of modules . . . 58
5.2 A modulated lattice and its containment tree . . . 60
5.3 Lower semi-modules . . . 61
5.4 Generalized modulation. Lower surrogates (left) are $\{a, e, l\}$ and upper surrogates (centre) are $\{b, e, f, n\}$. 62

6.1 Encoding implementations: sparse terms generalize other techniques . . . 67
6.2 Sparse logical terms . . . 67
6.3 Binding arity in sparse terms . . . 69
6.4 Anonymous functors in sparse terms . . . 69
6.5 Attribute-value matrix using sparse terms . . . 69
6.6 Chain and anti-chain encodings . . . 75
6.7 Binary tree encoding . . . 75
6.8 Square lattice transitive closure and compact encodings . . . 76
6.9 Transitive closure encoding of a crown S_5 . . . 77

7.1 Relation between taxonomic and set operations . . . 82
7.2 Venn diagrams of clausal taxonomy specification . . . 83
7.3 Aggregate specifications . . . 84
7.4 Using sort definitions to represent an instance of 3-SAT: $f = c_1 \wedge \cdots \wedge c_k$, where $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, $1 \leq i \leq k$ . . . 85

8.1 State of the reference order at various points in a predicate evaluation . . . 94
8.2 Reference order for separating the contexts for a person named John . . . 95
8.3 Reference order for ambiguous parses of "*Jack saw a dog on his way home*" . . . 96
8.4 Reference order during parse of the sentence "*When Sherry saw the chair, she shook her hand*" . . . 97
8.5 Reference constraints for default reasoning . . . 98

9.1 Conceptual graph representing "a cat sitting on a mat" . . . 100
9.2 Spanning tree normal form . . . 102
9.3 A cyclic graph and a tree representation . . . 103
9.4 A woman eating a dinner cooked by her husband . . . 103
9.5 Examples of pivoting the graph in Figure 3 . . . 104

10.1 Example neutral models. Each instance was generated on a 30×30 grid (m=30), with varying proportions of the white feature (p = 0.4, 0.6 and 0.8). . . . 109
10.2 Instances of landscape model prototypes produced on a 100×100 grid. Each model has four features with equal landscape area ratios (i.e. equal relative proportions). The value of contagion differs for each model instance, taking on the values 0.6, 0.8 and 0.99, respectively. The prototype for instance (a) is therefore $\{LAR = (0.25, 0.25, 0.25, 0.25)$, $size = 100 \times 100$, $richness \in [1, 4]$, $contagion = 0.6\}$. . 112
10.3 Geometric view of an instance of a landscape model prototype with spatial constraints. The instance is overlaid on the elevation model used to create it. The model size of this instance is 100×100, and the number of features is 5. The underlying elevation model provides a context in which spatial constraints, in the form of elevation responses, affect pattern generation. Thus, the prototype for instance (a) is $\{size = 100 \times 100$, $richness \in [1, 5]$, *spatial responses to elevation*$\}$. . . . 113
10.4 Instance of a landscape model prototype (b) generated using stochastic temporal constraints and input pattern (a). The model size is 30×30, and richness is 4. The prototype for instance (b) is therefore $\{size = 30 \times 30$, $richness = 4$, *temporal responses*$\}$. . . . 114
10.5 Sample fragment of the hierarchy of landscape model prototypes. Each node represents a prototype that consists of the constraints labeling the node and all higher nodes in the hierarchy. . . . 115
10.6 Sample fragment of the hierarchy of landscape model generators. Each node represents a generator that permits specification of the constraints labeling the node and all higher nodes in the hierarchy. . 115

Chapter 1

Introduction

"In all things of nature, there is something of the marvelous"
– Aristotle

The drive to categorize and organize knowledge has been ubiquitous throughout human intellectual development. Taxonomic knowledge was first formalized by Aristotle, who proposed to define the intention of a complex concept in terms of its *genus*, or general type, and *differentia*, or specific properties. It is therefore natural that a large portion of current knowledge is taxonomically related, and that taxonomies are prevalent in a multitude of fields.

In this thesis, we are concerned with research on the efficient representation and use of taxonomies, extending partial orders for taxonomic knowledge representation and reasoning, and applying taxonomies to a variety of applications. Central to this research is the partial order (Figure 1.1).

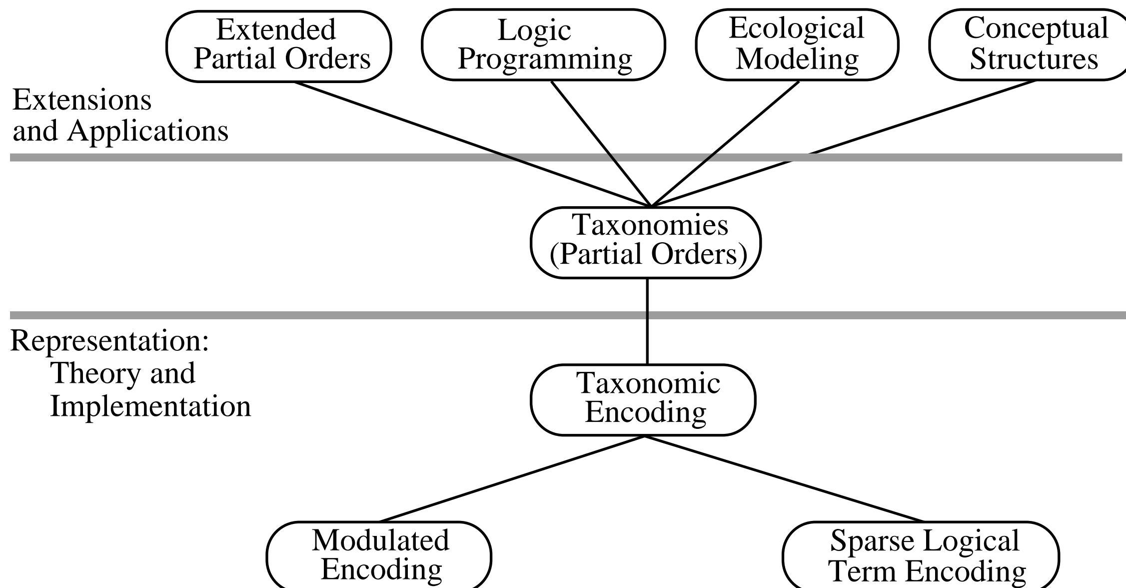

Figure 1.1: Research overview

The motivation for this thesis is based on the following observation:

Observation *Taxonomic knowledge is a useful artifact for organizing many aspects of human thought, much of which can be captured in a mathematically elegant way with partial orders. The capability of automated systems depends on the identification, application and efficient organization of taxonomic information.*

Due to the multi-disciplinary nature of this thesis, we pose a number of specific theses to explore this observation:

Thesis 1 (Taxonomic encoding) *: There exists a formal characterization for the representation, or encoding, of partial orders in computers as the expression of certain aspects of taxonomic information that is distinct from the manner in which that information is implemented.*

Thesis 2 (Modulation) *: Concepts naturally group into related, but not necessarily independent, partitions, and this can be exploited to decompose large taxonomies into manageable units.*

Thesis 3 (Sparse term encoding) *: There exists a universal encoding implementation that combines the advantages of other implementation techniques.*

Thesis 4 (Extending partial orders) *: Partial orders can be extended with taxonomic information beyond subsumption, and this can enrich the expressive power and consistency of a taxonomic reasoner.*

Thesis 5 (Reference constraints) *: The symmetry of equality constraints can be decoupled into two asymmetric reference constraints that induce a novel and practical hierarchy on equivalence classes of logical variables.*

Thesis 6 (Conceptual graph generalization hierarchy) *: Knowledge-bases of graphs that exhibit a derived hierarchical structure can be organized as a spanning tree that permits improved traversal efficiency for operations on that hierarchy.*

Thesis 7 (Landscape ecology: hierarchy of landscape models) *: Generators of landscape models can be viewed as imposing sets of constraints on pattern generation. These sets of constraints induce a hierarchy on landscape models that serves as an organizational framework for model generators and for the analysis of landscape patterns.*

1.1 Motivation and Summary of Thesis Results

We motivate the thesis by discussing a number of open problems that we focused our research efforts on, and some of the significant results that we obtained. This thesis crosses a number of disciplinary boundaries, and advances the state of the art in several different fields. The list below follows somewhat the structure of this thesis.

1. **Encoding:** Mellish [102] studied the use of logical terms for encoding lattices. He characterized the classes of lattices for which term encodings were *possible* for different forms of terms (e.g. flat terms). However, no algorithm was presented, and so no constructive solution to the problem of encoding was proposed.

   On the other hand, researchers advocating the use of bit-vectors and related approaches have applied encoding in real applications (e.g. object-oriented programming [24], operating systems [97]). However, these approaches have been ad hoc, and no formal apparatus has emerged to permit objective comparison and evaluation of the different techniques.

   We develop a formal apparatus for objectively characterizing all encoding algorithms. Our framework permits the separation of the informational content of an encoding from its implementational details, and allows comparison at an abstract level of different encoding techniques. Furthermore, the advantages and disadvantages of various approaches for implementing encodings can be analyzed for their effect on space and time efficiency, and their dynamic behaviour.

2. **Modulation:** Modulation is a well-known technique for the analysis of partial orders in discrete mathematics (e.g. [60]), but it wasn't until the seminal work of Aït-Kaci *et al.* [2] that its use for encoding was proposed. The algorithm proposed in this paper produces an approximate modulation in a time efficient manner. Researchers on partial order theory, on the other hand, have worked on exact modulation algorithms, but it was only recently that an efficient (linear) algorithm was developed [76]. Even with the ability to decompose taxonomies

into modules, however, the ability to take advantage of modulated taxonomies has received limited attention beyond the proposal in [2].

An additional issue, and perhaps more important, is that modules are rigidly defined constructs. Even if adequate modulations are possible in real taxonomies, dynamic updates have the potential to invalidate much of the work involved in modulation. Prior to our research, no proposal had been made to address this serious issue that undermines the potential advantages of modulated encoding by making modules too brittle for real applications.

Taking advantage of the decomposition tree of a modulation, we develop a technique for modulated encoding that reduces the size of codes, and the time to compute taxonomic operations, beyond that proposed in [2]. Furthermore, our abstract treatment of modulation permits a direct generalization to a relaxed definition of modules that degrade gracefully under dynamic updates. We design algorithms for operations on generalized modules, which we prove to be correct.

3. **Logical term encoding:** The viewpoint taken in the analysis of Mellish [102, 104] is: given a technique for implementing encodings, what forms of taxonomies can be encoded? We feel that, for real-world problems, this viewpoint is flawed. In applications that require encoding, we may not have the luxury to restrict the form of a taxonomy to encode. Thus, we believe that a better viewpoint is: given a taxonomy, what is the *best* approach to encode this taxonomy? This stance makes it easier for people to describe things naturally, and does not overly constrain their expressive power. We highlight “best”, since there are a number of criteria by which we may evaluate encoding. The most prevalent criterion is the size of the resulting codes, although we discuss others later.

   Due to the structural potential and flexibility of logical terms, we feel that term encodings are the most promising form of implementation. For example, logical terms may permit dynamic updates to a portion of a taxonomy without requiring a full re-encoding, while any change to the length of a bit-vector encoding requires updating every code. However, prior to research conducted for this thesis, no algorithms for encoding with logical terms had been proposed.

   Our early attempts at logical term encoding using Prolog terms were unsuccessful due to the vast number of anonymous variables that produced excessively large terms. For this reason, we developed and implemented *sparse logical terms* for the specific task of logical term encoding, although we later found other uses for them. Sparse terms vastly improved our term encoding results, but we later discovered how the benefits of encoding with logical terms, integer vectors and interval sets could be integrated into an extended form of sparse term.

   In this thesis, we propose these extended sparse terms as a *universal* encoding implementation that encompasses (in terms of efficiency) most other approaches to implementing encodings, and we devise and implement the first published logical term encoding algorithms. This claim is backed up by theoretical comparisons of sparse terms with other approaches to encoding, as well as an empirical comparison between bit-vectors and sparse terms for encoding two medium size taxonomies from existing applications. Even though each item of information in a sparse term uses more space in an absolute sense (i.e. one atom vs. one bit), sparse terms outperformed bit-vectors by nearly an order of magnitude. This result is strengthened by the improved flexibility obtained by the use of logical terms over more rigid implementations such as bit-vectors.

4. **Extending partial orders:** The maintenance of taxonomic knowledge has been polarized. At one extreme, systems use mathematically pure, but limited, partial orders for representing taxonomic information. At the other extreme, terminological systems provide rich formalisms for specifying knowledge, and taxonomic information is derived through the expensive (and potentially intractable) operation of *classification* [18, 19, 159]. In order to gain efficiency, some terminological systems limit expressive power to obtain tractable classification. However, there has been no corresponding push in the other direction, namely to embellish partial orders with further power to incorporate additional forms of taxonomic knowledge other than simple subset information.

   One of the dangers of this situation is that taxonomic operations, such as meets, have been interpreted as equivalent to conceptual, or set-theoretic operations, such as intersection. Although this correspondence appears

natural, it may lead to invalid inferences, as pointed out in [28] in the context of many-sorted logic.

The solution to this problem suggested in [28] is to embed the taxonomy in a special Boolean lattice that provides consistent inferences. This is adequate for logic, but inadequate for applications that must reason efficiently with taxonomic knowledge, due to a potentially exponential increase in space. We analyze sort reasoning as a distinct reasoning task, and suggest the inclusion of a sort reasoner in applications that utilize taxonomic knowledge. By developing a sound and complete *sort logic* (not a sorted logic for reasoning with sorts, but a logic for reasoning about sorts), we clearly identify the task required as the *sort reasoning problem*. We prove that this problem is NP-Hard, but analyze the sources of intractability. By limiting certain forms of taxonomic declarations and queries, we show that intractability can be bounded, resulting in a sort reasoning procedure that only requires polynomial time.

5. **Reference constraints:** During the development of a constraint based view of encoding, we identified the utility of constructing a hierarchy of logical variables (actually, of equivalence classes of variables). In this way, unification can be split into two uni-directional components that allows, for example, updates to a variable X to be automatically unified with variable Y, but not vice versa. This form of relation among logical variables has not been previously proposed.

   We develop a formal description of *reference constraints*, and show how they may be specified in a logic program. We also explain how the resulting hierarchy of equivalence classes is maintained and satisfied during the processing of a logic program. Finally, we discuss how reference constraints can be implemented, and propose potential modifications to the control strategy of logic programming languages that may take fuller advantage of this new form of constraint.

   While working out the details of reference constraints among logical variables, we identified a broad area of application in hypothetical reasoning systems. Reference constraints naturally lead to the notion of *individual-level inheritance*. The classical notion of inheritance involves inheritance of properties among classes (e.g. the class *cat* inherits properties from the class *mammal*) and from classes to individuals (e.g. the cat *Ash* inherits properties from the class *cat*). Individual-level inheritance is a novel and distinct form of inheritance among individuals, which are approximated by terms in logic programming. If individual A inherits from individual B, then the term that approximates A must be more fully specified than the term that approximates B. This notion has applications in systems that reason with uncertainty, to separate, but relate, hypothetical from known information in a given context.

6. **Conceptual structures:** *Conceptual structures* is a graph-based formalism for knowledge representation that relies heavily on taxonomies. The type and relation lattices are declarational structures to which encoding techniques are directly applicable. The *generalization hierarchy*, however, is a partial order formed by graphs using the complex operation of *projection*, which is akin to classification in terminological representations such as KL-ONE [18]. Essentially, one graph subsumes another if the former contains a subset of the information of the latter. However, the computation of this *derived* taxonomy is expensive, and the taxonomy itself is highly dynamic as changes to the knowledge base transpire. To organize this hierarchy, a number of techniques, including encoding [42], have been proposed, although research on this problem is ongoing.

   We develop a solution that takes advantage of the information content of graphs to organize the generalization hierarchy. Graphs are preprocessed using some *normalization* techniques to produce a standard form, called *spanning tree normal form* due to the representation of a graph as a tree with coreference links. The generalization hierarchy itself is also organized as a tree, and graphs are further normalized into *generalization hierarchy normal form* as they are inserted into the tree. The advantage of this tree form is that the projection operation between a node and its parent in the tree is greatly simplified, so traversals down branches are less costly than general traversals in the hierarchy. Furthermore, in [42] it is argued that the most efficient traversals of the generalization hierarchy are topological. We show that, given a spanning tree produced from a left-to-right depth first traversal of a partial order, a right-to-left depth first traversal of this tree corresponds to a depth first topological traversal of the partial order.

7. **Landscape ecology: model generation** Work on theoretical models of landscapes, known as *neutral models*, has proceeded steadily over the last few years (e.g. [25, 66, 67, 148]), but is now rapidly expanding, as the number of presentations that focused on neutral models at a recent landscape ecology symposium testifies (e.g. [64, 73, 83, 100, 157]). However, although the development and use of neutral models and neutral model generators has proliferated, no unifying framework for organizing and categorizing models has emerged.

   By defining the general notion of a *landscape model prototype*, we provide a formal framework for describing and comparing theoretical landscape models and model generators. A landscape model prototype describes an *expected pattern* in the absence of additional ecological information, and so defines a distribution of landscape patterns in a multi-dimensional space of possibilities. Using this notion, a hierarchy of prototypes is induced; near the top are general prototypes that correspond to neutral models, while lower down are more predictive models. Overall, the hierarchy clarifies gradients of neutrality in landscape models, and can be used to aid selection of existing landscape model generators, in guiding the development of new model generators, and for analyzing data sets of landscape models with respect to the degree of neutrality.

1.2 Organization of Thesis

The thesis is divided into two major parts. In Part I we look at some theoretical and implementational issues for representing taxonomies, while part II considers several applications and extensions of reasoning with taxonomies. The following chapter provides relevant background information for the thesis. In particular, some basic partial order theory is presented as well as deviations from standard theory that we found important for our research. Due to the diversity of topics covered, each chapter will also present background material and related work important to the chapter.

Part I, *taxonomic encoding*, is divided into four chapters that contain research on various aspects of this topic. Historical developments in taxonomic encoding are described in Chapter 3. In Chapter 4, we provide an in-depth study of encoding and develop our framework for formalizing encoding. We describe our generalizations of modulation in Chapter 5. In Chapter 6 we develop sparse logical terms as a universal encoding implementation. Theoretical and empirical evidence is presented to support this position.

Part II is divided into four chapters pertaining to research on extensions to, or applications of, reasoning with taxonomies. In Chapter 7, we present results on extending the mathematical notion of a partial order to enhance the ability to represent taxonomic knowledge. In chapter 8, we describe an application of partial orders in logic programming for generalizing equality constraints among logical variables. We present the use of taxonomies in conceptual structures in Chapter 9. In particular, we focus on techniques for organizing the generalization hierarchy induced by conceptual graphs, including graph normalization and a spanning tree representation of this hierarchy. Finally, we show in Chapter 10 how a partial order can be defined among abstract models of landscapes in order to enhance the organization and specification of generators of landscape models, and the analysis of data sets of landscape models.

Chapter 2

Background and Mathematical Preliminaries

"From here on down, it's uphill all the way"

– Walt Kelly

The cohesive theme of this thesis is the partial order, a simple yet elegant and powerful mathematical concept to which a lot of attention has been devoted (e.g. [15, 38, 144]). Partial orders underlie central aspects of many domains, such as mathematical logic [128], sorted logic [27, 28, 93] and logic programming [3, 4, 93], type systems [106], natural language processing (e.g. typed feature structures [23, 71, 118], systemic networks [80, 101]), object-orientation (e.g. databases [1], languages [24]), knowledge representation (e.g. conceptual structures [42, 136], knowledge bases [45], description logics [17, 18, 159], default inheritance and non-monotonic reasoning [22, 85, 143, 151]), machine learning (e.g. description identification [103] and concept learning [108, 156, 161]), formal concept analysis [153, 155], distributed systems [97], and ecology and ecological modeling [8, 11, 39, 75, 115].

As the size of partial orders increases, it becomes important to find representations that are both space efficient, and support fast execution of desired operations (e.g. greatest lower bounds). Suitable *encoding* techniques will depend on the nature of these partial orders (e.g. whether they can change dynamically, whether certain properties such as distributivity or bounded width are satisfied) and the operations to be supported. Research on *taxonomic encoding* has explored a variety of possibilities (e.g. [2, 24, 34, 35, 43, 45, 61, 77, 78, 79, 93, 97, 101, 102, 104, 114]).

In order to empower logical terms for encoding, we developed sparse terms [51], based on an analogy to sparse matrices. There are many similarities, but also some important differences, between sparse terms and ψ-terms in LIFE [4], as well as sorted feature structures [23, 118].

Although mathematically clean, partial orders limit the representation of taxonomic knowledge to subsort-supersort (or *isa*) relationships. We cannot, for example, directly state that two sorts are incompatible or define one sort as the intersection of a set of other sorts. This poses problems for specifying more complete taxonomic relationships as well as for denotational semantics in sorted logic [28]. Research on many sorted logics has addressed this issue by expanding the expressive power of relationships among sorts. In simple many sorted logics the sorts simply partition the domain of discourse, while more complicated logics permit much more expression [28].

The potential applications in which we could explore reasoning with taxonomies are many. We choose to focus on logic programming, conceptual structures and ecological modeling. An important application that we only explore superficially is natural language processing, where important uses of taxonomies include lexical specification and typed feature structures (e.g. [23, 118]). We have also used taxonomies in the resolution of anaphora and co-specification in discourse processing [54] (synthesizing and extending research in [9, 84, 133]), and for hypothetical reasoning [36].

Equality constraints partition logical variables into coreference classes, each of which denotes an individual (which may be unspecified or partially specified) in a domain of discourse. These constraints form a basis for a number of logic programming languages, such as Prolog [138] and LIFE [4]. However, the resulting classes are unrelated to each other. Our application is the exploration of a generalization of equality constraints that induces a partial ordering

among coreference classes.

Conceptual structures [136] is a rich application for taxonomies. Taxonomic encoding has been proposed for the type lattice [35], and for the generalization hierarchy of graphs [41, 42]. Other research has analyzed normalization techniques for conceptual graphs [107, 160]. Our focus is on the use of normalization techniques for a novel and efficient organization of the generalization hierarchy.

Landscape ecology [58] and ecological modeling are also prime application areas for taxonomies, particularly for spatially explicit population models [40], ethology (animal behaviour) models [39], individual-based modeling [14, 57, 130], and intelligent simulation [105, 110, 124, 129]. Our focus is on spatially explicit models of landscapes [10, 135, 146]. Work on theoretical landscapes has shown that models which contain no or very little ecological information, known as *neutral models*, provide a null hypothesis for landscape pattern and change [66, 67, 148, 145, 147, 150, 149]. We have extended this notion to provide an incremental path from neutral models to landscape models that incorporate ecological information, and possibly real data (e.g. from a GIS), inducing a partial ordering among landscape models [55, 56].

2.1 Partial Order Theory

Since the core of this thesis revolves around the partial order, it is important to have a clear understanding of the underlying mathematics upon which much of this research rests. In this section, we present some basic partial order theory, as can be found in [38], or any other lattice theory text. Definitions and theorems that introduce our additions to, or deviations from, standard theory will be followed by an asterisk.

A *(partially) ordered set* is a pair $(P, \leq)$ where P is a set and $\leq$ is a reflexive, transitive and anti-symmetric binary relation defined on P. Often, we leave $\leq$ implicit and simply call P an ordered set. We call $\leq$ *subsumption*, and use subscripts (e.g $\leq_P$) to disambiguate different orders. If $x \leq y$ or $y \leq x$, then we say that x and y are *comparable*. We denote that x and y are incomparable by $x||y$. If $x \leq y$ but $x \neq y$, we write $x < y$. We say that x is *covered by* y, or y *covers* x, if $x < y$ and $x \leq z < y$ implies that $x = z$. Genealogical terms are also used: if $x \leq y$, then we say x is a *descendant* of y, or y is an *ancestor* of x. If x is covered by y, then we say x is a *child* of y, or y is a *parent* of x.

An ordered set P is a *chain* (or *total order*) if $\forall x, y \in P$ either $x \leq y$ or $y \leq x$; P is an *anti-chain* if $\forall x, y \in P$ $x \leq y$ implies that $x = y$ (i.e. if $x \neq y$ then $x||y$). Any subset Q of P is a *suborder* if, for any $x, y \in Q$, $x \leq_Q y$ if and only if $x \leq_P y$.

Examples of ordered sets include families of subsets of some domain X ordered by set inclusion (i.e. $A \leq B$ if and only if $A \subseteq B$), sets of integers ordered by divisibility (i.e. $x \leq y$ if and only if x is a factor of y), and logical term spaces ordered by term instantiation. An example of a total order is the set of integers ordered by relative magnitude. Ordered sets can be shown diagrammatically (in *Hasse diagrams*) by placing subsuming elements above subsumed elements and only drawing arcs in the transitive reduction, as shown in the samples below.

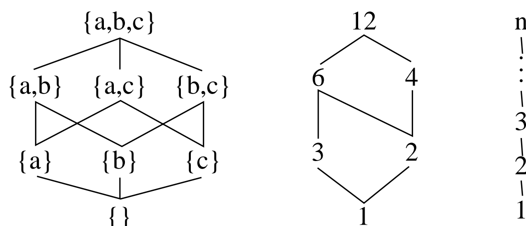

Figure 2.1: Sample ordered sets

2.1.1 Properties of ordered sets

We define the *dual* P^∂ of an ordered set P by reversing $\leq$. We similarly define the dual of a statement regarding ordered sets. The *Duality Principle* allows us to deduce the dual of a statement once the statement itself is proven.

Suppose we have a subset Q of an ordered set P. Then $q \in Q$ is a *maximal* element of Q if $q \leq x \in Q$ implies that $q = x$, and q is the *greatest* (or maximum) element of Q if $q \geq x$ for every $x \in Q$. Minimal and least elements are defined dually. The set of maximal (minimal) elements of a set Q is denoted as $\lceil Q \rceil$ ($\lfloor Q \rfloor$). If P has a greatest

(least) element, we call it *top* (*bottom*), denoted by $\top$ ($\perp$). If P has both $\top$ and $\perp$, then we call P *bounded*. An element $x \in P$ is an *upper (lower) bound* of Q if $q \leq x$ ($x \leq q$) for every $q \in Q$. The set of all upper (lower) bounds of Q is denoted Q^u (Q^l).

Definition 2.1 *Let P be an ordered set and Q a subset of P. If Q^u has a least element x, then x is called the* join *or* least upper bound *of Q, denoted $\sqcup Q$. If Q^l has a greatest element x, then x is the* meet *or* greatest lower bound *of Q, denoted $\sqcap Q$*[1].

If Q has exactly two elements, x and y, then $\sqcup\{x, y\}$ and $\sqcap\{x, y\}$ may be written $x \sqcup y$ and $x \sqcap y$, respectively. The join $x \sqcup y$ may fail to exist because x and y have no common upper bound or because they have no *least* upper bound (i.e. $\lfloor\{x, y\}^u\rfloor$ is not a singleton). In the former case we call x and y *join incompatible*, and if x and y have no common lower bound they are called *meet incompatible*. Note that in a finite ordered set, there exists a non-unique meet if and only if there exists a non-unique join. In the ordered set in Figure 2.2, we can see that $dog \sqcup wild$ doesn't exist, while $dog \sqcap wild = feral\ dog$.

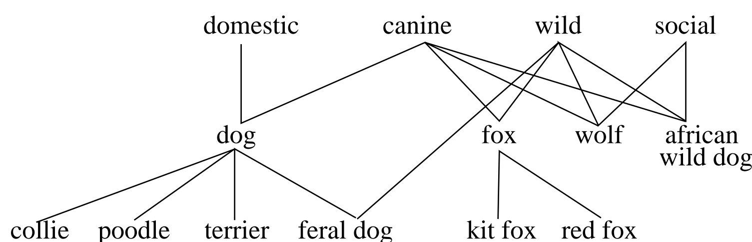

Figure 2.2: Example ordered set

Definition 2.2 (*) *Let P be an ordered set and Q a subset of P. The set of minimal upper bounds of Q is called the* join base *of Q and the maximal lower bounds of Q is the* meet crest *of Q.*

By abuse of notation, we denote lower bound, or meet, crests the same as meets (and upper bound, or join, bases the same as joins), although the result is a set, not a single element. Thus, in Figure 2.2, neither $dog \sqcap fox$ nor $dog \sqcup wild$ exist, but $wild \sqcap social = \{wolf, african\ wild\ dog\}$ and $fox \sqcup wolf = \{canine, wild\}$.

Definition 2.3 *Let P be an ordered set and Q a subset of P. Then Q is a* down-set *if for $x \in Q$ and $y \in P$, $y \leq x$ implies $y \in Q$.* Up-sets *are defined dually.*

We can construct the smallest down-set containing a set Q as $\downarrow Q = \{y \in P | \exists x \in Q, y \leq x\}$. If Q consists of the single element x, we write $\downarrow x$. Note that Q is a down-set if and only if $Q = \downarrow Q$. As an example, in the second ordered set in Figure 2.1, $\downarrow 6 = \{6, 3, 2, 1\}$. The family of all down-sets of an ordered set P is denoted by $\mathcal{O}(P)$, and is ordered by set-inclusion. A down-set with a single maximal element is called *principal*, otherwise it is *compound*. Compound down-sets can be viewed as the union of a set of principal down-sets. Note that if P is an anti-chain, then $\mathcal{O}(P) = 2^P$ (the power set of P). In general, $\mathcal{O}(P) \subseteq 2^P$ and is much smaller for most ordered sets.

There is a complementary correspondence between down-sets and up-sets, as formalized in the next theorem. Note that we use the symbol "\" for the set difference operator.

Theorem 2.1 (*) *If $\downarrow Q$ is a down-set in an ordered set P then $P \backslash \downarrow Q$ is an up-set.*

Proof: If e is not in the down-set, then it is not subsumed by any element in Q. So every ancestor of e is also not in the down-set. Thus, this complement is an up-set. □

When P is finite, every non-empty set $\downarrow Q \in \mathcal{O}(P)$ can be characterized by its maximal elements, called the *factors* of the down-set. In a *canonical* down-set $\downarrow Q$, every pair of elements in Q is incomparable (i.e. they form an anti-chain) and is thus the set of factors of $\downarrow Q$. Hereafter, we assume that all down-sets are canonical.

[1]Some order theory texts use $\wedge$ and $\vee$ to denote meets and joins. (e.g. [38]). These symbols, however, conflict with the symbols traditionally used in predicate logic for conjunction and disjunction. The symbols $\sqcap$ and $\sqcup$ are also used in order theory, and provide a more consistent notation.

2.1.2 Lattices

Definition 2.4 *Let L be a non-empty ordered set. If joins and meets exist for every $x, y \in L$, then L is a* lattice*. If the join and meet exists for every subset $S \subseteq L$, then L is a* complete lattice.

Every complete lattice must be bounded and every finite lattice is complete [38] (since the meet of any set can be expressed as the successive meets of pairs of elements). An example of a lattice is 2^X for a set X, ordered by set inclusion. Also, if P is an ordered set, $\mathcal{O}(P)$ is a complete lattice ordered by set inclusion. All of the examples in Figure 2.1 are lattices, but the example in Figure 2.2 is not. Note that the dual of a statement regarding lattices is obtained by interchanging $\sqcap$ and $\sqcup$ in addition to reversing the order relation.

Definition 2.5 *A non-empty down-set $\downarrow Q$ of a lattice L is an* ideal *if $a, b \in \downarrow Q$ implies $a \sqcup b \in \downarrow Q$.*

Thus, an ideal is a down-set that is closed under join. A dual ideal is called a *filter*. An ideal $\downarrow Q$ is called proper if $Q \subset L$. For each $a \in L$, $\downarrow a$ is an ideal and $\uparrow a$ is a filter, respectively called the *principal* ideal and *principal* filter induced by a. Thus, every principal down-set is an ideal. Also, in a finite lattice, every ideal or filter is principal [38].

Definition 2.6 *Let L be a lattice and Q a proper ideal in L. Then Q is a* prime *ideal if $a, b \in L$ and $a \sqcap b \in Q$ implies $a \in Q$ or $b \in Q$. A prime filter (ultrafilter) is defined dually.*

Definition 2.7 *Let P be an ordered set and $e \in P, e \neq \top$. Then e is* meet irreducible *if $x \sqcap y = e$ implies that $x = e$ or $y = e$.*

Thus, e is meet irreducible if it is not the (unique) meet of any set of elements not containing e. *Join irreducible* elements are defined dually. We represent the set of meet and join irreducible elements by $\mathcal{M}(P)$ and $\mathcal{J}(P)$, respectively. In a lattice L, the meet (join) irreducible elements are the elements that have exactly one parent (child). For ordered sets, however, the set of meet (join) irreducible elements is not as easily identified.

Theorem 2.2 (*) *Let P be an ordered set. Then an element $x \in P$ is meet irreducible if and only if the set of parents A of x is a singleton or has a non-singleton meet crest.*

Proof: Let x be an element of P and let A be the set of parents of x.

$\Rightarrow$ If A is not a singleton and has a singleton meet crest, then the meet is x, so x is not meet irreducible.

$\Leftarrow$ Suppose A is a singleton or has a non-singleton meet crest. In the former, x is clearly meet irreducible. For the latter case, suppose x is non-meet irreducible. Then there is a set of elements Q for which $\sqcap Q = x$. Let A' be the elements of A subsumed by some element of Q. It follows that $\sqcap A' = x$. Clearly $x \in \sqcap A$. Consider any lower bound b of A. Since b is also a lower bound of A', $b \leq x$. Thus x is the greatest lower bound, so A has a unique meet. □

2.1.3 Order mappings and lattice completions

Definition 2.8 *Let P and Q be ordered sets. A map $\varphi : P \to Q$ is*

i. order-preserving *(or monotone) if $x \leq y$ in P implies $\varphi(x) \leq \varphi(y)$ in Q.*

ii. an order-embedding *if $x \leq y$ in P if and only if $\varphi(x) \leq \varphi(y)$ in Q.*

iii. an order-isomorphism *if it is an order-embedding mapping P onto Q (denoted as $P \cong Q$).*

Note that if φ is an order-embedding, then $\varphi(P) \cong P$. Also, an order-embedding is one-to-one, so its inverse is a partial function, and an order-isomorphism is bijective, so its inverse is a total function. Figure 2.3 shows an ordered set and example order-preserving and order-embedding mappings. Two order-isomorphic sets must have isomorphic diagrams.

Definition 2.9 *Let K and L be lattices. A map $\varphi : L \to K$ is a* homomorphism *if φ is join and meet-preserving. That is, $\varphi(a \sqcup b) = \varphi(a) \sqcup \varphi(b)$ and $\varphi(a \sqcap b) = \varphi(a) \sqcap \varphi(b)$.*

A bijective homomorphism is a *lattice isomorphism*. If φ is one-to-one, then the sublattice $\varphi(L)$ of K is isomorphic to L and φ is an embedding of L into K. If $\varphi(\bot) = \bot$ and $\varphi(\top) = \top$, then it is called a $\{0,1\}$-homomorphism. Figure 2.4 shows a simple lattice and two homomorphisms, both of which happen to be $\{0,1\}$-homomorphisms. The second is also an order-embedding.

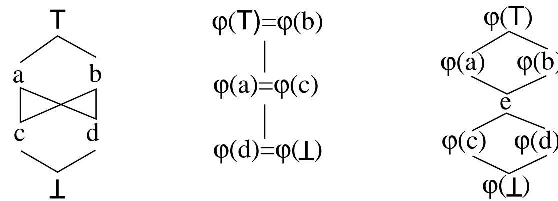

Figure 2.3: Example order mappings. The first (centre) mapping is order-preserving and the second (right-hand) mapping is an order-embedding.

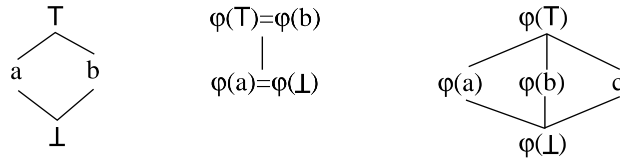

Figure 2.4: Example lattice mappings. Both mappings are {0,1}-homomorphisms and the second (right-hand) mapping is also order-embedding.

2.1.4 Lattice completions

Since many results depend on a lattice structure, we now describe how to form a lattice from an arbitrary ordered set using an order-embedding. This is known as lattice completion.

Definition 2.10 *Let P be an ordered set and L a complete lattice. If $\varphi : P \to L$ is an order-embedding, then L is a* completion *of P (via φ).*

For example, the mapping $\varphi(x) = \downarrow x$ embeds P into the complete lattice $\mathcal{O}(P)$. Other completions include the Boolean lattice completion of Cohn [28]. It is, however, possible to specify a completion of minimal size. The following definition is isomorphic to the Dedekind-MacNeille completion [38, 77] (which maps into a sublattice of $\mathcal{O}(P)$) and the completion described in [2] (which maps into a sublattice of 2^P). Recall that for ordered sets, we define the "$\sqcap$" operation to return the set of maximal lower bounds (as opposed to a single meet element).

Definition 2.11 *Let P be an ordered set and $L_P \subseteq 2^P$ be a lattice defined as follows: $A \in L_P$ if and only if $\exists a, b \in P$ for which $A = a \sqcap_P b$. For $A, B \in L_P$, $A \leq_{L_P} B$ if and only if $\forall a \in A$, $\exists b \in B$ such that $a \leq_P b$. The* minimal *lattice completion of P is the order-embedding $\varphi : P \to L_P$, where for $a \in P$, $\varphi(a) = \{a\}$.*

This lattice completion can be constructed simply by checking each pair of elements in P. If their meet is not unique, then create a new element that represents this meet. Clearly, $L_P \cong P$ if and only if P is already a lattice. We could also define a minimal completion in terms of joins, which is isomorphic for finite lattices. As an example, Figure 2.5 shows a minimal completion of the lattice in Figure 2.2, where $pack\ dog = \{wolf, african\ wild\ dog\}$ and $wild\ dog = \{feral\ dog, fox, wolf, african\ wild\ dog\}$.

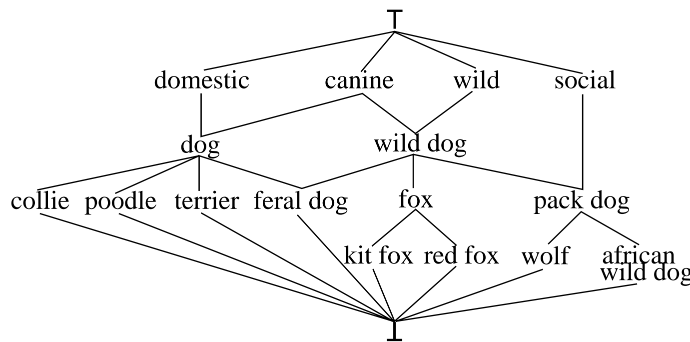

Figure 2.5: Minimal completion of the ordered set in Figure 2.2

A minimal completion can be viewed in two ways. The first is as an abstract construct that gives formal meaning to meet crests within P (by adding new nodes to stand as proxies for non-singleton meet crests). In this context, we work with the original ordered set. When computing meets, we may obtain a non-singleton meet crest, which requires additional search in the ordered set. This is the approach taken in [2] and is useful when many lattice operations are performed before output to the user is required. The second viewpoint, taken in [24, 102], is to realize the completion. Working with a lattice leads to simpler encoding algorithms and decoding schemes. Unfortunately, completion may result in adding an exponential number of elements to our original set. This problem can be alleviated somewhat using the technique of *lazy completion* in [77], where elements representing non-unique meets and joins are only added as they are computed.

An ordered set P that does not possess a $\bot$ element is called $\bot$-*unbounded*. For a lattice L, every meet in $L \backslash \{\bot\}$ exists, except those that result in $\bot$. All finite lattices must be bounded, otherwise they would not be closed under joins and meets. In many real lattices, however, $\bot$ is only implicit (e.g. as an absurd element). There are several ways that we can handle $\bot$. First, we can treat it as any other element, which is simple but may not be very satisfactory, particularly for orders that are wide or that may change dynamically. A second approach (espoused in [102]) is to treat $\bot$ as meet failure. That is, if $a \sqcap b = \bot$, then the meet operation must fail. We can also treat it as decode failure - if the code computed for a meet has not been assigned to any element, then assume it is $\bot$. These latter two approaches essentially treat the lattice as $\bot$-unbounded.

Part I:

Taxonomic Encoding

"Discovery consists of looking at the same thing as everyone else and thinking something different"

– Albert Szent-Goygyi

Chapter 3

The Evolution of Taxonomic Encoding

"In rivers, the water you touch is the last of what has passed and the first of that which comes: so with time present"

– Leonardo da Vinci

Leibniz (in [136]) initiated the quest for representations, or *encodings*, of lattices and partial orders that could be used to efficiently compute operations, such as *greatest lower bound* and *comparability*. This quest continues today, and has been an active area of research in the past few years. In this chapter, we review the developmental history of taxonomic encoding.

3.1 Introduction

Taxonomies appear in a multitude of guises and in many fields. As the size of these taxonomies increases, there is a growing need to represent them in a form that is amenable to performing operations, such as meets, efficiently. Encoding taxonomies in a manner that permits quick execution of such operations has been a goal in logic programming, and in other areas computer science, for some time now. Although many encoding schemes have been successful, research in this area is ongoing in the quest for general purpose, compact, flexible and efficient encoding techniques.

In logic programming, encodings have been used to reduce the length of the proofs needed to deduce some kinds of facts, to facilitate intensional replies and to achieve partial execution of some queries (e.g. [33, 34, 87]), and to integrate many-sorted logic [4]. In natural language processing, they have been used to permit quick semantic agreement verifications on queries, to calculate domain intersections through unification, and for incremental description refinement (e.g. [32, 98]). In systemic linguistics, these techniques have been used for representing and making inferences from systemic networks [101].

The evolution of taxonomic encoding has involved interactions among researchers working with both the logic programming and bit-vector approaches. Other techniques are introduced within our formal framework for encoding in the following chapter. The early work in the logic programming [32, 34] and bit-vector [2] directions has been expanded within [24, 96] and between [101, 102] research lines.

Schemes for encoding taxonomies so that the basic operations can be performed through unification have been studied, e.g., in [34, 98, 101, 120]. Alternative approaches involve rewriting the logic programming interpreter or compiler to extend unification to facilitate efficient encodings [52], or to encompass type operations directly [3]. Bit-vector encoding techniques can be applied using logical terms, but logical terms may possess structure not easily mimicked with bit-vectors, so the converse may not be as apparent. In general, most schemes can be abstracted from the particular space used for the codes (e.g. terms or bit-vectors) to analyze the actual taxonomic information encapsulated in the encoding.

The following sections of this chapter outline early research on encoding. The viewpoints are expressed in the form of the original research. In the next chapter, some of these approaches and other techniques are re-cast in our formal framework.

3.2 Encoding tree-shaped hierarchies

One of the early encoding techniques [33, 34] dealt efficiently with tree-shaped hierarchies (i.e. hierarchies that do not allow multiple inheritance). It was inspired by the simple observation that by representing a type t as a term $t \& t_1 \& ... \& t_n$, where we assume that the relationships $t \subset t_1, t_1 \subset t_2, \ldots, t_{k-1} \subset t_k$ hold, we can also represent partially known types by similar terms in which a variable stands for the unknown sequence of set inclusions, and then check for operations, such as set inclusion, through unification. By extending Horn-clause terms, a simple representation of taxonomic information is obtained. Essentially, a type in a such a hierarchy can be represented as the (unique) path from the root node to the type. As meets are always $\perp$ in a tree-shaped hierarchy, we are only concerned with joins and subsumption checking.

As an example, the elements *chameleon* and *dog* in Figure 3.1 can be encoded as the paths $[animal, reptile, chameleon]$ and $[animal, mammal, dog]$, respectively.

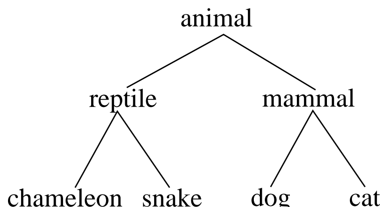

Figure 3.1: A tree-shaped hierarchy

Checking subsumption in this representation can be done by checking if the path of the subsuming label is a prefix of the path of the subsumed label. So, for example, the path of *mammal*, $[animal, mammal]$, is a prefix of that of *dog*, as *mammal* subsumes *dog*. By representing the paths as difference lists[1], this operation can be performed with a single unification. Thus, *mammal* and *dog* would actually be represented by $[animal, mammal|X]\backslash X$ and $[animal, mammal, dog|Y]\backslash Y$, respectively. If this unification fails, then the two elements are incompatible. The join operation can be achieved by simply retaining the longest common prefix of the two paths. Thus, $dog \sqcup cat$ will find the longest common prefix of $[animal, mammal, dog]$ and $[animal, mammal, cat]$ which is $[animal, mammal]$. Decoding is done by finding the label with this path, which is *mammal*. Since each element has no more than one parent, joins will always be unique.

With the difference list representation of paths, we can express *incomplete types*. That is, we store a path from the root to the most specific type known, with the possibility of extending this path as more information is obtained. For example, if we all know about an object is that it is a mammal, the code for *mammal*, $[animal, mammal|X]\backslash X$, can be extended as more information is discovered.

This technique permits us to formulate intensional replies, to perform quick semantic agreement verifications on natural language queries and to achieve partial execution of some queries. For example, we can state that all reptiles crawl: $crawl(A \in [animal, reptile|X]\backslash X)$. Now we can ask which animals crawl (e.g. `?- animal(A), crawl(A).`). This will quickly reply with *reptile*. If we desire further information, we can backtrack to find more specific elements in our hierarchy which crawl.

This approach has the advantage of being simple, efficient and entirely within the framework of Prolog terms. However, limiting taxonomies to being trees imposes a severe restriction on the types of inheritance and operations that can be performed.

[1] A difference list is a list representation that allows for appends to execute in one unification step. To achieve this, a list is viewed as the difference between two other lists. For example, the list $[1,2,3]$ can be viewed as the difference between $[1,2,3,4,5]$ and $[4,5]$. By using a variable as the second list (e.g. representing $[1,2,3]$ as $[1,2,3|X]\backslash X$), we can append any list to it simply by giving a value to X through unification.

3.3 Extending trees to graphs

Extending the above method to deal with general partial orders, Massicotte [96] partitions the nodes into two sets: nodes with a unique path from the root (*deterministic* nodes) and nodes with multiple paths from the root (*non-deterministic* nodes). Non-deterministic nodes are a result of one or more ancestors having multiple inheritance.

In essence, the maximal tree portion of the hierarchy (the *tree prefix*), starting at the root, is treated in the same way as above. Thus, a deterministic node is represented by a path, expressed as a difference list, from the root to the node. For a nondeterministic node, the paths from the closest ancestors with multiple inheritance are explicitly represented, and the paths from the root to these ancestors are implicitly represented (through a predicate call associated with each such path). If a node has multiple parents, then multiple paths are associated with it, one from each closest ancestor with multiple inheritance, or from the root if no such ancestors exist.

To demonstrate, Figure 3.2 shows a hierarchy in which we have emphasized the tree prefix. The deterministic nodes are $\{\top, person, adult, child, butterfly, larva\}$ and the non-deterministic nodes are $\{teenager, caterpillar, \bot\}$. To represent *adult* requires only storing the path $[\top, person, adult]$, but to store *teenager* requires the paths $[adult, teenager]$ and $[child, teenager]$. To find all paths from $\top$ to *teenager* requires appending the path $[adult, teenager]$ to each path from $\top$ to *adult* and appending $[child, teenager]$ to each path from $\top$ to *child*. This can be achieved via unification; the recursive nature of the implicit paths ensures that all paths will be found.

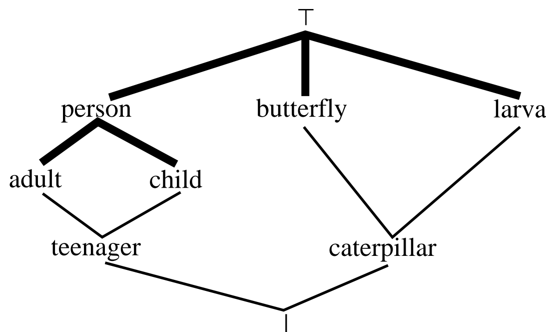

Figure 3.2: Taxonomy showing tree prefix

To test whether a label, e_1, subsumes another label, e_2, now requires checking if there exists a path from the root to e_1 which is a prefix of some path from the root to e_2. If both e_1 and e_2 are deterministic nodes, then this operation can be achieved in one unification. If either one is a non-deterministic node, this will require one unification for each possibility in the worst case. Provided the taxonomy is a join semi-lattice, joins may also be formulated in a recursive manner. There is, however, no simple way to use this approach for meets, or for finding join crests in non-lattices.

This approach enjoys the simplicity of Dahl's encoding, and it also remains within the scope of Prolog. However, it cannot tolerate many multiple inheritances before its recursive nature will limit its efficiency.

3.4 Characterizing term encodable hierarchies

The technique of using unification to perform hierarchical operations can be generalized to use logical terms as codes, rather than difference lists. We first note that the approach of [34] for encoding tree-shaped hierarchies, can also be achieved by representing the partial paths as nested, unary function symbols (as pointed out in [101]). So, for example, the taxonomy in Figure 3.1 can be represented using terms as shown in Figure 3.3. Checking subsumption still requires one unification. If the unification succeeds, then the term that was further instantiated subsumes the term that was not. If the unification fails, then the two elements are incompatible. Joins can be achieved through *anti-unification*, the dual of unification. For example, to compute the join $dog \sqcup cat$, we anti-unify the terms $animal(mammal(dog(_)))$ and $animal(mammal(cat(_)))$, resulting in $animal(mammal(_))$ which is the term associated with *mammal*.

With this scheme, it is possible to utilize functions with more than one argument. The technique in [21] is direct extension of [33] that allows a set of tree shaped hierarchies, leading to multi-argument terms where a subterm has

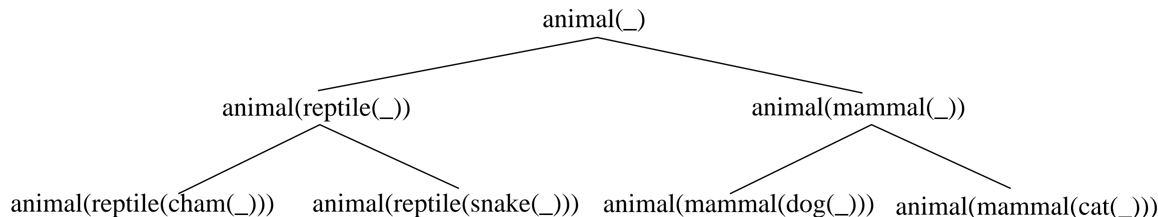

Figure 3.3: Logical term encoding of a tree-shaped hierarchy

one argument per tree rooted at that node. This can be taken even further to encode more general taxonomies, by permitting logical variables. As an example, consider the term encoding shown in Figure 3.4 of our example hierarchy from Figure 3.2.

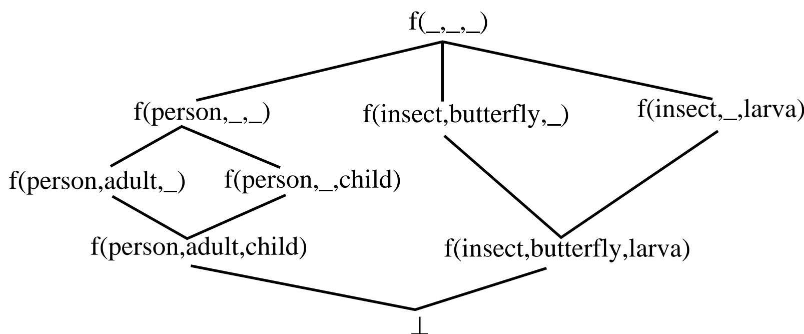

Figure 3.4: Encoding of type hierarchy in Figure 3.2

Mellish (in [102]), provides a characterization of lower semi-lattice taxonomies (i.e. unique meets exist) for which a particular type of term encoding exists. Such encodings are targeted at determining meets and checking subsumption. Essentially, a term encoding, in Mellish's sense, requires that the meet of two elements can be determined by unifying the terms associated with these elements. If the unification fails, then the result is bottom. Otherwise, the resulting term is exactly the term associated with the unique meet element. This is defined more formally as follows:

Definition 3.1 *A hierarchy* $H = (\Sigma, \leq)$ *is* term encodable *iff, for some term space* G*, there is a mapping* $\tau : \Sigma \rightarrow G$ *satisfying:*

1. *If* $\tau(e_1) = \tau(e_2)$ *then* $e_1 = e_2$
2. $\tau(\bot) = \bot$
3. $\tau(e_1 \sqcap e_2) = \tau(e_1) \sqcap \tau(e_2)$

where e_1 *and* e_2 *are elements of* Σ*, and* $\sqcap$ *represents the term unification operation.*

The first condition ensures that the mapping is invertible, which is necessary for decoding if we are to support meets. The third condition requires that τ not only preserves subsumption, but also that the unification of the terms of two elements is exactly the term of the meet of those elements. The second condition guarantees that if this meet is $\bot$, the unification fails. Therefore, if we can find a term encoding for our taxonomy, meets can be determined using one unification step.

Although no algorithm for constructing term encodings is given, Mellish does categorize taxonomies according to the complexity of the types of terms required for such encodings. The simplest encodings require only *tree* terms (i.e. terms in which all variables are singletons). Such terms can always be drawn as trees. At the next level, *flat* terms are studied (i.e. terms in which variables may corefer, but the depth is restricted to one). Flat terms can then be generalized to the set of all terms. Going beyond terms leads us to the use of *rational trees* in encodings [30].

Unfortunately, determining which type of terms are required for encoding a given taxonomy appears to be difficult. Also, constructing encodings that employ terms more complex than simple tree terms may be non-trivial, and limits the possibility of exploiting parallelism in unification. Even some simple taxonomies turn out to be non-tree term encodable, according to the above definition of encodability. We provide examples of this in the next chapter.

Furthermore, a change to the taxonomy may require recomputation of the entire, or a significant portion of, the encoding.

In [104], Mellish extends his characterization to taxonomies encodable by graphs.

3.5 Bit-vector encodings

A number of researchers have explored the possibility of encoding taxonomies using bit-vectors, using the operations of logical (bit-wise) AND and OR to compute meets and joins. The founding research on using bit-vectors was by Aït-Kaci *et al.* [2] for use in the logic programming language LIFE [4]. The definition of encoding used assumes that the taxonomy is a lower semi-lattice. In order to achieve this, a minimal semi-lattice completion is presented. It is important to note that this semi-lattice construction is not actually computed, but rather is used to provide a semantics for computing meets that are not unique. This contrasts with the approaches by Mellish, above, and Caseau, below, which actually require the taxonomy to be a lower semi-lattice. Performing this construction may be exponential in the worst-case.

Transitive closure. A simple bit-vector encoding, called *transitive closure*, can be achieved by associating one position in the bit-string with each element in a taxonomy (except $\bot$). Let us call $element(i)$ the element associated with position i in this bit-vector. For each element e, position i is a 1 if e subsumes $element(i)$ and a 0 otherwise. Thus, each code for an element incorporates all of the lower bounds of that element. To demonstrate, consider the taxonomy of Figure 3.2. Table 3.1 associates one bit with each element, and Figure 3.5 shows the transitive closure of the table according to subsumption (in a bottom-up manner).

Table 3.1: Assigning bits to elements from Figure 3.2

	$\top$	person	butterfly	larva	adult	child	teenager	caterpillar
$\top$	1	0	0	0	0	0	0	0
person	0	1	0	0	0	0	0	0
butterfly	0	0	1	0	0	0	0	0
larva	0	0	0	1	0	0	0	0
adult	0	0	0	0	1	0	0	0
child	0	0	0	0	0	1	0	0
teenager	0	0	0	0	0	0	1	0
caterpillar	0	0	0	0	0	0	0	1

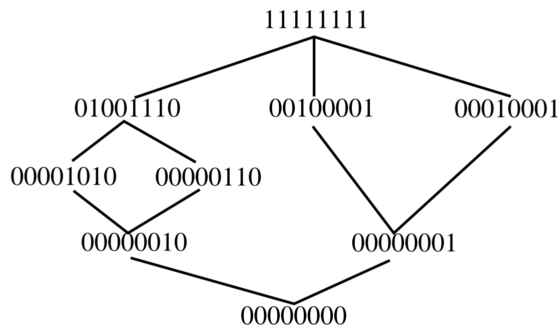

Figure 3.5: Bottom-up bit-vector encoding of taxonomy in Figure 3.2

Both subsumption checking and meet operations can be performed using logical AND operations. That is, $e_1 \leq e_2$ if and only if $\tau(e_1)$ AND $\tau(e_2) = \tau(e_1)$. Also, $e1 \sqcap e2$ is computed by $\tau(e_1)$ AND $\tau(e_2)$. If the meet is unique, this will be the code of that element. If not, this code will represent the crown and additional decoding must be done to extract the elements comprising this crown.

Compact encoding. The above approach requires one bit for every element except $\bot$. Thus, a taxonomy with n elements requires $n-1$ bits per code. By analyzing the structure of the taxonomy, it is possible to reduce this number. When an element has exactly one child, we must use an additional bit to distinguish its code from that of its child. But when an element has multiple children, it may be possible to encode it simply using the OR of the codes of its children. The compact encoding scheme optimistically assigns codes in such a way, and if this leads to two incomparable elements having comparable codes, then additional bits are added. Thus, while transitive closure indiscriminately uses one bit per element, compact encoding adds bits only as necessary, saving space on elements that do not require a bit to maintain the encoding homomorphism. Subsumption checking and meets are computed using logical AND, as before.

Consider our example taxonomy. We start with 0 for $\bot$. Then we assign 1 to *teenager* and 10 to *caterpillar*. Next *adult* is allotted 101 and *child* 1001. *Butterfly* is given 10010 and *larva* 100010. Then *person*, since it has two children is assigned 101 AND 1001 = 1101. Finally $\top$, with three children, gets 1101 AND 10010 AND 100010 = 111111. In this simple example, we reduce the code size from 8 bits to 6 bits. This compact encoding is shown in Figure 3.6.

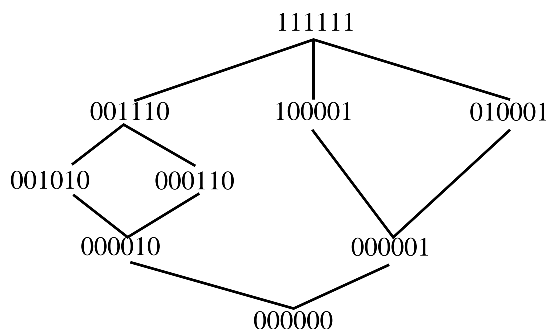

Figure 3.6: Compact bit-vector encoding of taxonomy in Figure 3.2

Which elements require a bit? For a bottom-up compact encoding, it is precisely the join irreducible elements. If this scheme was applied in a top-down manner, it would be the meet irreducible elements. Therefore, unlike the transitive closure approach, a compact encoding may require a different number of bits depending on whether it is applied in a top-down or a bottom-up fashion.

Modulation Many objects naturally group themselves into relatively disjoint, dense groups with few links between groups. This can be exploited by treating these groups, or *modules*, as a single unit in the taxonomy [2]. Then the modified taxonomy (with one module node replacing all the elements of the module) can be encoded separately from the elements in the module. To do this, the module must itself have the form of a taxonomy. That is, modules have a top and a bottom element, and every path from outside to lower elements inside the module goes through the top node of the module, and every path from inside to lower elements outside the module goes through the bottom node of the module.

Since modules are sub-taxonomies, this process can continue recursively, until each module contains a small number of elements. The difficulty lies in finding modules. The heuristic algorithm provided in [2] attempts to modulate a given taxonomy, but is not guaranteed to find a maximal modulation. A fast (linear) algorithm for modulation has recently been developed [76].

An element may now reside within a module, which is itself within a module and so on. In [2], the code of such an element is the juxtaposition of the codes of the containing modules (starting with the maximal containing module) and the code of the element, which was calculated in the least containing module.

The operations of subsumption checking and meet are complicated by modulation and will be described only for one level of modulation. To check if element e_1 subsumes element e_2, we must first check which modules they are in. If they reside in the same module, we simply check if the code for e_1 subsumes the code for e_2, as

before. If they are in different modules, we check if the code for the module containing e_1 subsumes the code for the module containing e_2. Otherwise e_1 does not subsume e_2.

To determine the meet of e_1 and e_2 involves a similar process. If they are in the same module, then simply take the AND of their codes. If e_1 subsumes e_2, then the meet is e_2. If e_2 subsumes e_1, then the meet is e_1. Otherwise, take the logical AND of the containing module codes to obtain the meet module and the meet element is the topmost element of this module. For non-unique meets, crowns are found, as in the compact encoding method above.

To illustrate, we add an *insect* element above *butterfly* and *larva* in our example taxonomy. Now, the portion of the hierarchy dealing with people can form one module, and the portion dealing with insects can form another. These modules can then be encoded using the compact encoding. This modified taxonomy and its modulated encoding are shown in Figure 3.7, where the module codes have been separated from the element codes by a colon.

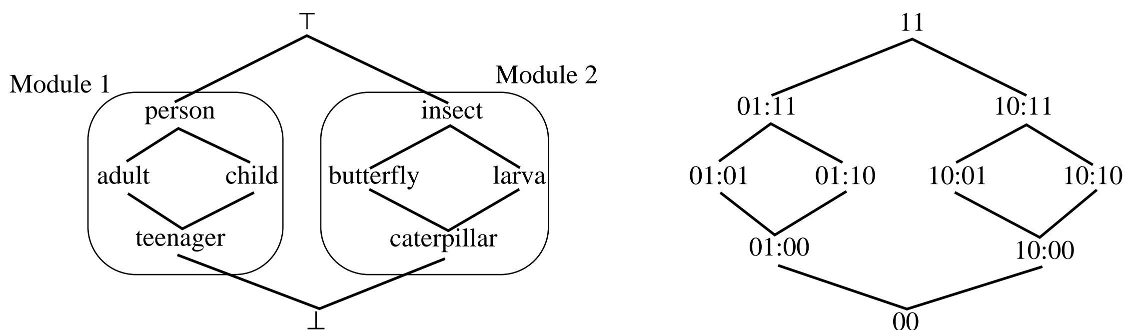

Figure 3.7: A modulated taxonomy and its encoding

To find the meet $adult \sqcap child$, we AND the element codes 01 AND 10, and prepend the module code 01 to get 01:00, which is the code of *teenager*. To find $adult \sqcap butterfly$, we AND the module codes 01 AND 10 to get 00, which is the module code of $\bot$.

These operations can be extended in an obvious way for further levels of modulation. Since each level of modulation adds one more step in the process and since there can be at most $log N$ levels of modulation for a taxonomy of N elements, these operations take at most $log N$ steps. So, although modulation has the potential to reduce the size of the codes substantially, it also increases the complexity of computing operations. The assumption is that most operations will be within, not between, modules, so that only one step is required.

Also, the complexity of determining a modulated encoding is substantial. Modifications to the taxonomy can be either more or less costly than for non-modulated taxonomies. Changes within a module restrict the extent of changes to within that module. If, however, one or more modules are breached (e.g. a link is added that enters or leaves a module at a mid-point), then we may have to re-modulate a significant portion of the hierarchy.

In Chapter 5, we formally deal with and extend modulation.

Encoding for subsumption only If the only operation required is subsumption checking, then it may be possible to reduce the length of codes further, without resorting to modulation. In this situation no decoding is necessary and the codes can be such that neither meets nor joins can be determined, as long as the subsumption relation is maintained.

One such approach has been developed for the Laure object-oriented programming language [24]. This scheme modifies a top-down version of compact encoding, but is restricted to taxonomies that are lattices. The algorithm basically assigns a bit position, or *gene*, to each meet irreducible element. Since the taxonomy is a lattice, these are the elements with a unique parent. The code for an element is the union of the genes (i.e. logical OR) of its ancestors, plus its gene, if it is meet irreducible. Since we are not concerned with computing meets or joins, it is possible to assign the same gene for some elements, provided this doesn't violate the

subsumption relation. Caseau's algorithm performs this incrementally, in a top-down manner. As each meet irreducible element is processed, an attempt is made to assign a gene already in use. For other elements, a check is made to see if the union of the parent genes violates subsumption. If so, mutations of ancestral genes are performed until subsumption is respected.

Using this algorithm, we encode the taxonomy in Figure 3.7 as shown in Figure 3.8. In the taxonomy at the left, we display the genes assigned to each meet irreducible element. As can be seen the *adult* and *butterfly* elements share a gene, as do *child* and *larva*. This reduces the code size to 4 bits, as achieved by modulation. Checking for subsumption requires only one logical AND operation: element e_1 subsumes e_1 if and only if $\tau(e_1) AND \tau(e_2) = \tau(e_1)$. We cannot, however, compute meets or joins due to the polymorphic character of genes.

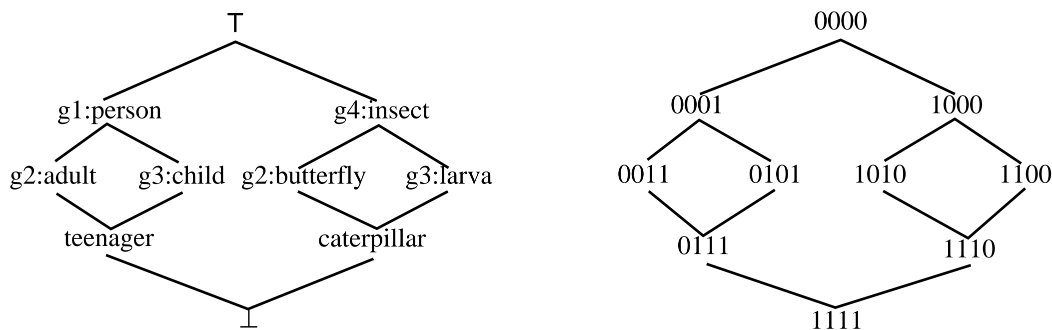

Figure 3.8: A subsumption only encoding

3.6 Discussion

In this chapter, we have attempted to describe the evolution of taxonomic encoding in a general and intuitive manner. Where possible, we described techniques from the viewpoint of the original research. Some of the techniques covered here, and additional techniques, are described in the following chapter, where the emphasis is on characterizing techniques using our formal framework.

Chapter 4

The Foundations of Taxonomic Encoding

"Everything is simpler than you think and at the same time more complex than you imagine"

– Goethe

Most of the research on encoding has focused on algorithmic and implementational details of encoding, and has largely ignored or left unstated the informational content of the technique. In this chapter, we explore a fundamental structure underlying encoding. By characterizing encoding using *spanning sets* we are able to provide a concise framework in which all schemes can be compared, regardless of the actual implementation. This analysis permits a separation of the informational content of an encoding scheme from the implementational details, and allows us to see how both of these aspects affect time and space requirements. This exploration expands and formalizes our introduction of spanning sets for encoding that appeared in a short workshop paper [48].

In addition to the theoretical appeal of our framework, we also develop several important results. We show a correspondence among several existing encoding techniques (sections 4.5 and 4.6). We prove two NP-Hardness results, which demonstrate limitations to encoding algorithms and reveal avenues for approximation algorithms (sections 4.7 and 4.8). Our abstraction also exposes a more comprehensive view of some existing techniques, indicating directions for further research. We discuss in more detail in section 4.10 our contributions to taxonomic encoding as well as specific directions for future research.

In the following section, we motivate and define taxonomic encoding. We rely heavily on the lattice theory concepts introduced in section 2.1, including our departures from standard theory. In section 4.2 we characterize encoding as order-embedding mappings induced by spanning sets. Since the result of these mappings is a set, taxonomic operations reduce to set operations, independent of the implementation. Section 4.3 introduces a variety of implementations of order subsets, specifically for the implementation of spanning sets and section 4.4 describes how we can permit portions of a taxonomy to be infinite while still benefiting from encoding techniques.

Using this framework, we analyze the information content of various spanning set types and develop formal techniques to reduce the representation cost of the spanning set mapping. Through much of this analysis, we introduce existing encoding techniques, characterize them in terms of our spanning set framework, and then abstract general properties and limitations of such spanning sets. We first characterize some simple encodings in terms of spanning sets of *principal down-sets* in section 4.5. This includes the transitive closure and compact encodings of [2]. We then show a correspondence between principal down-sets and *prime up-sets*, providing a direct link to the approach of [77]. Section 4.7 explores and characterizes spanning sets that preserve only subsumption, and we prove that determining a minimal such spanning set is NP-Hard. The approach of [24] is shown to be an approximation of the optimum. We next consider how decomposing a spanning set can achieve more concise results, as in the proposals of [97] and [102]. We also prove that, for certain forms of decomposition, finding the optimal is NP-Hard. Section 4.9 views partial orders as systems of constraints, and encodings as preserving certain properties by representing a subset of these constraints. Using coreference, more expressive encodings are possible. Finally, we discuss areas for future

research, including expanding the theory presented, exploring implementational issues and designing approximation algorithms.

4.1 Setting the Stage

The general problem we wish to address is as follows: given an ordered set P, how do we represent P to provide fast computation of subsumption, and possibly meets and/or joins? We focus on encoding finite ordered sets, although we later describe how these can be augmented with certain forms of infinite orders. Some ordered sets, such as families of subsets ordered by set inclusion, sets of integers ordered by divisibility (i.e. $x \leq y$ if and only if x is a factor of y), and logical term spaces ordered by term instantiation, have in common the simplicity of element comparisons: determining if $x \leq y$ can be done locally (i.e. using only information directly related to x and y) and efficiently. This is not true, however, of many others, such as sets of graphs ordered by subgraph isomorphism and multiple inheritance hierarchies in object-oriented systems. In the former case, local information can be used to check subsumption, but this is costly. In the latter case, only the intransitive, irreflexive portion of the partial order is maintained (i.e. the *transitive reduction*), so there is no local information to determine if $x \leq y$. It is in contexts such as these that encoding is beneficial.

We will assume that we are given an ordered set P as a graph $G = (P, E)$, where E is either the *transitive closure* (i.e. $(x, y) \in E$ if and only if $x \leq y$) or the transitive reduction of P. We need a way to implement P that is both space efficient and facilitates fast computation of operations. Directly implementing P using standard graph representation techniques is straightforward (where $G = (P, E)$); two common techniques are *adjacency matrices*, which take $O(|P|^2)$ space, and *adjacency lists*, which take $O(|E| log|P| + |P|)$ space. If G is the transitive reduction graph of P, then adjacency list representation corresponds to maintaining the list of parents (or children) for each element. Subsumption, meets and joins can be determined in $O(|E|)$ time for either implementation. If G is the transitive closure graph of P, then subsumption can be computed in constant time for adjacency matrices, and $O(|P|)$ time for adjacency lists. In both cases, meets and joins take $O(|P|)$ time.

Before defining encoding, we recall our generalizations of meet and join: for a subset Q of an ordered set P, we call the set of minimal upper bounds of Q the *join base* and the maximal lower bounds of Q the *meet crest*[1]. A join (meet) is simply a singleton join base (meet crest). We use the same notation for joins and join bases (and meets and meet crests). Thus, in Figure 2.2, $fox \sqcup wolf = \{canine, wild\}$ and $wild \sqcap social = \{wolf, african\ wild\ dog\}$, whereas $dog \sqcap wild = \{feral\ dog\}$.

Definition 4.1 *Let P and Q be ordered sets, and τ an order mapping $\tau : P \to Q$. Then τ is*

- *a* (subsumption) encoding *for P if τ is an order-embedding (i.e. $x \leq_P y$ if and only if $\tau(x) \leq_Q \tau(y)$).*
- *a* meet encoding *for P if τ is* meet-crest-preserving*: if $a, b \in P$ then $a \sqcap_P b = \tau^{-1}(\tau(a) \sqcap_Q \tau(b))$, where τ^{-1} is the inverse of τ*[2]*.*
- *a* join encoding *for P if τ is* join-base-preserving*: if $a, b \in P$ then $a \sqcup_P b = \tau^{-1}(\tau(a) \sqcup_Q \tau(b))$.*

Although $\leq_Q$ defines a partial order on Q, determining if $x \leq_Q y$ may be accomplished in a number of ways, as we discuss in section 4.3. The intent of an encoding is that taxonomic operations in Q can be performed more efficiently than in P. There are several forms of encoding that have appeared in the literature; the trademark of encoding is the pre-computation of the encoding function τ and the association with each element $x \in P$ the value, or *code*, $\tau(x)$. Thus encoding trades the cost of explicitly storing τ for improved time to compute taxonomic operations.

In most schemes, the target space Q has the property that elements are *independent*. That is, the order relation is somehow *encoded* in the elements themselves. Examples of this include bit-vectors and logical terms. In the

[1] The set of upper bounds (lower bounds) is an up-set (down-set). The join base (meet crest) is precisely the set of factors for this set - its base (crest). Join bases and meet crests are anti-chains.

[2] In general, $\tau(a) \sqcap_Q \tau(b)$ is a set of elements in Q, so τ^{-1} must map this set back to the meet crest in P. Depending on the structure of Q, however, τ^{-1} is normally treated in one of two ways: (i) If Q is a lattice, then $\tau(a) \sqcap_Q \tau(b)$ reduces to a single element of Q. In this case, Q embeds a minimal completion of P, and the inverse τ^{-1} must map back to the meet crest in P; (ii) If τ is an order isomorphism (i.e. it maps P *onto* Q), then $\tau(a) \sqcap_Q \tau(b)$ reduces to the set of elements in Q corresponding to the meet crest in P. Here, τ^{-1} must map each element in this set back to P. Note that if P and Q are both lattices, then τ must be meet-preserving in the lattice-theoretic sense.

tree encoding scheme of [78], however, Q is a tree data structure, and τ maps elements of P to nodes of the tree. Operations in P are translated to operations on this data structure.

In this chapter, it is our goal to develop a unified framework that separates the content (semantics) of the encoding map from its implementation (syntax). We do this using a structure called a *spanning set*, which we introduce in section 4.2. Through this separation we provide a common ground on which different encoding schemes can be compared, analyze the effect on time and space of different implementations, and study the semantic content that encodings *must* possess in order to preserve certain properties of an ordered set. We also strive to provide a principled basis on which to select or design encoding algorithms for particular taxonomic applications, and to expose some of the limitations and restrictions to encoding.

There are several aspects by which we can characterize encoding algorithms:

- The taxonomic operations supported.
- The time and space complexity of the encoding algorithm.
- The space requirements of resulting encodings.
- The time complexity of performing operations using resulting encodings.
- The complexity of modifying an encoding.
- The complexity of *decoding* (i.e. computing τ^{-1}).

We show how various encoding techniques and implementations affect these characteristics. Since the focus and requirements of particular taxonomic applications may differ, it is apparent that there may be no *best* encoding algorithm to satisfy all needs. Rather, the designer of an encoding algorithm must take into account the needs of the application, and the form of the taxonomies to encode, in order to determine the relative importance of the above characteristics. Using our framework, appropriate techniques and implementations can be selected, leading to existing algorithms, or the need to design new algorithms.

Our framework would be improved with empirical results that demonstrate the behaviour of various encoding algorithms with respect to the above characteristics. In order to be useful, however, such testing would have to be extensive and this is beyond the scope of this thesis. Our research, however, provides an organizational basis with which such testing could be carried out. Some empirical results on the space efficiency of different encoding algorithms is available in [43].

4.2 Spanning Sets

Now we present spanning sets as a basis for encoding, generalized from [102].

Definition 4.2 *Let P be an ordered set. A family of subsets S of P is called a* spanning set *if the function $\mathcal{C} : P \rightarrow 2^S$ defined by $\mathcal{C}(x) = \{s \in S | x \in s\}$ is one-to-one.*

A spanning set S is ordered under set inclusion (where, for $s_1, s_2 \in S$, $s_1 \leq_S s_2$ if and only if $s_1 \subseteq s_2$), and the function $\mathcal{C}$ is an order mapping, called the *component mapping* (where elements of S can be regarded as components from which P is constructed). In the next subsection we describe some structural restrictions that enable us to use spanning sets to perform taxonomic operations locally. Encoding can then be viewed as computing a spanning set that preserves the desired properties of an order P, and then efficiently representing the component mapping. As an example, the figure below shows a simple lattice and two spanning sets: $S_1 = \{s_1 = \{a, \bot\}, s_2 = \{b, \bot\}, s_3 = \{c, \bot\}\}$, and $S_2 = \{s_1 = \{a, b, c\}, s_2 = \{b, \bot\}, s_3 = \{\top, b, c\}\}$. It can easily be verified that component mappings for both of these are one-to-one. For S_1, we have $\mathcal{C}(a) = \{s_1\}$, $\mathcal{C}(\top) = \emptyset$ and $\mathcal{C}(\bot) = \{s_1, s_2, s_3\}$.

In [12], a variation of spanning sets was studied to produce a number of fundamental duality results. It is also worth noting the similarity between spanning sets and *reduced* or *minimal bases* in Wille's concept lattices [155], where lattice elements and spanning set components correspond to objects and attributes, respectively, in Wille's terminology.

We are primarily concerned with spanning sets of down-sets (and up-sets), where $S \subseteq \mathcal{O}(P)$ and $\mathcal{C} : P \rightarrow 2^{\mathcal{O}(P)}$. What makes these interesting components is that they encapsulate much of the order information. In Chapter 5, we introduce the concept of a spanning set of order intervals.

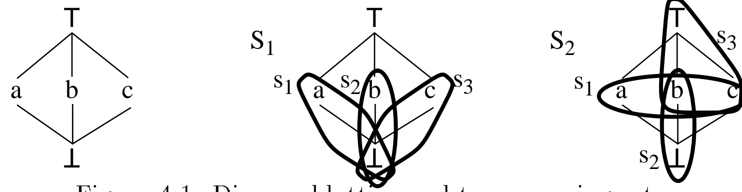

Figure 4.1: Diamond lattice and two spanning sets

We hypothesize that all encodings can be characterized as computing a spanning set of down-sets, up-sets or intervals, possibly augmented with constraints, and implementing its associated component mapping. Rather than trying to establish this claim, we portray all the encodings we are aware of by using spanning sets. These portrayals are supported by a number of formal results. We later discuss augmenting spanning sets with constraints (such as coreference constraints as provided by logical variables) (section 4.9) and spanning sets of intervals (Chapter 5). Viewing encoding in terms of spanning sets allows us to separate the implementation details of any particular encoding algorithm from the structural properties of the spanning set being constructed. The spanning set embodies the content (semantics) of an encoding and the implementation embodies the form (syntax).

4.2.1 Taxonomic operations using spanning sets

We now demonstrate how spanning sets that satisfy certain restrictions reduce taxonomic operations to set operations.

Definition 4.3 *A spanning set S on an ordered set P* preserves subsumption *if either (i) for all $a, b \in P$, $a \leq b$ if and only if $\mathcal{C}(a) \subseteq \mathcal{C}(b)$, or (ii) for all $a, b \in P$, $a \leq b$ if and only if $\mathcal{C}(a) \supseteq \mathcal{C}(b)$.*

Equivalently, this requires the component mapping to be an order-embedding. Although order-preserving mappings maintain comparability, we need to also preserve incomparability. We say that subsumption is preserved with subsets in case (i) (i.e. a is subsumed by b if and only if $\mathcal{C}(a)$ is a subset of $\mathcal{C}(b)$) and with supersets in case (ii). If S is a spanning set of down-sets, then the component mapping is monotonically increasing as we descend the order (since if $x \in \downarrow Q$ then any descendant of x is also in $\downarrow Q$). In this case, subsumption may only be preserved with supersets. Conversely, if S preserves subsumption with supersets, then S must be a spanning set of down-sets. Thus, not all spanning sets preserve subsumption. In the above example, S_1 preserves subsumption (with supersets) but not S_2, since $\mathcal{C}(a) = \{s_1\} \subseteq \{s_1, s_3\} = \mathcal{C}(c)$ yet $a||c$.

Definition 4.4 *A spanning set S on a lattice L* preserves meets *if either (i) for all $a, b \in L$, $\mathcal{C}(a \sqcap b) = \mathcal{C}(a) \cap \mathcal{C}(b)$, or (ii) for all $a, b \in L$, $\mathcal{C}(a \sqcap b) = \mathcal{C}(a) \cup \mathcal{C}(b)$. S* preserves joins *if either (i) for all $a, b \in L$, $\mathcal{C}(a \sqcup b) = \mathcal{C}(a) \cap \mathcal{C}(b)$, or (ii) for all $a, b \in L$, $\mathcal{C}(a \sqcup b) = \mathcal{C}(a) \cup \mathcal{C}(b)$*[3]*.*

If a spanning set preserves meets or joins, then it preserves subsumption, because $a \leq b$ if and only if $a \sqcap b = a$ and $a \sqcup b = b$. Also, a spanning set of down-sets can preserve joins only with intersection and meets only with union. In general, if a spanning set S preserves subsumption with supersets (i.e. S is a spanning set of down-sets) then $\mathcal{C}(a) \cup \mathcal{C}(b) \subseteq \mathcal{C}(a \sqcap b)$ and $\mathcal{C}(a \sqcup b) \subseteq \mathcal{C}(a) \cap \mathcal{C}(b)$. Unfortunately, it is not always possible for a spanning set to preserve both meets and joins (unless the ordered set is distributive[4], as discussed in section 4.2.2). Consider again the non-distributive ordered set in Figure 4.1. The spanning set $\{\downarrow a, \downarrow b, \downarrow c, \downarrow\{a, c\}\}$ preserves subsumption, but not joins or meets, since $a \sqcap c = \bot$, but $\mathcal{C}(a) \cup \mathcal{C}(c) = \{\downarrow a, \downarrow c, \downarrow\{a, c\}\} \neq \{\downarrow a, \downarrow b, \downarrow c, \downarrow\{a, c\}\} = \mathcal{C}(\bot)$. Also, $a \sqcup c = \top$, but $\mathcal{C}(a) \cap \mathcal{C}(c) = \{\downarrow\{a, c\}\} \neq \emptyset = \mathcal{C}(\top)$. The spanning set $\{\downarrow a, \downarrow b, \downarrow c\}$ preserves joins with intersection but not meets, while $\{\downarrow\{a, b\}, \downarrow\{b, c\}, \downarrow\{b, c\}\}$ preserves meets with union but not joins. Suppose we have a spanning set S that preserves joins with intersection. Since the join of any pair of a, b, c is $\top$, the intersection of any pair of their component mappings must be $\mathcal{C}(\top)$. Further, each must be in at least one component different from the others. But then the union of any pair cannot possibly be $\mathcal{C}(\bot)$.

[3] To generalize this definition to an ordered set P, we say *S preserves meet crests* if either (i) for all $a, b \in P$, $a \sqcap b = \mathcal{C}^{-1}(\mathcal{C}(a) \cap \mathcal{C}(b))$, or (ii) for all $a, b \in L$, $a \sqcap b = \mathcal{C}^{-1}(\mathcal{C}(a) \cup \mathcal{C}(b))$.

[4] A lattice L is *distributive* if $\forall a, b, c \in L$, $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

Theorem 4.1 Spanning Set Duality Theorem. *Let L be a lattice and S a spanning set of down-sets for L. Let $\overline{S}$ be the set of up-sets defined as $\overline{S} = \{L\backslash{\downarrow}Q \mid {\downarrow}Q \in S\}$. Then (i) S preserves subsumption with supersets if and only if $\overline{S}$ preserves subsumption with subsets and (ii) S preserves joins with intersection if and only if $\overline{S}$ preserves joins with union.*

Proof: Consider the component mapping for $\overline{S}$: $\overline{\mathcal{C}}(x) = \{L\backslash{\downarrow}Q \in \overline{S} | x \in L\backslash{\downarrow}Q\}$. But $x \in L\backslash{\downarrow}Q$ if and only if $x \notin {\downarrow}Q$, so $\overline{\mathcal{C}}$ is isomorphic to the converse mapping of $\mathcal{C}$: $\mathcal{C}^c(x) = \{{\downarrow}Q \in S | x \notin {\downarrow}Q\}$.

(i) Suppose S preserves subsumption with supersets. Consider any two elements, $a, b \in L$. The converse mapping maps these elements as follows: $\mathcal{C}^c(a) = S\backslash\mathcal{C}(a)$ and $\mathcal{C}^c(b) = S\backslash\mathcal{C}(b)$. If $a \leq b$ then $\mathcal{C}(a) \supseteq \mathcal{C}(b)$, so clearly $\mathcal{C}^c(a) \subseteq \mathcal{C}^c(b)$. If $a \not\leq b$ then $\mathcal{C}(a) \not\supseteq \mathcal{C}(b)$, and so $\mathcal{C}^c(a) \not\subseteq \mathcal{C}^c(b)$. The case when $\overline{S}$ preserves subsumption with subsets is similarly proved.

(ii) Consider the join of any two elements $a, b \in L$. If S preserves joins with intersection then $\mathcal{C}(a) \cap \mathcal{C}(b) = \mathcal{C}(a \sqcup b)$. The converse mapping maps these as: $\mathcal{C}^c(a) = S\backslash\mathcal{C}(a)$, $\mathcal{C}^c(b) = S\backslash\mathcal{C}(b)$ and $\mathcal{C}^c(a \sqcup b) = S\backslash(\mathcal{C}(a) \cap \mathcal{C}(b)) = S\backslash\mathcal{C}(a) \cup S\backslash\mathcal{C}(b) = \mathcal{C}^c(a) \cup \mathcal{C}^c(b)$. Now, if $\overline{S}$ preserves joins with union then $\mathcal{C}^c(a) \cup \mathcal{C}^c(b) = \mathcal{C}^c(a \sqcup b)$. The component mapping for S maps these as: $\mathcal{C}(a) = S\backslash\mathcal{C}^c(a)$, $\mathcal{C}(b) = S\backslash\mathcal{C}^c(b)$ and $\mathcal{C}(a \sqcup b) = S\backslash(\mathcal{C}^c(a) \cup \mathcal{C}^c(b)) = S\backslash\mathcal{C}^c(a) \cap S\backslash\mathcal{C}^c(b) = \mathcal{C}(a) \cap \mathcal{C}(b)$. □

This theorem demonstrates that for every spanning set of down-sets that preserves joins with intersection, there is a spanning set of up-sets that preserves joins with union. Since this construction is invertible, the converse is also true. Together with the dual, this shows we can characterize all spanning sets that preserve joins or meets with intersection or union by analyzing only those that preserve joins with intersection.

We require an efficient means to evaluate the component mapping $\mathcal{C}$. A key feature of encoding is that $\mathcal{C}$ is calculated *a priori*, or incrementally, and stored in a form amenable to efficient computation. This amounts to associating with each element x of the taxonomy the set representing $\mathcal{C}(x)$, as we describe in section 4.3.

4.2.2 Representation theory and encoding

Representation theory attempts to identify a small suborder Q of a lattice L from which the entire lattice can be constructed easily and uniquely. In [38], it is shown that this can be done satisfactorily in the finite case for distributive lattices. In this case L is uniquely identified by its set of join (or meet) irreducible elements, where $Q = \mathcal{J}(L)$ and $L \cong \mathcal{O}(\mathcal{J}(L))$. The general case for lattices and partial orders is not so amenable to such an analysis.

Although encoding can benefit from the results of representation theory, there are a number of important differences. First, although we associate with an ordered set P a small set (i.e. the spanning set), we want a subset $S \subseteq 2^P$, not $Q \subseteq P$. Second, we are interested in representing P in order to facilitate efficient computation. To this end, we associate a code with each element of P. This contrasts with the above goal of uniquely representing P by the set Q. We do not want to reconstruct P, but rather we wish to associate with it a spanning set S from which codes can be formed.

There are, however, some results from representation theory that are fundamental to encoding, particularly the identification of join and meet irreducible elements as basic elements from which all other elements in an ordered set can be defined. This conclusion is also found in section 4.5, but doesn't require the ordered set to be a distributive lattice (as in Birkhoff's representation theorem [38]), so we can view spanning sets as partial representations of ordered sets (only preserving certain properties such as meets).

Since we are given an arbitrary ordered set P, we may not have the luxury to ensure that certain properties are satisfied (e.g. that P is a lattice or is distributive) - maintaining certain properties may entail adding an inordinate number of elements to P (e.g. the minimal lattice completion for a *standard example* S_n [144], which has $2n$ elements, contains 2^n elements [38]). If we can be sure that our set observes certain properties, or that the addition of a small (or bounded) number of elements can achieve these properties, then our encoding scheme can utilize this structure to generate more concise and/or flexible codes. For example, if we are guaranteed to have a distributive lattice, then we can specify spanning sets that preserve both meets and joins, although in general this is not possible [153]. In fact, every distributive lattice is isomorphic to a lattice of sets [38] (i.e. where meets and joins are computed by intersections and unions, respectively). This suggests a fundamental connection between representation theory and spanning sets. For a detailed analysis of properties of distributive and simplicial lattices related to encoding see [78]. In our presentation, we focus on the problem of encoding general partial orders and lattices and make no further

structural assumptions regarding the given ordered set, although our analysis should apply to techniques designed for more constrained orders.

4.3 Efficient Implementations of Component Mappings

In this section we describe some approaches to implementing subsets of ordered sets, particularly down-sets and up-sets, as returned by component mappings. This list is by no means exhaustive, but includes all the implementations that have been used for encoding. We are interested in implementing subsets within the order induced by a spanning set S, not in our original order P. This order is isomorphic to a suborder of P for spanning sets of principal down-sets, but not for more complicated spanning sets. Note that for any spanning set S, the subset $\mathcal{C}(x)$ is an up-set in S.

Given a spanning set S for an ordered set P, our goal is to represent, for each $x \in P$, the mapping $\mathcal{C}(x)$. In general $\mathcal{C}$ can be viewed as a relation: for $x \in P, s \in S$, $(x, s) \in \mathcal{C}$ if and only if $s \in \mathcal{C}(x)$. We may, however, be able to exploit the structure of the order induced by S.

4.3.1 Unordered implementations

By treating $\mathcal{C}(x)$ as an unordered subset of the domain S (i.e. by treating $\mathcal{C}$ as an unordered relation) we can realize implementations that do not utilize the hierarchical structure of the ordered set S. Such representations employ existing techniques for implementing sets. In the representations we describe below, the elements of S are given a linear order $\preceq$ (which is not necessarily a linear extension of S).

Characteristic vectors In a characteristic (or bit) vector implementation, we represent a subset $Q \subseteq S$ using a bit-vector of length $n = |S|$, essentially embedding S into the Boolean lattice of bit-vectors of length n. We place a 1 in position i if element i (in the chain $\preceq$) is a member of the subset and a 0 otherwise. This approach is analogous to *adjacency matrix* representations of graphs[5]. Set union and intersection are computed using bitwise OR and AND, respectively. For two subsets Q_1 and Q_2, $Q_1 \subseteq Q_2$ if and only if $Q_1 \cap Q_2 = Q_1$ (or $Q_1 \cup Q_2 = Q_2$). As an example, suppose $S = \{s_1, s_2, s_3, s_4, s_5\}$. We can represent the subsets $\{s_1, s_4\}$ and $\{s_2, s_3, s_4\}$ by the strings 10010 and 01110, respectively. The advantages of this representation include minimal storage requirements for each position (one bit) and immediate hardware implementation of set operations. Disadvantages include the need to store unfilled positions (i.e. every subset has length n), and more complicated processing required for large domains (asymptotically, the set operations grow linearly with the size of the domain).

Interval sets An alternative (proposed in [1]) is to represent a subset Q with a set of intervals, where each contiguous sequence of elements (in $\preceq$) is represented by an interval. For example, the above subsets would be represented as $\{[1, 1], [4, 4]\}$ and $\{[2, 4]\}$. Although this scheme alleviates the need to store unfilled positions, the set operations become more complex. Unlike the bit-vector approach, the order $\preceq$ may have a significant effect on the size of resulting codes. We discuss in section 4.5 how the approach in [1] finds optimal orderings.

Adjacency lists and hashing Analogous to adjacency list graph implementations, we can maintain for each element $x \in P$ the list of the elements $\mathcal{C}(x)$. This is space efficient for cases when $\mathcal{C}(x)$ is relatively small (i.e. the spanning set is large, but the component mapping only maps each element to a small number of elements), but becomes unwieldy as the size of $\mathcal{C}(x)$ increases. To speed up access to particular elements, we can hash $\mathcal{C}(x)$ for each $x \in P$ (i.e. for a given $x \in P, s \in S$, we can quickly determine if $s \in \mathcal{C}(x)$). Using this technique, there is no direct support for union and intersection operations.

4.3.2 Tree representations and code sharing

Using a linear ordering $\preceq$ of a spanning set S, we can implement the component mapping in a labeled tree form that permits some sharing of common subsets. We propose a generalization of the tree encodings in [77, 78, 114], which apply only to distributive lattices. In fact, this technique can be used to implement any family of finite subsets

[5] If $|P| = m$, then an adjacency matrix requires m^2 bits, whereas here we require $n * m$ bits.

from the same domain. The basic structure of such a tree representation is as follows. The elements in the original ordered set P are nodes in the tree (although there may be additional *empty* nodes, as discussed below). Each label is a subset of elements of S, and the union of all labels on the path from an element $x \in P$ to the root forms the set $\mathcal{C}(x)$.

There are several ways that we can build this tree. If our original ordered set is a distributive lattice L, then the approach of [78, 114] builds a very efficient tree for the spanning set $S = \{\downarrow x | x \in \mathcal{M}(L)\}$. Every node of the tree is an element of L and each label is a single element from S. Thus, the size of the tree is linear with respect to the size of L. Furthermore, the labels on all paths from a node to the root are monotonically increasing according to the linear extension $\preceq$ of S, and paths are joined at common suffixes. By ordering the children of each node according to $\preceq$, operations can be performed in $O(|S|)$ time, using the algorithms in [78, 114]. *Decoding* (i.e. the inverse of the component mapping) is achieved for free as a by-product of computing operations in these trees.

We can apply this technique to a general ordered set P, although we can no longer guarantee that labels will be singletons, or that there will be no empty nodes. We order the results of $\mathcal{C}$ according to $\preceq$, and form the tree by joining elements at common prefixes (or suffixes). If a common prefix is not the code of any element, this results in the creation of an *empty* node. As above, the code for $x \in P$ is the union of all labels on the path from x to the root. To illustrate, consider the lattice in Figure 4.2. This lattice is not distributive since $a \sqcap (b \sqcup c) = a \sqcap \top = a$, but $(a \sqcap b) \sqcup (a \sqcap c) = \bot \sqcup c = c$. The tree T_1 implements the spanning set $S_1 = \{\downarrow a, \downarrow b, \downarrow c, \downarrow d, \downarrow e, \downarrow f\}$, where $\preceq$ is the given order of S_1 and elements are assigned numeric values according to $\preceq$. In this case, no empty nodes are created, but there is one edge with a non-singleton label. The second tree, T_2, implements the spanning set $S_2 = \{\downarrow\{b, d\}, \downarrow\{b, c\}, \downarrow b, \downarrow\{a, f\}, \downarrow\{a, e\}, \downarrow a\}$, where $\preceq$ is the given order of S_2. Here, two empty nodes were created as well as edges with non-singleton labels.

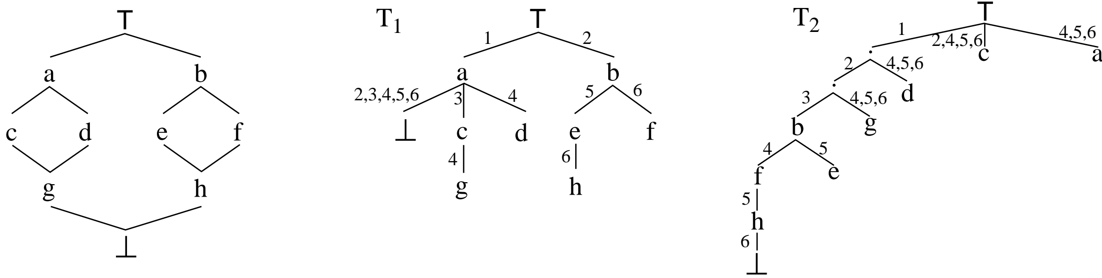

Figure 4.2: Tree representation

Performing unions, intersections, and subset checking is accomplished by locating the position of the two elements in the tree and comparing the labels along the paths from these elements to the root. To be more concrete, consider the above spanning set S_2 that preserves meets with union (and thus subsumption with subsets). To check if $x \leq y$, we incrementally compare the set of labels $\mathcal{C}(x)$ and $\mathcal{C}(y)$ on the paths from x and y to the root, respectively. From the structure of the spanning set, we know that $x \leq y$ if and only if $\mathcal{C}(x) \supseteq \mathcal{C}(y)$. Since the components in labels are monotonically ordered within labels and along these paths, this comparison is linear in the size of the label sets. For example, $\mathcal{C}(g) = \{1, 2, 4, 5, 6\} \supseteq \{1, 4, 5, 6\} = \mathcal{C}(d)$, so $g \leq d$, but $\mathcal{C}(g) \not\supseteq \{1, 2, 3, 4, 5\} = \mathcal{C}(h)$, so $g \not\leq h$.

To compute $x \sqcap y = z$, we incrementally union the labels on the two paths from x and y to the root. Then we descend the tree using this union to find the meet element. For example, to find $c \sqcap d$, we find $\mathcal{C}(c) \cup \mathcal{C}(d) = \{1, 2, 4, 5, 6\}$, and descend to find that this set is $\mathcal{C}(g)$. Thus, $c \sqcap d = g$.

Operations can be further optimized by finding the node in the tree at which the two paths converge, and only considering the portions of the paths below this point (which is how the algorithm in [77] works). We can avoid further comparisons above this point, since the remainders of the two paths coincide. For details of the tree traversal algorithms that compute subsumption, meets and joins for distributive lattices, see [77, 78]. The modifications required to handle our generalization of this tree representation are trivial.

Determining the space complexity of these trees is not as simple as before. Since empty nodes must have at least two children, the number created will be bounded by $|P|$. Non-singleton labels cause these trees to be non-linear in the size of the ordered set, but the code sharing can still greatly reduce the overall space requirements. Operations

are no longer bounded by the depth of the tree, but rather by the number of labels on a path to the root. This is also true in the distributive case, but there each edge has a singleton label. As before, children of nodes are ordered lexicographically by edge labels. Since the labels from an element to the root are in strictly decreasing lexicographic order, operations are linear in the size of the codes using an adaptation of the algorithms in [78, 114], and decoding can still be achieved efficiently. Clearly, the tree constructed will depend on the ordering $\preceq$ of S (which is usually a linear extension of S), so algorithms need to be developed that find orderings for which optimal trees can be found or approximated.

4.3.3 Logical terms

We can also implement sets using logical terms, embedding our order into the lattice of generalized atomic formulae [121]. Terms with no structure can be used in a manner similar to bit-vectors using anonymous variables in place of 0. For example, 11010 can be represented as $p(1,1,_,1,_)$ for an arbitrary predicate p. However, terms can also be used to capture some structural information. Set union and intersection correspond to unification and anti-unification, respectively. Subset checking becomes term subsumption checking. We can also exploit the hierarchical structure of an up-set to reduce storage requirements. It is important to note that logical terms also provide the ability to implement unions that produce the entire domain (e.g. $\bot$) by unification failure. To illustrate, consider the ordered set in Figure 2.2. We may represent the up-set $\uparrow kit\ fox$ by the term $p(canine(fox(kfox)), wild, _)$ and $\uparrow collie$ by $p(canine(dog(collie)), _, domestic)$. Their intersection is obtained by anti-unification: $p(canine(_), _, _)$ (representing $\uparrow canine$). If we represent $\uparrow dog$ by $p(canine(dog(_)), _, domestic)$ and $\uparrow wolf$ by $p(canine(wolf), wild, social)$, we capture the fact that $dog \sqcap wolf = \bot$ with unification failure. Although desirable, we shall see that this is not always easy to achieve. We show in section 4.8 how compact *tree terms* (terms in which all variables are anonymous) can be derived from spanning sets. In section 4.9 we discuss the use of coreference constraints, as provided by logical variables, in encoding.

A disadvantage of logical terms is that specifying filled positions (with an atom or functor) requires more space than the 1 bit required for the bit-vector approach. An advantage is that not all unfilled positions need to be specified. In our example, the subset for $\uparrow canine$, $p(canine(_), _, _)$, only reserves three additional spaces (via anonymous variables); additional spaces become available dynamically through instantiation at these positions. It is also possible to implement parallel algorithms in hardware for unification and anti-unification of tree terms.

4.3.4 Sparse logical terms

Sparse terms [51] allow an efficient and direct implementation of hierarchical sets by providing the tree-shaped structure of ordinary terms as well as several other key features. They are similar to the directed acyclic graphs (DAGs) and feature structures used in natural language processing systems (e.g. [118]). In [104], the use of DAGs to implement encodings is explored in detail. In Chapter 6, we develop sparse terms in detail as a universal implementation for encoding.

4.3.5 Integer vectors

Natural numbers can be used to implement chains or anti-chains. All finite total orders of size n are isomorphic to the interval $[1, n]$, providing a simple and efficient binary number implementation using only $\log n$ space for each element. We find it convenient to use the dual of the natural order, so that 1 is the top of the chain. Each integer then represents all the preceding elements in the chain (i.e. k, $1 \leq k \leq n$ represents the interval $[1, k]$). Subsets can be checked in an obvious way ($a \subseteq b$ if and only if $a \leq b$), while $a \cup b = max(a, b)$ and $a \cap b = min(a, b)$.

Every anti-chain of size n is isomorphic to the flat lattice of the natural numbers $[1, n]$. In this lattice, each pair of unequal integers is treated as meet and join incompatible. To represent an anti-chain, we assign each element a unique number in $[1, n]$, and use 0 to represent the empty set. The set operations are defined as follows:

subsets: $i \subseteq j \Leftrightarrow i = j$ or $i = 0$.
union: $i \cup j$ fails if $i \neq 0, j \neq 0$ and $i \neq j$. Otherwise $i \cup j = max(i, j)$.
intersection: $i \cap j = i$ if $i = j$, otherwise $i \cap j = 0$.

By viewing an ordered set as being composed of a number of chains or anti-chains, we can use integer vectors to represent up-sets.

Definition 4.5 *Let P be an ordered set. A partition $Q = \{P_1, P_2, \cdots, P_m\}$ of P is called a* chain (anti-chain) partition *if the suborder defined on each of the P_i is a chain (anti-chain).*

An anti-chain Q is called *meet (join) incompatible* if every pair of elements in Q is meet (join) incompatible. In essence, the above partitions view a partial order as a number of parallel interconnected chains or anti-chains. As an example, consider the chain and meet incompatible anti-chain partitions of the ordered set of Figure 2.2, shown in Figures 4.3 (where each chain is represented vertically) and 4.4 (where each anti-chain is represented horizontally).

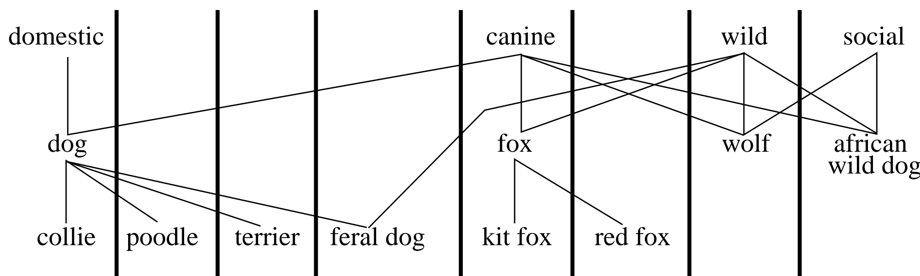

Figure 4.3: Chain partition of the ordered set in Figure 2.2

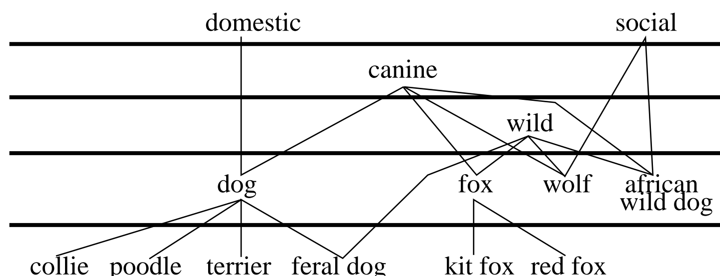

Figure 4.4: Meet incompatible anti-chain partition of the ordered set in Figure 2.2

Integer vectors can be used to represent up-sets using chain or incompatible anti-chain partitions by assigning one position in the vector to each chain or anti-chain, since we only need to represent at most one element of each. The integer vector encoding in [97] uses a chain partition. A partition of size k requires vectors of length k. We need to have a special integer (we use 0) to place in a position when the up-set does not contain any element from the corresponding chain or anti-chain. For chain partitions, an entry represents all preceding elements in the corresponding chain. For meet incompatible anti-chain partitions, at most one element from each anti-chain can be present, so a non-zero entry represents an element plus the absence of all other elements in the anti-chain. The entire vector then represents the union of the information represented in its entries. We denote each entry of a vector V of size k as $V[i]$, $1 \leq i \leq k$. The set operations for chain partitions are defined as follows:

subsets: $V_1 \subseteq V_2 \Leftrightarrow \forall 1 \leq i \leq k, V_1[i] \leq V_2[i]$.

union: $V_1 \cup V_2 = V \Leftrightarrow \forall 1 \leq i \leq k, V[i] = max(V_1[i], V_2[i])$.

intersection: $V_1 \cap V_2 = V \Leftrightarrow \forall 1 \leq i \leq k, V[i] = min(V_1[i], V_2[i])$.

In our example, we represent $\uparrow kit\ fox$ by $[0,0,0,0,3,0,1,0]$ and $\uparrow terrier$ by $[2,0,1,0,1,0,0,0]$. Their intersection is the code for $\uparrow canine$: $[0,0,0,0,1,0,0,0]$. We now consider the set operations for meet incompatible anti-chain partitions:

subsets: $V_1 \subseteq V_2 \Leftrightarrow \forall 1 \leq i \leq k, V_1[i] = V_2[i]$ or $V_2[i] = 0$.

union: $\forall 1 \leq i \leq k$, $V_1 \cup V_2 = V$ fails if $V_1[i] \neq 0, V_2[i] \neq 0$ and $V_1[i] \neq V_2[i]$.
Otherwise $V[i] = max(V_1[i], V_2[i])$.

intersection: $V_1 \cap V_2 = V \Leftrightarrow \forall 1 \leq i \leq k,\ V[i] = V_1[i]$ when $V_1[i] = V_2[i]$
Otherwise $V[i] = 0$.

In our example, we represent $\uparrow$*kit fox* by $[0,1,1,2,5]$ and $\uparrow$*terrier* by $[1,1,0,1,3]$. The intersection of these is $[0,1,0,0,0]$ ($\uparrow$*canine*) but their union fails.

Bit-vectors can be viewed as a special case of both forms of integer vectors, where an ordered set is seen as a set of n chains or anti-chains of size 1. Note that any singleton anti-chain is vacuously meet incompatible. For both cases, 0 represents that no element of the corresponding chain or anti-chain is in the subset, and 1 represents that the first, and only, element is in the subset. The logical operations of AND and OR compute the set operations. Also, flat logical terms (i.e. terms with no functors or nesting) provide a direct logical realization of incompatible anti-chain vectors, using anonymous variables instead of 0 and atomic symbols instead of integers. For example, the above vectors could be represented as $p(_,1,1,2,5)$ and $p(1,1,_,1,3)$, respectively. Note that we can apply sparse representations to integer vectors (i.e. introduce indices for non-zero elements, and eliminate the zero entries), as we show in Chapter 6.

4.4 Infinite Suborders

Our analysis of encoding assumes that the original ordered set is finite. For many applications we require the integration of a finite order with one or more infinite orders such as real numbers, integers, strings, intervals, etc. Clearly, we cannot compute codes for the elements of an infinite suborder *a priori*, so we need to be able to perform taxonomic operations involving one or more elements in an infinite suborder dynamically. Provided certain restrictions are obeyed, we can permit portions of our set to be infinite while still benefiting from encoding. As far as we know, such a formulation has not previously been described.

Suppose we have an ordered set P with an infinite suborder Q. We can encode the finite portion of P using the techniques described in this chapter provided Q obeys the following:

Classification Given any element x in Q, we must be able to ascertain that in fact $x \in Q$. Note that one infinite suborder may be a suborder of another infinite suborder (e.g. integers and reals). Thus, we must be able to classify elements correctly (e.g. checking if $1 < 3/2$, we must classify 1 as a rational number).

Locality The order relation within Q must be locally determined and efficient. This is required for operations involving only elements of Q, so that encoding is not necessary. For example, it is easy to locally determine order between integers, strings or intervals of real numbers. If meets or joins must also be preserved in Q, then these operations must also be locally computable.

Encapsulation In order to compute operations involving one element in Q and another not in Q, Q must be bounded (i.e. it must have top and bottom elements, $\top_Q$ and $\bot_Q$)[6]. In a sense, these elements provide entry and exit points to the infinite suborder and can be incorporated into the finite portion of the ordered set. Normally, the bottom will simply be the bottom of the ordered set. We also require that Q be closed. That is, $Q = \downarrow\top_Q \backslash \downarrow\bot_Q \cup \{\bot_Q\}$ and $Q = \uparrow\bot_Q \backslash \uparrow\top_Q \cup \{\top_Q\}$. This requires that the bounds of Q must provide the only entry and exit points. We show in Chapter 5 that bounding and closure implies that Q must be a module within P.

These requirements allow us to encode the finite portions of an ordered set, including the bounds of any infinite suborder, as though the entire set was finite. For operations involving elements within an infinite suborder, we use locality to compute the operation. In the case of meets and joins, the result will also be in the infinite suborder. For operations involving one element in an infinite suborder Q and another not in Q, we can use the one of the bounds in place of this element. If the result of a meet or join is this bound, it can be replaced by the original element. We provide more details of how this may be achieved when we discuss modulation in Chapter 5.

[6] It may be possible to relax this restriction to require a finite number of maximal and minimal elements of the infinite suborder. This, however, complicates taxonomic operations. For example, the meet of two elements not in an infinite suborder Q may result in any element in Q, not just one of the maximal elements.

4.5 Spanning Sets of Principal Down-sets and Up-sets

4.5.1 All principal down-sets

The transitive closure encoding introduced in [2] and described in section 3.5 encodes a partial order with k elements using bit-vectors of length $k-1$ as follows. Each element $a_i \in P$ (except $\perp$) is assigned a unique integer i in $[1, k-1]$. For any element $a_j \in P$, bit i, $1 \leq i < k$ will be 1 if and only if $a_i \leq a_j$. The actual procedure given in [2] produces this encoding in a bottom-up manner, starting at $\perp$ and propagating codes upwards towards $\top$.

In terms of our framework, this procedure simply computes the spanning set S_1 consisting of every principal up-set for the bottom-up case described, or the spanning set of every principal down-set for the top-down case. The encoding is the characteristic vector implementation of these component mappings. The orders induced by these spanning sets are isomorphic to the original order. As an example, the following figure shows a lattice, a component mapping, and its bit-vector implementation.

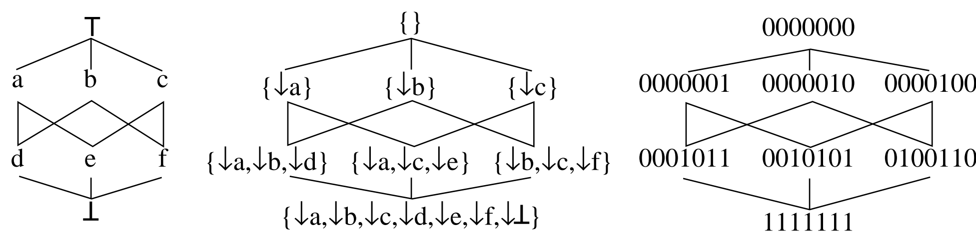

Figure 4.5: Principal down-set encoding

The interval encoding in [1] is closely related, and is based on the same spanning set S_1, but implemented using sets of integer intervals. Recall from section 4.3 that, under a total order $\preceq$ of S_1, any set of components can be implemented using the corresponding set of intervals in $\preceq$. In [1], an algorithm for finding an optimal ordering is described. A *cover tree* T for the ordered set P is identified by choosing, for each element $x \in P$, the parent that has the most ancestors. The authors show that the total order $\preceq$ defined by the postorder traversal of T produces interval set codes that minimize the overall space requirements of the encoding (i.e. the total number of intervals for all codes)[7]. In case P is a tree, for each element $x \in P$, $\mathcal{C}(x)$ will be exactly one interval. To illustrate, Figure 4.6 shows a cover tree T, the preorder number of T, and an interval implementation of the lattice in Figure 4.5.

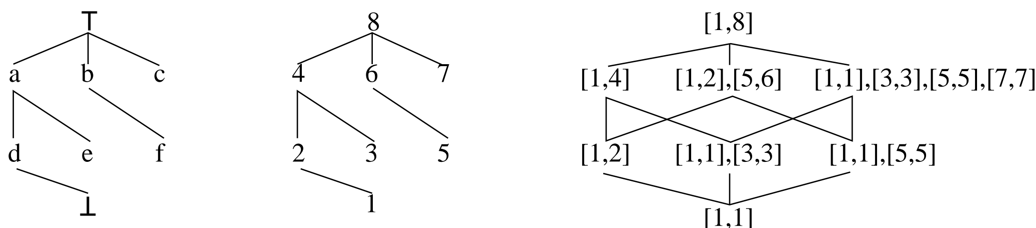

Figure 4.6: Cover tree, preorder numbering and interval encoding for the lattice in Figure 4.5

Theorem 4.2 *Let L be a lattice. The set of principal down-sets of L forms a spanning set S_1 that preserves joins through intersection.*

Proof: We need to show that $e_1 \sqcup e_2 = e$ if and only if $\mathcal{C}(e_1) \cap \mathcal{C}(e_2) = \mathcal{C}(e)$. Suppose that $e_1 \sqcup e_2 = e$. Consider any principal down-set $\downarrow a \in \mathcal{C}(e_1) \cap \mathcal{C}(e_2)$. Then $e_1 \leq a$ and $e_2 \leq a$. By the definition of join, $e \leq a$, so $\downarrow a \in \mathcal{C}(e)$. Consider any principal down-set $\downarrow a \in \mathcal{C}(e)$. Then $e \leq a$. Since $e_1 \leq e$ and $e_2 \leq e$, $\downarrow a \in \mathcal{C}(e_1) \cap \mathcal{C}(e_2)$. Therefore, $\mathcal{C}(e_1) \cap \mathcal{C}(e_2) = \mathcal{C}(e)$.

Assume that $\mathcal{C}(e_1) \cap \mathcal{C}(e_2) = \mathcal{C}(e)$. Since $\downarrow e \in \mathcal{C}(e)$, $e_1 \leq e$ and $e_2 \leq e$. So e is an upper bound of e_1 and e_2. Now if $e_1 \sqcup e_2 = a$ then $\downarrow a \in \mathcal{C}(e_1) \cap \mathcal{C}(e_2)$, so $\downarrow a \in \mathcal{C}(e)$ and $e \leq a$, implying e = a. □

[7] This optimum in fact only holds when we do not consider merging two adjacent intervals (e.g. $[i_1, i_2]$ and $[j_1, j_2]$ where $j_1 = i_2 + 1$ could be replaced by $[i_1, j_2]$). When merging is performed, the total order identified may not be optimal. However, adjacent intervals in the codes resulting from $\preceq$ may be merged to provide an approximation to the optimal.

The dual of the above theorem shows that the set of principal up-sets forms a spanning set that preserves meets through intersection.

Such spanning sets lead to a particularly time efficient implementation using a Boolean matrix in which entry $(i,j) = 1$ if $i \leq j$ and 0 otherwise [114]: checking subsumption can be accomplished in constant time[8]. In [93], the encodings of [2] are used in the typed feature logic programming language $\mathcal{TDL}$, and in [45], a transitive closure encoding implemented using tree terms is proposed.

4.5.2 Principal down-sets of meet irreducible elements

Since a focus of encoding is space and time efficiency, we are interested in finding spanning sets with a minimal number of elements. In [2] it is recognized that not all principal down-sets are required to maintain joins. This led to the compact encoding algorithm described in section 3.5. Let us denote the set of meet irreducible ancestors of an element e as $\mu(e)$. It is easy to show that μ is monotonically increasing as we descend the taxonomy from parents to children (i.e. if $e_1 \leq e_2$ then $\mu(e_2) \subseteq \mu(e_1)$). We now show that in a lattice, μ also preserves joins.

Lemma 4.1 *Let L be a lattice. Then for $e_1, e_2 \in L$, $e_1 \leq e_2$ if and only if $\mu(e_2) \subseteq \mu(e_1)$.*

Proof: $\Rightarrow$ By the monotonicity of μ, if $e_1 \leq e_2$, $\mu(e_2) \subseteq \mu(e_1)$.

$\Leftarrow$ Suppose $\mu(e_2) \subseteq \mu(e_1)$ and $e_1 \not\leq e_2$. Clearly, any ancestor of e_2 that does not subsume e_1 must not be meet irreducible. So e_2 cannot be meet irreducible. If two of the parents of e_2 subsume e_1, then the meet of these two parents is not unique. Thus, at least one parent p of e_2 does not subsume e_1. Since p cannot be meet irreducible, we can continue until we have an ancestor of e_2 that is a child of $\top$ and does not subsume e_1. But all children of $\top$ are meet irreducible. $\square$

Theorem 4.3 *The set of principal down-sets for the meet irreducible elements of a lattice L, $S_{\mathcal{M}(L)} = \{\downarrow e | e \in \mathcal{M}(L)\}$, forms a spanning set that preserves joins through intersection.*

Proof: The component mapping for the set of principal down-sets of meet irreducible elements is defined as $\mathcal{C}(x) = \{\downarrow e | e \in \mu(x)\}$. Consider any two elements e_1 and e_2. If $\mathcal{C}(e_1) = \mathcal{C}(e_2)$ then $\mu(e_1) = \mu(e_2)$ and so $\mu(e_1) \subseteq \mu(e_2)$ and $\mu(e_2) \subseteq \mu(e_1)$. By the above lemma, $e_2 \leq e_1$ and $e_1 \leq e_2$, so $e_1 = e_2$. Thus, $\mathcal{C}$ is one-to-one and so $S_{\mathcal{M}(L)}$ forms a spanning set.

We need to show that $e_1 \sqcup e_2 = e$ if and only if $\mathcal{C}(e_1) \cap \mathcal{C}(e_2) = \mathcal{C}(e)$. This is equivalent to showing that $e_1 \sqcup e_2 = e$ if and only if $\mu(e_1) \cap \mu(e_2) = \mu(e)$.

$\Rightarrow$ Suppose that $e_1 \sqcup e_2 = e$. Consider any meet irreducible $x \in \mu(e_1) \cap \mu(e_2)$. Then $e_1 \leq x$ and $e_2 \leq x$. By the definition of join, $e \leq x$, so $x \in \mu(e)$. Consider any meet irreducible $x \in \mu(e)$. Then $e \leq x$. Since $e_1 \leq e$ and $e_2 \leq e$, $x \in \mu(e_1) \cap \mu(e_2)$. Therefore, $\mu(e_1) \cap \mu(e_2) = \mu(e)$.

$\Leftarrow$ Assume that $\mu(e_1) \cap \mu(e_2) = \mu(e)$. Then e is an upper bound of e_1 and e_2, since $\mu(e) \subseteq \mu(e_1)$ and $\mu(e) \subseteq \mu(e_2)$ imply that $e_1 \leq e$ and $e_2 \leq e$, by the above lemma. For any upper bound x of e_1 and e_2 we have $\mu(x) \subseteq \mu(e_1)$ and $\mu(x) \subseteq \mu(e_2)$, by the lemma, and so $\mu(x) \subseteq \mu(e_1) \cap \mu(e_2)$. From our assumption and the lemma, we deduce that $\mu(x) \subseteq \mu(e)$ and $e \leq x$, implying $e_1 \sqcup e_2 = e$. $\square$

The dual of this theorem states that the set of principal up-sets for the join irreducible elements of a lattice L, $S_{\mathcal{J}(L)} = \{\uparrow e | e \in \mathcal{J}(L)\}$, forms a spanning set that preserves meets through intersection. Also note that the order induced by $S_{\mathcal{M}(L)}$, for a lattice L, is isomorphic to the suborder obtained by restricting L to the meet irreducible elements $\mathcal{M}(L)$.

The compact encoding in [2] for a lattice L implements the component mapping of $S_{\mathcal{J}(L)}$, for the bottom-up case described, and $S_{\mathcal{M}(L)}$ for the top-down case, using characteristic vectors. We again use the lattice in Figure 4.5 to illustrate. Figure 4.7 shows the component mapping for $S_{\mathcal{M}(L)}$ and its bit-vector implementation.

For distributive lattices, the ideal tree in [78, 114] encodes $S_{\mathcal{M}(L)}$ in a tree data structure (see section 4.3.2) that permits computation of both meets and joins in $O(|\mathcal{M}(L)|)$ time. We now demonstrate that $S_{\mathcal{M}(L)}$ and $S_{\mathcal{J}(L)}$ are the smallest spanning sets of principal down-sets or up-sets that preserve not only joins and meets, respectively, but also subsumption.

[8] This is simply the adjacency matrix implementation of the transitive closure graph of the ordered set.

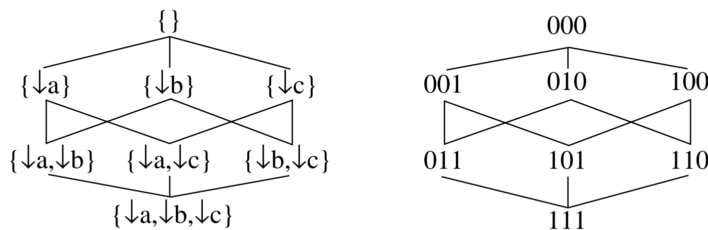

Figure 4.7: Meet irreducible encoding

Lemma 4.2 *Let L be a lattice. Then every meet irreducible element of L must be a factor*[9] *of at least one down-set in a spanning set of down-sets.*

Proof: If not, it has the same component mapping as its parent. □

Theorem 4.4 *Let L be a lattice. If $|\mathcal{M}(L)| = m$, then any spanning set of principal down-sets that preserves subsumption with supersets must have at least m down-sets.*

This theorem is a direct consequence of the above lemma. Thus, for subsumption preservation, the smallest size spanning set of principal down-sets or up-sets has $min(|\mathcal{M}(L)|, |\mathcal{J}(L)|)$ elements.

Theorem 4.5 *Let L be a lattice and S a spanning set of down-sets on L that preserves joins by set intersection. Then every component of S must be a principal down-set.*

Proof: Suppose there is a component $Q = \downarrow\{q_1, q_2, \cdots, q_n\} \in S$ that is not principal (i.e. $n \geq 2$). Consider the join of any two of the maximal elements, say q_1 and q_2. Clearly the join must properly subsume both of these elements (since $q_1||q_2$, and so $Q \notin \mathcal{C}(q_1 \sqcup q_2)$). But $Q \in \mathcal{C}(q_1)$ and $Q \in \mathcal{C}(q_2)$, so $Q \in \mathcal{C}(q_1) \cap \mathcal{C}(q_2)$. Thus, S does not preserve joins by intersection. □

This last theorem, along with the Spanning Set Duality theorem, shows us that $|S_{\mathcal{M}(L)}|$ ($|S_{\mathcal{J}(L)}|$) is the minimum size of any spanning set that preserves joins (meets).

Much of the above discussion assumes that we are encoding a lattice. For a general ordered set P, the spanning set of all principal down-sets preserves subsumption, as does $S_{\mathcal{M}(P)}$, provided we recognize the meet irreducible elements of the order, which do not necessarily have a single parent as shown by Theorem 2.2. Both techniques, however, can be used to encode for join bases (meet crests) instead of joins (meets). When computing a join base $a \sqcup b$, the intersection of the two component mappings $\mathcal{C}(a) \cap \mathcal{C}(b) = \mathcal{C}_{a \sqcup b}$ will result in a component set that represents the join base. If the join base is a singleton (i.e. a join: $a \sqcup b = c$), then $\mathcal{C}(c) = \mathcal{C}_{a \sqcap b}$; otherwise, we need to find the maximal elements whose component mappings are subsets of $\mathcal{C}_{a \sqcup b}$.

4.6 Spanning Sets of Prime Down-sets and Up-sets

This section describes spanning sets of prime down and up-sets and shows a direct correspondence with spanning sets of principal up-sets and down-sets, respectively. Although not standard in lattice theory, we define *prime down-sets* analogously to prime ideals: a down-set $\downarrow Q$ of a lattice L is prime, if when $x \sqcap y \in \downarrow Q$, either $x \in \downarrow Q$ or $y \in \downarrow Q$. That is, we cannot get into $\downarrow Q$ from two elements not in $\downarrow Q$. For an ordered set P, we generalize this definition: a down-set $\downarrow Q$ of P is prime, if when $x \sqcap y \subseteq \downarrow Q$, either $x \in \downarrow Q$ or $y \in \downarrow Q$.

Lemma 4.3 *Let L be a lattice. If e is an element and $\downarrow e$ is its principal down-set then $\lfloor L \backslash \downarrow e \rfloor$ (i.e. the principal factors of the up-set $L \backslash \downarrow e$) are all join irreducible.*

[9] Recall that a factor is a maximal element of a down-set.

Proof: Suppose f is a minimal element in $L\backslash{\downarrow}e$ and is not join irreducible. Then it has at least two children, x and y. Both x and y must be in ${\downarrow}e$ or else f is not minimal. Since both x and y are subsumed by e (by the definition of down-set), e is an upper bound of x, y. But $f \not\leq e$ and f clearly must be the join of x and y, so we don't have a lattice. $\square$

Theorem 4.6 *Let L be a lattice. Then ${\uparrow}Q$ is principal if and only if $L\backslash{\uparrow}Q$ is prime.*

Proof: $\Rightarrow$ Suppose an up-set ${\uparrow}Q$ is principal, $Q = \{e\}$. Let ${\downarrow}Q_{\overline{e}} = L\backslash{\uparrow}e$. By the dual of the above lemma, the factors of this down-set must all be meet irreducible. Suppose $\exists\ e_1$ and e_2 such that $e_1 \sqcap e_2 \in {\downarrow}Q_{\overline{e}}$ but $e_1 \notin {\downarrow}Q_{\overline{e}}$ and $e_2 \notin {\downarrow}Q_{\overline{e}}$. By the construction of ${\downarrow}Q_{\overline{e}}$, $e_1 \in {\uparrow}e$, so $e \leq e_1$. Similarly, $e \leq e_2$. Therefore $e \leq e_1 \sqcap e_2$. But then $e_1 \sqcap e_2 \in {\uparrow}e$.

$\Leftarrow$ Suppose an up-set ${\uparrow}Q$ is not principal. Consider any two factors e_1 and e_2 of ${\uparrow}Q$. Since $e_1 \sqcap e_2 \notin {\uparrow}Q$, $L\backslash{\uparrow}Q$ is not a prime down-set. $\square$

We say that $L\backslash{\uparrow}e$ is the prime down-set induced by e, the elements not in its principal up-set. In [102], Mellish shows that if we have a spanning set of prime down-sets, we can guarantee that the meet of two elements can be found with unification (down-set union). With the Spanning Set Duality Theorem (Theorem 4.1), we can see that a spanning set of down-sets that preserves meets with union can be easily constructed from the join irreducible elements. The above theorem shows that this is a spanning set of prime down-sets and the final result of the previous section shows that this is the smallest such spanning set. Naturally, for an ordered set P, the order induced by a spanning set of prime down-sets is dually isomorphic to that produced by a $S_{\mathcal{J}(P)}$.

As an example, in Figure 4.8, $\mathcal{J}(P) = \{d, e, a, c\}$. The first encoding shows a bit-vector implementation of the spanning set $S_{\mathcal{J}(P)} = \{{\uparrow}d, {\uparrow}e, {\uparrow}a, {\uparrow}c\}$ where meets are preserved with intersection. The spanning set of prime down-sets associated with these join irreducible elements is $S_{\overline{\mathcal{J}(P)}} = \{{\downarrow}c, {\downarrow}a, {\downarrow}\{b, c\}, {\downarrow}\{a, b\}\}$, preserving meets with union. The second encoding shows the implementation of this spanning set.

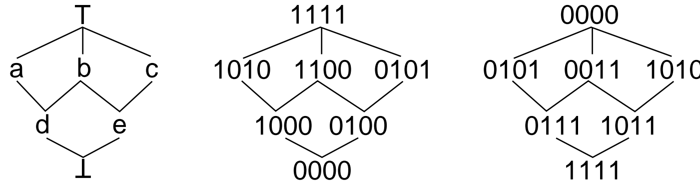

Figure 4.8: Principal up-set and prime down-set encodings

The encoding of [77] represents each element by the set Q of join irreducible elements that it doesn't subsume, which is equivalent to the set of prime down-sets induced by elements in Q. The underlying spanning set therefore consists of the prime down-sets induced by $\mathcal{J}(P)$, and so preserves meets with union. The bit-vector implementation of such a spanning set is identical to the bitwise negation of the bit-vector implementation of $S_{\mathcal{J}(\mathcal{P})}$, as can be seen in the above example.

We have now shown a correspondence between the compact encoding of [2] based on set intersection (e.g. bitwise AND), and prime down-set encodings of [77, 102] based on set union (e.g. bitwise OR and logical term unification). There is, however, one important distinction to make for the approach of Mellish [102]. In the above construction, if the meet of two elements is $\perp$, set union will produce the entire domain (i.e. the entire spanning set S) because $\perp$ is treated as any other element. It is also possible (as Mellish's approach requires) to implement meet incompatibility as failure (e.g. with unification failure). This strict requirement essentially treats the ordered set as $\perp$-unbounded. We discuss in sections 4.8 and 4.9 how incompatibility as failure may be achieved.

4.7 Spanning Sets of Compound Down-sets and Up-sets

So far, we have studied spanning sets of principal down-sets that preserve joins with intersection, and spanning sets of prime (possibly compound) down-sets that preserve meets with union. We showed that the latter case is equivalent to spanning sets of principal up-sets that preserve meets with intersection. Between these extremes lie spanning sets

that preserve subsumption, but neither meets nor joins. We now consider such spanning sets, which may contain down-sets with multiple factors. Recall that the factors of a down-set $\downarrow Q$ is the set of maximal elements of $\downarrow Q$ (which is an anti-chain). Initially, we focus on spanning sets that do not permit multiple occurrences of factors. That is, elements that are factors of several spanning set components. Later in the section, we relax this restriction.

Our first theorem shows that, for any spanning set S of down-sets, there is a spanning set containing only meet irreducible factors which is no larger than S. This means that, as in section 4.5, we need only be concerned with irreducible elements when constructing minimal size spanning sets.

Theorem 4.7 *Let S be a spanning set for a lattice L that preserves subsumption. Then there exists another spanning set S' that (i) contains no more down-sets than S (ii) preserves subsumption and (iii) has only meet irreducible factors in all down-sets.*

Proof: Suppose we have a subsumption preserving spanning set S for which there exists a down-set $\downarrow Q = \downarrow\{q_1, q_2, \cdots, q_m\}$ where q_i is not meet irreducible, for some $1 \leq i \leq m$. Further suppose we remove q_i from Q (this may reduce the number of components in the spanning set if Q becomes empty or equivalent to another down-set in S). This produces a new spanning set S' that is identical to S except that $Q' = \{q_1, \cdots, q_{i-1}, q_{i+1}, \cdots, q_n\}$ has fewer elements than Q (and so $\downarrow Q' \subseteq \downarrow Q$) and $S' = S \backslash \{\downarrow Q\} \cup \{\downarrow Q'\}$. The component mapping for S' will be denoted by $\mathcal{C}'$. The only difference between $\mathcal{C}$ and $\mathcal{C}'$ (modulo the name change of Q to Q') is that the mapping of elements in $\downarrow Q \backslash \downarrow Q'$ does not contain Q' (i.e. descendants of q_i not subsumed by some $q_j \in Q$, $i \neq j$ and $1 \leq j \leq m$, are not in $\downarrow Q'$).

If S' does not preserve subsumption, then $\exists e_1, e_2 \in L$ for which $e_2 \not\leq e_1$ and $\mathcal{C}'(e_1) \subseteq \mathcal{C}'(e_2)$ (due to the monotonicity of the component mapping for spanning sets of down-sets, the case $e_2 \leq e_1$ but $\mathcal{C}'(e_2) \not\subseteq \mathcal{C}'(e_1)$ cannot occur). Since $\mathcal{C}(e_1) \not\subseteq \mathcal{C}(e_2)$, $\mathcal{C}'(e_1) = \mathcal{C}(e_1) \backslash \{\downarrow Q\} \subseteq \mathcal{C}(e_2) = \mathcal{C}'(e_2)$. This situation is only possible if $e_1 \leq q_i$ but $e_1 \notin \downarrow Q'$ and $e_2 \notin \downarrow Q$, otherwise $\mathcal{C}(e_1) \subseteq \mathcal{C}(e_2)$.

Let $p_1, p_2, \cdots, p_n$, $n \geq 2$ be the parents of q_i. Since $q_i \in Q$, none of its parents are in $\downarrow Q$, so $\downarrow Q \notin \mathcal{C}(p_1)$ and $\mathcal{C}(p_1) \subseteq \mathcal{C}(q_i) \subseteq \mathcal{C}(e_1)$, $\mathcal{C}(p_1) \subseteq \mathcal{C}(e_2)$. Thus, $e_2 \leq p_1$. Similarly, $e_2 \leq p_2, \cdots, p_n$. Also $e_2 \not\leq q_i$, since $e_2 \notin \downarrow Q$. This implies that L is not a lattice, since q_i must be the meet of its parents, but e_2 is a lower bound of these parents not subsumed by q_i. Therefore S' must preserve subsumption. Clearly, we can similarly remove all non-meet irreducible elements from S to produce a subsumption preserving spanning set that has no more components than S. □

Hereafter, we assume that the components of all spanning sets have only meet irreducible factors. We have already shown that no spanning set S of compound down-sets can preserve joins by intersection. Can S preserve meets with union? If it does, the Spanning Set Duality Theorem tells us that there is a corresponding spanning set S' that preserves meets with intersection. Since S' can have only principal up-sets for components, S must be a spanning set of prime down-sets.

We now focus on how compound down-sets can reduce the size of a spanning set that preserves only subsumption. First let us consider when two elements can be factors of the same down-set.

Theorem 4.8 *Let P be an ordered set and S be a spanning set of down-sets for P with no multiple occurrences of factors. Then S preserves subsumption if and only if, for every compound down-set $\downarrow Q \in S$ with factors e_1, e_2, $\nexists$ an element that is (i) a descendant of the parent of e_1, but not of e_1 itself and (ii) a descendant of e_2.*

Proof: $\Rightarrow$ Suppose e_1 and e_2 are factors of the same down-set $\downarrow Q$ of S, and $\exists$ an element q that is (i) a descendant of the parent p of e_1, but not of e_1 and (ii) a descendant of e_2. Since e_1 is a factor of no down-set in S other than $\downarrow Q$, $\mathcal{C}(e_1) = \mathcal{C}(p) \cup \{\downarrow Q\}$. Also $q \leq p$ and $q \leq e_2$, so $\mathcal{C}(p) \subseteq \mathcal{C}(q)$ and $Q \in \mathcal{C}(q)$. Therefore, $\mathcal{C}(e_1) \subseteq \mathcal{C}(q)$, but $q \not\leq e_1$, so S does not preserve subsumption.

$\Leftarrow$ Suppose for every down-set $\downarrow Q \in S$, if e_1, e_2 are factors of $\downarrow Q$ then $\nexists$ an element that is (i) a descendant of the parent p of e_1, but not of e_1 itself and (ii) a descendant of e_2. So if e_1, e_2 are factors of $\downarrow Q$ then for every element q, if $q \leq p$ and $q \leq e_2$, then $q \leq e_1$. If S does not preserve subsumption, then $\exists x, y \in P$ for which $\mathcal{C}(y) \subseteq \mathcal{C}(x)$, but $x \not\leq y$. Let e_1 be a maximal ancestor of y for which $x \not\leq e_1$ and $\mathcal{C}(e_1) \subseteq \mathcal{C}(x)$. If e_1 is non-meet irreducible, then the meet of the parents Q of e_1 is unique. Clearly, this meet must be e_1. Also, every parent of e_1 must subsume x, otherwise it is not maximal, so x is a lower bound of Q. But then $x \leq e_1$.

Thus e_1 is meet irreducible, and so must be a factor of some down-set $\downarrow Q$. Since $\mathcal{C}(e_1) \subseteq \mathcal{C}(x)$, $\downarrow Q \in \mathcal{C}(x)$. Since $x \not\leq e_1$, $\downarrow Q$ must have at least one other factor e_2 for which $x \leq e_2$. But then our assumption is violated, since $e_1, e_2 \in Q$, $x \leq p$ where p is the parent of e_1, $x \not\leq e_1$ and $x \leq e_2$. □

Figure 4.9 illustrates the case when e_1 and e_2 do not satisfy the constraints of the theorem. If we put e_1 and e_2 as factors of the same component, the component mapping for the descendant d will be a superset of that of e_1, and so we will incorrectly conclude that $d \leq e_1$.

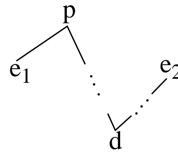

Figure 4.9: Elements that cannot be in the same down-set

In [24], Caseau proposes an encoding scheme that preserves subsumption. His algorithm computes a subsumption preserving spanning set of down-sets, implemented with bit vectors. Through his notion of "gene sharing", compound down-sets may be formed. The algorithm proposed computes the spanning set incrementally as the ordered set is constructed from top to bottom. When meet irreducible elements are added, the algorithm adds the element as a factor of the first down-set permitted according to the above theorem. When non-meet irreducible elements are added, a check is made to see if the conditions of the theorem are violated. If they are, a factor of some down-set contributing to this violation is moved to another down-set in a process called "gene mutation".

Below is an example ordered set and the encoding that the algorithm determines immediately before and after the addition of element i (which causes a gene mutation, since i is (i) a descendant of the parent a of c, but not of c itself and (ii) a descendant of e). The spanning sets prior to and following the mutation are respectively $\{\downarrow a, \downarrow b, \downarrow\{c, e\}, \downarrow\{d, f\}\}$ and $\{\downarrow a, \downarrow b, \downarrow\{c, e\}, \downarrow d, \downarrow f\}$. The rightmost encoding shows a more compact encoding than Caseau's that satisfies the above theorem, but which the algorithm does not find. The spanning set for this encoding is $\{\downarrow a, \downarrow b, \downarrow\{c, f\}, \downarrow\{d, e\}\}$.

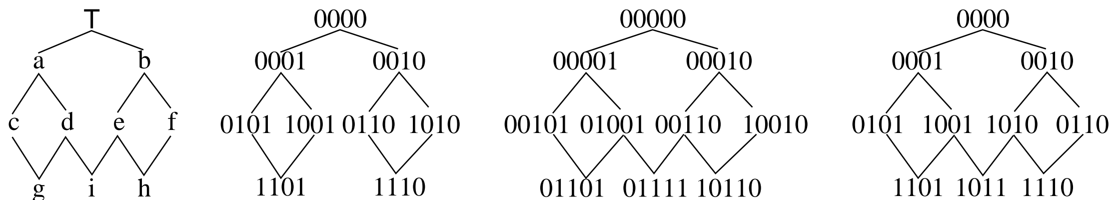

Figure 4.10: Subsumption preserving encoding

Intuitively, it seems that subsumption preservation should not rely on the existence of meets or joins. However, Caseau's incremental algorithm forms the minimal (i.e. Dedekind-MacNeille) lattice completion of the given ordered set, which is potentially costly.

Theorem 4.9 *Let P be an ordered set. Then the elements that must be represented as factors of down-sets for a subsumption preserving spanning set are the meet irreducible elements of P.*

It is easy to show that $\mathcal{M}(P) = \mathcal{M}(L_P)$, where L_P is the minimal lattice completion of P. The proof of the above theorem follows from this fact and previous theorems. Thus, we don't need to actually realize the lattice completion. Rather, we need only recognize which elements are meet irreducible.

4.7.1 Finding a minimal subsumption preserving spanning set is NP-Hard

In Caseau's paper, a suggestion is made for the gene mutation process to attempt to detect more compact ways to rectify a violation, once detected. Both the original algorithm and this suggested improvement, however, provide approximations to the problem of finding a minimal spanning set of down-sets that preserves subsumption. Unfortunately, as we show through the next theorem, this problem is NP-Hard. This result is related to one suggested in [77] regarding the bounded dimension of an ordered set, $dim_2(P)$.

Definition 4.6 Minimum Subsumption Preserving Spanning Set. *Given a lattice L and a positive number $k \leq |L|$. Is there a spanning set of down-sets of size k that preserves subsumption?*

Theorem 4.10 *The Minimum Subsumption Preserving Spanning Set problem is NP-Complete.*

Proof: Consider the following problem, which is known to be NP-Complete [69]:

Partition into Cliques. Given a graph $G = (V, E)$ and a positive number $k \leq |V|$. Is there a partition of G into k cliques?

We provide a polynomial transformation from this problem to our problem. Let us construct a lattice L from $G = (V, E)$, where $n = |V|$ and $e = |E|$, as follows: (i) start with a $\top$ element ($\perp$ will be left implicit). (ii) Add n elements $P_1, P_2, \cdots, P_n$, where $P_i < \top$. (iii) Add n elements $v_1, v_2, \cdots, v_n$, where $v_i < P_i$. (iv) Add $m = n(n-1)/2 - e$ elements as follows: For each pair of vertices v_i, v_j, where $i < j$, that does not have a connecting edge in E, add an element (v_i, v_j) where $(v_i, v_j) \leq P_i$ and $(v_i, v_j) \leq v_j$.

Claim: L has a subsumption preserving spanning set of size $n + k$ if and only if G has a partition into k cliques.

$\Rightarrow$ Suppose L has a subsumption preserving spanning set S of size $n + k$. First note that, by theorem 4.8, S must contain n principal down-sets corresponding to the P_i meet irreducibles. Since the (v_i, v_i) elements are not meet irreducible, all other down-sets must be composed of the v_i elements. Further, there must be exactly k of these down-sets. Consider any one of these down-sets $\downarrow Q$. Claim: The corresponding vertices in G forms a clique. Consider any pair of elements $v_i, v_j \in Q$, where $i < j$. Since they are factors of the same down-set, $\nexists$ an element that is (i) a descendant of the parent of v_i, but not of v_i itself and (ii) a descendant of v_j. By the above construction, the only possible element for which this could occur is (v_i, v_j), which only exists if v_i, v_j are not connected by an edge. Thus, v_i, v_j have a connecting edge. Therefore, the corresponding vertices within each of these k down-sets forms a clique in G.

$\Leftarrow$ Suppose G has a partition into k cliques. Each of the P_i meet irreducibles must form a down-set for any spanning set that preserves subsumption on L. This makes n down-sets. Consider any one of the k cliques, Q. Claim: The corresponding meet irreducibles in L can be factors of the same down-set. By the theorem, any pair v_i, v_j, $i < j$, can be factors of the same down-set provided $\nexists$ an element that is (i) a descendant of the parent of v_i, but not of v_i itself and (ii) a descendant of v_j. By the above construction, the only possible element for which this could occur is (v_i, v_j), which only exists if v_i, v_j are not connected by an edge. But since v_i, v_j are in a clique, they are connected by an edge. Thus, the corresponding meet irreducibles within each of these k cliques can be factors of the same down-set in a spanning set that preserves subsumption on L. □

Figure 4.11 shows an example of this reduction. Elements a, b, c, d form a clique in the graph and can also be factors of the same down-set in a subsumption preserving spanning set for the lattice.

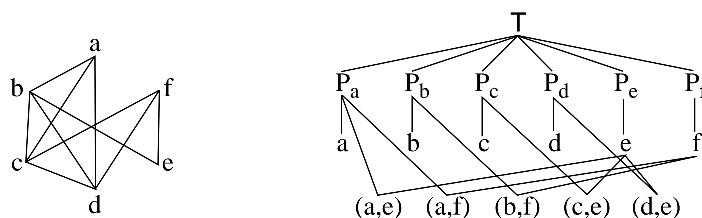

Figure 4.11: Transformation of a graph to a lattice

4.7.2 Multiple occurrences of factors

Although, it may seem unnecessary for an element to be a factor of more than one down-set, more compact spanning sets may result by allowing multiple occurrences of factors. We characterize the general conditions such spanning sets must satisfy. In Figure 4.12, any spanning set without multiple occurrences of factors has at least ten elements. It is easy, however, to verify that the spanning set $S = \{\downarrow\{a, b, c, d, e, f\}, \downarrow\{a, b, c, g, h, i\}, \downarrow\{a, d, e, g, h, j\}, \downarrow\{b, d, f, g, i, j\}, \downarrow\{c, e, f, h, i, j\}\}$ preserves subsumption.

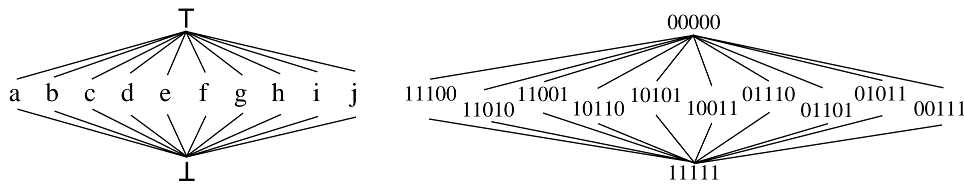

Figure 4.12: Subsumption preserving encoding

Theorem 4.11 *Let P be an ordered set and S be a spanning set of meet irreducible down-sets for P. Then S preserves subsumption if and only if, for every meet irreducible element $e_1 \in \mathcal{M}(P)$, $\nexists$ an element x for which (i) x is a descendant of the parent of e_1, but not of e_1 itself and (ii) $\forall{\downarrow}Q \in S$ where e_1 is a factor of ${\downarrow}Q$, $\exists$ a factor e_2 of ${\downarrow}Q$ for which x is a descendant of e_2.*

The proof of this theorem is similar to that for Theorem 4.8. Figure 4.13 illustrates the case when the constraints of the theorem are violated for an element e_1. If every component for which e_1 is a factor, has one of the f_i as a factor, the component mapping for the descendant d will be a superset of that of e_1, and so we will incorrectly conclude that $d \leq e_1$. Allowing multiple occurrences of factors provides greater flexibility to subsumption encoding and permits more compact spanning sets. Finding a minimal sized spanning set is undoubtedly NP-Hard, but it may be possible to design an approximation algorithm (such as an extension to Caseau's greedy algorithm) that performs better than existing algorithms.

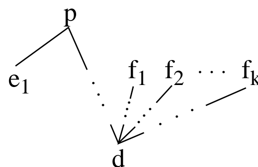

Figure 4.13: Violation of subsumption

There have been two encoding schemes ([61, 79]) that permit multiple factors in compound spanning sets. Although the algorithms are too detailed to describe fully in this thesis, there are several issues of interest.

The algorithm in [61] constructs a bit-vector encoding using two passes over a lattice L: one upwards and one downwards. The resulting encoding preserves subsumption with subsets, and thus implements a spanning set of up-sets. One of the goals of this encoding is to provide efficient meet computations (join computations are described, but are not efficiently handled). Meet computations are achieved in this subsumption preserving encoding by using an interesting indexing method. Suppose L is the lattice to encode, and S is the spanning set of up-sets generated by the algorithm. With each non-meet irreducible element $x \in L, x \notin \mathcal{M}(L)$, one of the components $s_x \in S$ is associated in the following way:

Definition 4.7 *Let L be a lattice, and S be a spanning set of up-sets on L. Then S* discriminates *the non-meet irreducible elements of L if $\forall x \in L, x \notin \mathcal{M}(L)$, $\exists s_x \in S$ for which (i) $x \in s_x$ and (ii) if $y \in L, y \notin \mathcal{M}(L)$, and $y \in s_x$, then $x \leq y$ (i.e. x is the unique minimum non-meet irreducible element of s_x).*

To compute a meet $x \sqcap y$, we first check if $x \leq y$ or $y \leq x$. If neither of these hold, we know that the meet must be a non-meet irreducible element. We then intersect the component mappings for x and y: $\mathcal{C}_{a\sqcap b} = \mathcal{C}(x) \cap \mathcal{C}(y)$. Using a linear extension $\preceq$ of the lattice L, a linear ordering is formed for S; the details of the particular linear extension formed in [61] are unimportant, but what is essential is that, for two non-meet irreducible elements $x, y \in L$, if $x \leq y$ then $s_x \leq s_y$. By the manner in which S is formed, the meet will correspond to either the first or second spanning set component in $\mathcal{C}_{a\sqcap b}$ corresponding to a non-meet irreducible element[10]. Using a bit-vector mask (which contains a

[10] A generalization of this property is proven below.

1 in each position corresponding to a non-meet irreducible component), these components can be identified. A table indexed by the bit corresponding to these components is then used to decode the meet.

Note that this approach to decoding meets through a table lookup can be applied to any spanning set that preserves subsumption with subsets and discriminates the non-meet irreducible elements. In particular, the transitive closure method of [2] could use this indexing technique for efficient decoding.

Rather than elaborate on the details of this algorithm, it will be more fruitful to elucidate its important contributions. First, although this approach usually requires less space than the transitive closure method of [2], there are cases in which a spanning set contains redundancy. By the dual of Theorem 4.7, subsumption preservation needs only to deal with join irreducible elements. For the indexing method to function, however, we need to keep those components associated with non-meet irreducible elements (which may contain non-join irreducible factors). However, there are other redundancies that may result from the algorithm in [61]: (i) it is possible to have a factor that is meet irreducible but not join irreducible; such factors can be removed (by Theorem 4.7). (ii) it is possible to have duplicate and redundant components. By remediating these problems in the resulting spanning set, the algorithm could be improved.

As an example, consider the ordered set in Figure 4.14. The first encoding results from the algorithm in [61]. The spanning set that is implemented is $S = \{\uparrow\{e,g\}, \uparrow\top, \uparrow e, \uparrow g, \uparrow f, \uparrow\{e,g\}\}$. Note that the component $\uparrow\{e,f\}$ appears twice (in the first and last bit positions), which is clearly unnecessary. Secondly, this component is redundant, since it is not associated with any non-meet irreducible element, and $\uparrow e$ and $\uparrow g$ are both components of S. A more efficient spanning set that preserves the desired properties is $S' = \{\uparrow\top, \uparrow e, \uparrow g, \uparrow f\}$; its bit-vector implementation is shown on the right-hand side of Figure 4.14.

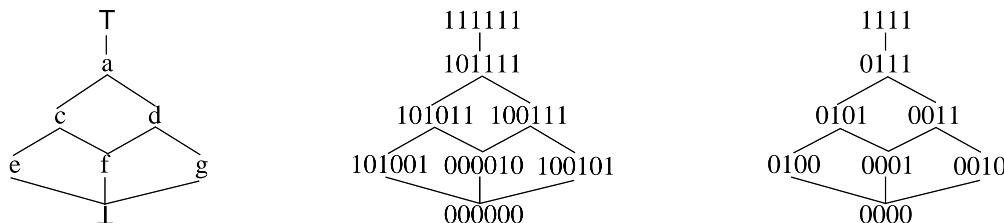

Figure 4.14: Example encodings that discriminate non-meet irreducible elements

We now formulate the encoding problem tackled by the algorithm in [61] in a general manner, which may lead to the development of more efficient solutions. Suppose we have a lattice L and we wish to construct a spanning set S that (i) preserves subsumption with subsets (i.e. x is subsumed by y ($x \leq y$) if and only if $\mathcal{C}(x) \subseteq \mathcal{C}(y)$), and (ii) discriminates non-meet irreducible elements. For each element $x \in L, x \notin \mathcal{M}(L)$, define the set $\mathcal{R}(x) = \{y \in L | y \leq x, \forall z \in L, z \notin \mathcal{M}(L), y \leq z \rightarrow x \leq z\}$. These are the elements that are subsumed by x, but not by any other non-meet irreducible element that is not an ancestor of x. Note that $x \in \mathcal{R}(x)$. Now the problem can be described as constructing a subsumption preserving spanning set of up-sets S with the restriction that $\forall x \in L, x \notin \mathcal{M}(L)$, $\exists s_x \in S$ for which the factors of s_x are a subset of $\mathcal{R}(x)$ (i.e. $\lfloor s_x \rfloor \subseteq \mathcal{R}(x)$). This ensures that S discriminates non-meet irreducible elements. The component s_x will be called the component associated with x.

We know from theorem 4.7 that to preserve subsumption, we need only be concerned with the join irreducible elements $\mathcal{J}(L)$. Thus, for optimality, we need only consider the join irreducible elements of $\mathcal{R}(x)$; if there are none, then we can use $s_x = \uparrow x$.

The interesting result is as follows:

Theorem 4.12 *Let L be a lattice and S be a spanning set of up-sets for L such that*

i. S preserves subsumption

ii. S discriminates non-meet irreducible elements

iii. S is partitioned into those components that are associated with non-meet irreducible elements, S_1 and those that are not, S_2

iv. There is a linear extension $\preceq$ of S_1.

Then, for any meet $a \sqcap b = c$, consider $\mathcal{C}_{a \sqcap b} = \mathcal{C}(a) \cap \mathcal{C}(b)$.

i. if $\mathcal{C}_{a \sqcap b} = \mathcal{C}(a)$, then $a = c$.

ii. if $\mathcal{C}_{a \sqcap b} = \mathcal{C}(b)$, then $b = c$.

iii. if $\mathcal{C}_{a \sqcap b} = \emptyset$, then $c = \perp$.

iv. if $\mathcal{C}_{a \sqcap b} \cap S_1 = \{s_x\}$, then: if $a \leq x$ (or $b \leq x$), then $\perp = c$, otherwise $x = c$.

v. if $|\mathcal{C}_{a \sqcap b} \cap S_1| \geq 2$, then let s_x and s_y be the first and second elements (according to $\preceq$) in $\mathcal{C}_{a \sqcap b} \cap S_1$. If $a \leq x$ (or $b \leq x$), then $y = c$, otherwise $x = c$.

Proof: Let L be a lattice and S a spanning set of up-sets for L that satisfies the above conditions. Consider any meet $a \sqcap b = c$ and the set $\mathcal{C}_{a \sqcap b} = \mathcal{C}(a) \cap \mathcal{C}(b)$. Since S preserves subsumption, cases (i-iii) hold.

Now suppose s_x is the first component (according to $\preceq$) of $\mathcal{C}_{a \sqcap b} \cap S_1$. It is possible that $a \in \mathcal{R}(x)$ and $b \in \mathcal{R}(x)$, in which case $a \leq x$ and $b \leq x$ (i.e. the factors of s_x are below a, b and x). Since x subsumes every element in $\mathcal{R}(x)$, either both a and b subsume x or both are subsumed by x. Since $\preceq$ is a linear extension, if both a and b subsume x, clearly $x = a \sqcap b$.

Claim: For any component $s_y \in \mathcal{C}_{a \sqcap b} \cap S_1$, $s_y \neq s_x$, both a and b subsume y (or conversely, it is impossible for y to subsume a and b). Suppose y subsumes a and b. Thus, $a \in \mathcal{R}(y)$ and $b \in \mathcal{R}(y)$. Since $\preceq$ is a linear extension of S_1, x must also subsume a and b, and either $y \leq x$ or $x || y$. In the first case, we can infer that $x \in \mathcal{R}(y)$, which is impossible, since x is non-meet irreducible. In the second case, we can infer that L is not a lattice.

Thus, in case x subsumes a and b, we can select the second element s_y of $\mathcal{C}_{a \sqcap b} \cap S_1$. If no such element exists, then $a \sqcap b = \perp$, otherwise $a \sqcap b = y$. □

This theorem provides a general and efficient procedure for computing and decoding meets, which abstracts the algorithm in [61]. Given a and b, if neither subsumes the other, and the intersection of their component mappings is non-empty, then we can determine their meet simply by extracting the first component s_x corresponding to a non-meet irreducible element x. If x does not subsume either a or b, then $a \sqcap b = x$; otherwise extract the second component s_y corresponding to a non-meet irreducible element y. If no such component exists, $a \sqcap b = \perp$; otherwise $a \sqcap b = y$.

Another approach that implements spanning sets of compound down-sets, described in [79], decomposes an ordered set P into *co-atomic* sublattices[11]. By grouping elements together that have the same set of subsuming co-atoms, the authors show that the resulting order is a co-atomic lattice. If P is already a co-atomic lattice, then the resulting order is isomorphic to P. This partitioning is performed repeatedly on each group of elements, forming a tree of co-atomic lattices that is used as the basis for generating a bit-vector encoding of the original ordered set. Their algorithm can also be viewed as computing a spanning set of compound up-sets, although the details are beyond the scope of this thesis.

4.8 Spanning Set Decomposition

We have seen that with spanning sets of down-sets, we can only preserve joins with principal down-sets (section 4.5) and meets with prime down-sets (section 4.6)[12]. The preceding section discussed combining principal down-sets into compound down-sets while still preserving subsumption. In this section, we describe how decompositions of spanning sets that satisfy certain restrictions can lead to some efficient implementations using, for example, integer vectors or logical terms.

[11] A co-atomic lattice is a lattice in which every element is a meet of one or more co-atoms.

[12] Without the use of additional constraints, such as coreference, as discussed in section 4.9, and in [102, 104].

Suppose a spanning set S for an ordered set P is decomposed into $\alpha_1, \alpha_2, \ldots, \alpha_k$ (i.e. $\alpha_1 \cup \alpha_2 \cup \cdots \cup \alpha_k = S$). In order to use this decomposition, we modify the component mapping to return, in addition to each component, the subset containing it. We use the notation $\alpha(s)$ to denote that component s is in subset α. For example, if $\alpha(s) \in \mathcal{C}(e)$, then $e \in s$ and s is a member of the subset α. We say that an element of P is in a subset if it is in any of its constituent components. If we can guarantee that subsets possess certain structure, we can implement them with space logarithmic to the number of components, as opposed to the linear space required to represent the components individually.

4.8.1 Chain decomposition

For a spanning set S on an ordered set P, a chain partition, as defined in section 4.3.5, of the order induced by S is one form of chain decomposition. If the components of S are principal down-sets, a chain partition of S is also isomorphic to a chain partition of P. In general, if S is subsumption preserving, it corresponds to a chain product embedding of P, as we discuss below.

The key feature of a chain decomposition $S = \alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_k$ is that, given a component s_i of α_j, we can infer every component preceding s_i in the chain. Thus, we need not represent all components explicitly - the component mapping need only return at most one for each subset. Integer vectors, described in section 4.3.5, provide a direct and efficient implementation.

The *virtual time* proposal in [97], addressing the problem of global time in distributed systems, essentially performs a chain partition on a spanning set of principal down-sets implemented using integer vectors. At each of k sites, transitions are caused by internal state changes, and message sends and receives, forming a partial order based on precedence constraints among events (e.g. a send must precede its corresponding receive). Note that this partial order is not necessarily a lattice, since two sites may simultaneously send to each other. The transition events for each site represent local clock advances. Possible combinations of the local clocks constrain the possible global times. No global time is maintained in the system, but each site approximates it using its local time plus the times obtained from other processes by messages received.

The transitions at each site form a chain, interconnected by message sends and receives, producing a natural chain partition that is represented by a vector of k integers. Since the clock at each site is updated after each transition, the code of an event for site i consists of the code of its parent at this site, with the ith entry incremented and, if the event is a receive, the union is formed with the vector sent with this message. The underlying spanning set is thus the set of all principal down-sets, so it preserves joins but not meets. As an example, a three site system is depicted in Figure 4.15. A space reduction could be realized if down-sets were restricted to the meet irreducibles.

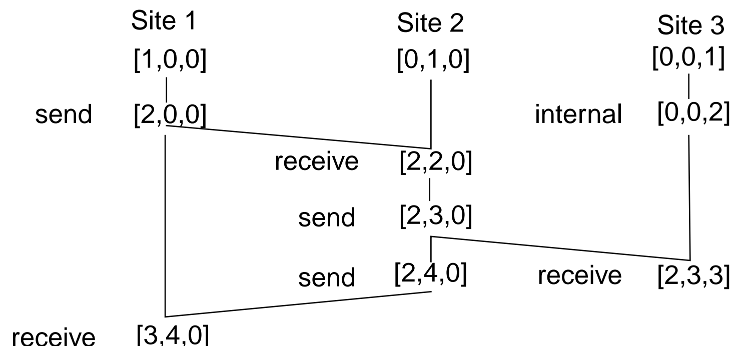

Figure 4.15: Distributed virtual time encoding

Generalizing this scheme requires partitioning an arbitrary spanning set S into the minimum number of chains, which is equivalent to finding the maximum sized anti-chain of S [74]. The cardinality of this anti-chain, called the *width* of S, determines the minimum number of chains needed to represent S, and thus the minimum size of a vector implementation. In the distributed system, the width is the number of sites. In general, determining the width of S is possible in $O(|S|^3)$ time [74]. The next theorem shows the space requirements for a balanced chain partition.

Theorem 4.13 *Let S be an ordered set with n elements and width k. Further suppose that there is chain partition of S into k chains of size n/k. Then the integer vector encoding for S on this partition requires $O(nk(\lfloor log(n/k) \rfloor + 1))$ space.*

Since each element requires a vector of size k, and the maximum sized integer in each vector is n/k (requiring $O(log(n/k))$ space), the result follows. Note: If k = 1, then we have a total order and we require $O(logn)$ space to represent each element. If k = n, then we have an anti-chain and we require $O(n)$ space for each element. In both cases, bit-vectors require $O(n)$ space.

Chain product embeddings

Chain partitions are in fact a special case of *chain product embeddings.*

Definition 4.8 *Let P be an ordered set, $\{C_1, \ldots, C_k\}$ be a set of chains, and $\tau : P \rightarrow C_1 \times \cdots \times C_k$ be a function from P to the cross product of these chains. Then τ is a* chain product embedding *if, for $x, y \in P$, $x \leq y$ if and only if $\tau(x) = (c_1^x, \ldots c_k^x)$, $\tau(y) = (c_1^y, \ldots c_k^y)$ and $c_i^x \leq_{C_i} c_i^y$ for $1 \leq i \leq k$.*

We define element i of the vector $\tau(x)$ as $\tau(x)[i]$ (i.e. $\tau(x)[i] = c_i^x$). A chain partition is the case when the C_i are chain suborders of P that partition P. Chain product embeddings are closely related to order dimension [144], and *encoding dimension* [79].

Theorem 4.14 *Let P be an ordered set. Then every chain partition of a subsumption preserving spanning set of up-sets S for P corresponds to a chain product embedding of P, and every chain product embedding τ of P corresponds to a chain partition of some spanning set of up-sets for P that preserves subsumption.*

Proof: $\Rightarrow$ Let S be a subsumption preserving spanning set of up-sets for P, and let $\{C_1, C_2, \ldots, C_k\}$ be a chain partition of S. Let us also define a special null component $s_\emptyset \notin S$ that subsumes every component of S. Define the mapping $\tau : P \rightarrow C_1 \times \cdots \times C_k$ as $\tau(x) = (c_1, c_2, \ldots, c_k)$ where, for $1 \leq i \leq k$, c_i is the least element in C_i that is in $\mathcal{C}(x)$. If $C_i \cap \mathcal{C}(x) = \emptyset$ (i.e. there is no element in chain C_i that is in $\mathcal{C}(x)$), then $c_i = s_\emptyset$. Thus, ignoring the null components in this mapping, $\mathcal{C}(x) = \uparrow\{c_1, c_2, \ldots, c_k\}$[13].

Claim: τ is a chain product embedding. If $x \leq y$, then $\mathcal{C}(x) \subseteq \mathcal{C}(y)$. Clearly, for $1 \leq i \leq k$, we have $\tau(x)[i] \leq_{C_i} \tau(y)[i]$[14]. Conversely, suppose for $1 \leq i \leq k$, we have $\tau(x)[i] \leq_{C_i} \tau(y)[i]$. Then $\mathcal{C}(x) \subseteq \mathcal{C}(y)$, so $x \leq y$.

$\Leftarrow$ Let τ be a chain product embedding of P into the set of chains $\{C_1, C_2, \ldots, C_k\}$. Define $|C_i| = n_i$. Define the spanning set $S = \{s_1^1, \ldots, s_{n_1}^1, s_1^2, \ldots s_{n_2}^2, \ldots, s_1^k, \ldots s_{n_k}^k\}$, where, for $1 \leq i \leq k, 1 \leq j \leq n_i$, we define $s_j^i = \{x \in P | j \leq_{C_i} \tau(x)[i]\}$. Note that $\alpha_i = \{s_1^i, \ldots s_{n_i}^i\}$, for $1 \leq i \leq k$, defines a chain partition of S.

Claim: S is a subsumption preserving spanning set of up-sets. If $x \leq y$, then for $1 \leq i \leq k$, $\tau(x)[i] \leq_{C_i} \tau(y)[i]$. Suppose $s_j^i \in \mathcal{C}(x)$. Since $j \leq_{C_i} \tau(x)[i]$ and $\tau(x)[i] \leq_{C_i} \tau(y)[i]$, $j \leq_{C_i} \tau(y)[i]$, and $s_j^i \in \mathcal{C}(y)$. Thus, $\mathcal{C}(x) \subseteq \mathcal{C}(y)$. Conversely, if $\mathcal{C}(x) \subseteq \mathcal{C}(y)$, then $\tau(x)[i] \leq_{C_i} \tau(y)[i]$, for all $1 \leq i \leq k$. Thus $x \leq y$. □

Chain products have a natural implementation using integer vectors. A nice description of encoding by embedding ordered sets in products of chains is given in [79]. Unfortunately, finding a minimal size product of chains into which an ordered set can be embedded is NP-Hard[15].

4.8.2 Meet incompatible decomposition

A *meet incompatible* subset $\alpha(s_1, s_2, \cdots, s_k) \subseteq S$ is a subset in which components are pairwise meet incompatible. That is, if $i \neq j$ then $\forall a \in s_i, b \in s_j$, $a \sqcap b = \perp$. If the spanning set is composed of down-sets, this is equivalent to $s_i \cap s_j = \{\perp\}$. For a meet incompatible subset α, any non-bottom element in α will be in exactly one of the constituent components. So if $\alpha(s_i) \in \mathcal{C}(x)$, then $x \in s_i$ and for all other components s_j of α, $x \notin s_j$. Within

[13] Recall that for a spanning set S, $\mathcal{C}(x) = \{s \in S | x \in s\}$ is an up-set in S.

[14] This holds even if $\tau(y)[i]$ or both $\tau(y)[i]$ and $\tau(x)[i]$ are equal to $s_\emptyset$.

[15] This is called finding the *encoding dimension* in [79], and is closely related to the NP-Hard problem of finding the dimension of an ordered set P (the minimum number k for which P can be embedded in a product of k chains).

this framework, subset checking, union and intersection are essentially the same as before. Now, however, if we are computing the union of two component mappings and they contain a subset α with different components, the union fails. This is facilitated by treating our lattice as $\bot$-unbounded.

A spanning set S of all principal down-sets of an ordered set P is isomorphic to P. In this case, a meet incompatible partition of S is just a meet incompatible anti-chain partition of P, as defined in section 4.3. This is the basis for the tree term encoding in [34], which gives a logical term encoding of tree shaped taxonomies. In general, however, this does not hold. Note that a decomposition need not partition the components of S. By allowing components to be members of more than one subset, implementing meet incompatibility as union failure may be more viable. In addition, even if we are not concerned with meet incompatibility, specifying that a set of components is meet incompatible can permit a large space savings, as shown for the following representations.

Bit-vectors Instead of representing a component in a subset of size n by one bit, we assign $\lfloor \log n \rfloor + 1$ bits to the subset and assign a number from $1 \ldots n$. For elements not in the subset, we place a 0 in these positions, as before. For an element in the subset, we place the number of the unique component containing this element. This derives the integer vector representation of section 4.3.

Logical terms In a term, we use one position for each subset. For elements not in the subset, we place an anonymous variable for ordinary terms and nothing for sparse terms. For an element in the subset, we place a unique symbol for the component containing this element. Unification and anti-unification operate as expected. We can exploit the hierarchical structure of terms by introducing a subset $\alpha(s_1, s_2, \cdots, s_k)$ at the functor for one of the components in $\sqcup\{s_1, s_2, \cdots, s_k\}$. This can provide a significant space savings over integer vector (or flat term) implementations. This is the form of tree term encodings discussed in [102]. More general term encodings permit the use of logical variables (coreference), as discussed in section 4.9.

As an example, Figure 4.16 shows a meet incompatible anti-chain partition of the spanning set $S_{\mathcal{M}(P)}$ (i.e. the principal down-sets associated with the meet irreducible elements) for the ordered set P in Figure 2.2. Note that since *dog* and *feral dog* are not meet irreducible, they do not have corresponding elements in Figure 4.16. Figure 4.17 then shows a logical term implementation of this partitioned spanning set.

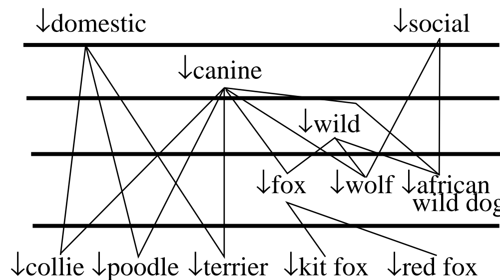

Figure 4.16: Meet incompatible decomposition

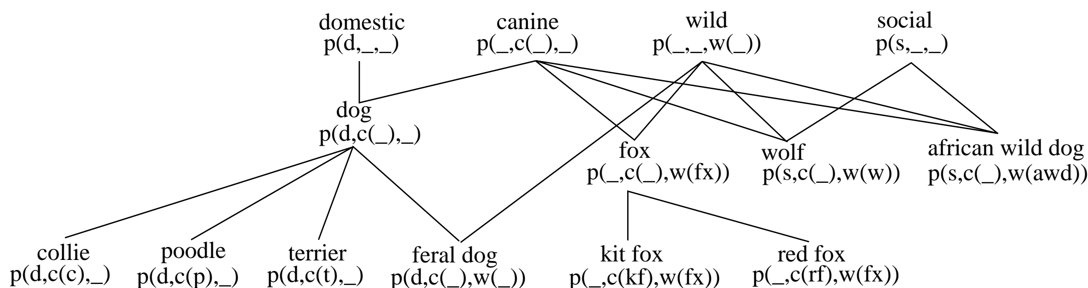

Figure 4.17: Logical term implementation of meet incompatible decomposition

In section 4.6 we analyzed spanning sets of prime down-sets and showed a direct correspondence with spanning sets of principal up-sets. We were able to then claim that any finite lattice has a spanning set of prime down-sets that preserves meets with union – this can easily be implemented using tree terms. In [102], an additional constraint is imposed on such spanning sets: if $a \sqcap b = \bot$ then the $\mathcal{C}(a) \cup \mathcal{C}(b)$ must fail. As we saw above, this may be accomplished using decomposition, but this is not always possible. Logical terms provide an implementation of this with unification failure. For implementations using tree terms, this constraint is formulated as follows.

Theorem 4.15 *[102] Let L be a lattice. Then L has a meet preserving tree term encoding if and only if, for any $a, b \in L$, $a \sqcap b = \bot$ if and only if there are two meet incompatible prime down-sets P_1, P_2 for which $a \in P_1$ and $b \in P_2$.*

Clearly, if there are two meet incompatible prime down-sets containing a and b, respectively, $a \sqcap b = \bot$. Requiring the converse, however, means that many lattices are not tree term encodable, according to Mellish's definition. Surprisingly, this includes even the lattice shown in Figure 4.5. Encoding this lattice so that $\bot$ is implemented as unification failure requires coreference, as shown in [47, 102]. Determining if a lattice is tree term encodable in this sense can be accomplished in polynomial time since all meet incompatibility must be incorporated into a decomposition.

In general, we want to find the smallest decomposition of a spanning set. Unfortunately, this is NP-Hard for the simpler case of partitioning an ordered set into meet incompatible subsets.

Definition 4.9 Meet Incompatible Ordered Set Partitioning. *Given an ordered set P, and a positive number $k \leq |P|$. Is there a partition of P into k meet incompatible subsets?*

Theorem 4.16 *Meet Incompatible Ordered Set Partitioning is NP-Complete.*

Proof: We give a polynomial transformation from the *Partition into Cliques* problem, described in section 4.7, to our problem. Let us construct an ordered set P from G as follows: Let $n = |V|$ and $e = |E|$. (i) Add n *vertex* elements $v_1, v_2, \cdots, v_n$. (ii) Add $m = n(n-1)/2 - e$ *non-edge* elements as follows: For each pair of vertices v_i, v_j, where $i < j$, which does not have a connecting edge in E, add the element (v_i, v_j) where $(v_i, v_j) < v_i$ and $(v_i, v_j) < v_j$.

Claim: P has a partition into $k+1$ meet incompatible subsets if and only if G has a partition into k cliques.

$\Rightarrow$ Suppose P has a partition into j meet incompatible subsets. Select one subset α' that does not contain any vertex element. If no such subset exists, $j = k$ and let $\alpha' = \emptyset$ (a trivial meet-incompatible subset) to bring the number of subsets to $k+1$; otherwise $j = k+1$. Consider any subset $\alpha \neq \alpha'$. Claim: The vertices corresponding to the vertex elements in α form a clique in G. Consider any pair of vertex elements $v_i, v_j \in \alpha$, where $i < j$. Since they are components of the same subset, they are incompatible. By the above construction, this could only occur if v_i, v_j have a connecting edge. Therefore, the corresponding vertices within each of these k subsets forms a clique in G.

$\Leftarrow$ Suppose G has a partition into k cliques. Consider any one of the k cliques, α. Claim: The corresponding elements in P can be components of the same subset. Any pair v_i, v_j, $i < j$, can be components of the same subset provided they are incompatible. By the above construction, this can only occur if v_i, v_j are connected by an edge. Since v_i, v_j are in a clique, they are connected by an edge. Thus, the corresponding elements within each of these k cliques can be components of the same subset. One additional meet incompatible subset can be formed from all of the non-edge elements. □

The following figure shows an example of the above transformation. It is easy to see that the elements a, b, c, d form a clique in the graph and are meet incompatible in the lattice.

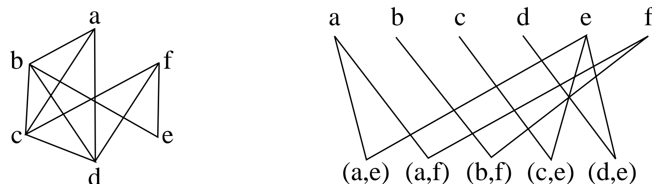

Figure 4.18: Transformation of a graph to a lattice

Any meet incompatible decomposition of a spanning set S of an ordered set P corresponds to a meet incompatible decomposition of the induced subset order of S, but not vice versa (since we may have two components $s_1, s_2 \in S$ for which $s_1 \sqcap_S s_2 = \bot_S$, but $s_1 \cap s_2 \neq \emptyset$). However, we can add elements (s_i, s_j) for any pair of components in S that are incompatible with respect to the induced order of S, but compatible with respect to the order of P. These elements would ensure equivalence between the two forms of meet incompatibility among components in S. Thus, the more general problem of finding a minimal meet incompatible decomposition of a spanning set is also NP-Hard.

4.8.3 Meet homogeneous decomposition

We now generalize the notion of meet-incompatible subsets; we hope that this generalization can be exploited in the development of new encoding algorithms. We call a subset $\alpha(s_1, s_2, \cdots, s_k)$ *meet homogeneous* (or simply homogeneous) if for any two distinct components $s_1, s_2 \in \alpha$, $a \in s_1$ and $b \in s_2$ implies $a \sqcap b \in s, \forall s \in \alpha$. That is, every element is either in 0, 1 or all the components of the subset. A meet incompatible subset can be viewed as a special case of a homogeneous subset, with the added restriction that $a \sqcap b = \bot$. Since any element in the subset will either be in exactly one or all of the components, we need to associate a special symbol, $\bot_\alpha$, with each subset indicating that every component is present. We redefine below the set operations for meet homogeneous subsets.

subsets: $\mathcal{C}(e_1) \subseteq \mathcal{C}(e_2) \Leftrightarrow \forall \alpha(x) \in \mathcal{C}(e_1)$, either
i. $\alpha(x) \in \mathcal{C}(e_2)$ or
ii. $\alpha(\bot_\alpha) \in \mathcal{C}(e_2)$.

union: $\mathcal{C}(e_1) \cup \mathcal{C}(e_2) = Q \Leftrightarrow \forall \alpha(z) \in Q$, either
i. $\alpha(z) \in \mathcal{C}(e_1)$ and $e_2 \notin \alpha$,
ii. $\alpha(z) \in \mathcal{C}(e_2)$ and $e_1 \notin \alpha$ or
iii. $\alpha(x) \in \mathcal{C}(e_1), \alpha(y) \in \mathcal{C}(e_2)$ and $x = y = z$ or $z = \bot_\alpha$.

intersection: $\mathcal{C}(e_1) \cap \mathcal{C}(e_2) = Q \Leftrightarrow \forall \alpha(z) \in Q$ either
i. $\alpha(z) \in \mathcal{C}(e_1)$ and $\alpha(z) \in \mathcal{C}(e_2)$,
ii. $\alpha(z) \in \mathcal{C}(e_1)$ and $\alpha(\bot_\alpha) \in \mathcal{C}(e_2)$ or
iii. $\alpha(\bot_\alpha) \in \mathcal{C}(e_1)$ and $\alpha(z) \in \mathcal{C}(e_2)$.

We can implement these operations with a modification to the sparse term or integer vector representations. By partitioning a spanning set into meet homogeneous subsets, we can achieve the benefits of meet incompatible subsets. The generality and flexibility of this structure, however, may permit more dense decomposition, decreasing the space requirements of an encoding, which may over-compensate for the increased operational complexity. To illustrate these concepts, consider the ordered set below. The minimal subsumption preserving spanning set of down-sets (with no multiple occurrences of factors) is $S = \{\downarrow a, \downarrow b, \downarrow c, \downarrow d, \downarrow e, \downarrow f, \downarrow h, \downarrow \bot\}$, which also preserves joins. Since every pair of components is compatible, meet incompatible decomposition provides no benefit. However, the following is one possible homogeneous decomposition of S: $\{\alpha_1(\downarrow a, \downarrow f, \downarrow h), \alpha_2(\downarrow b, \downarrow c, \downarrow d, \downarrow e), \alpha_3(\downarrow \bot)\}$. The component mapping corresponding to this decomposition is also shown in the figure.

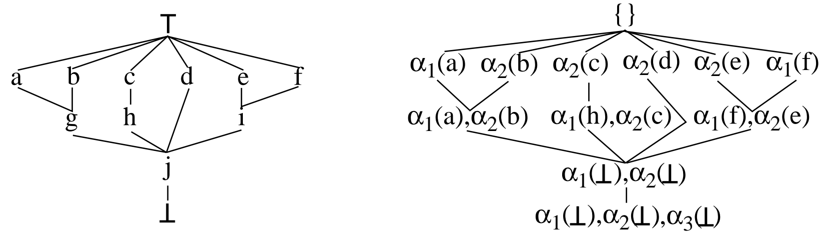

Figure 4.19: Meet homogeneous decomposition

4.9 Constraints and Coreference

We now develop a constraint-based examination of encoding, viewing both ordered sets and spanning sets as systems of constraints, and we formulate an integration of spanning sets with other forms of constraints. In this context, we are able to view the process of taxonomic encoding as a special case of constraint satisfaction. We first introduce the various types of constraints imposed by an ordered set. Preserving certain properties involves satisfying some of these. We next show how these constraints can be incorporated into the components of any subsumption preserving spanning set S of down-sets, through the use of *guarded* constraints, which are analogous to Dijkstra's guarded commands. This involves restating the initial constraints in terms of the components of S, and may alter the properties of S with respect to joins and meets. Many constraints can be implemented using techniques previously covered, such as chain partitions. We introduce coreference, such as that offered by logical variables, as a complementary implementation tool, formalized through equivalence classes of constraints. We also hypothesize about more general implementations.

4.9.1 Types of constraints

We will view constraints in a top-down manner as logical implications, denoted using the $\rightarrow$ symbol. Inferences on constraints are denoted using the $\vdash$ symbol, and sets of constraints are denoted using Γ. Given a set of elements and a constraint involving one or more of these elements, some consequence may follow through the application of modus ponens, where we use "$\wedge$" to denote logical conjunction and "$\vee$" to denote logical disjunction. For example, given a, b and $a \wedge b \rightarrow c$ we infer c, written $a, b, a{\wedge}b{\rightarrow}c \vdash c$. Different categories of constraints are distinguished by subscripting the Γ symbol. To be precise, we should also specify the partial order to which the constraints apply, but this is usually obvious.

Order constraints ($\Gamma_{\leq}$): The constraint imposed by the relation $a \leq b$ is simply $a \rightarrow b$. Thus, given element a and this constraint, we can infer element b. This constraint has been implicit in our analysis, and is integral to any subsumption preserving spanning set of down-sets. The cover relation dictates a set of cover constraints $\Gamma_{\leq'}$. Inferring $\Gamma_{\leq}$ from the reflexive and transitive closure of $\Gamma_{\leq'}$ follows.

Meet and join constraints ($\Gamma_{\sqcap}, \Gamma_{\sqcup}$): Suppose we have $a_1 \sqcap a_2 \sqcap \cdots \sqcap a_k = b$. Interpreting this logically, if we have all of the a_i, we can infer b. The constraint imposed by this relation is then $a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$[16]. An important effect of this constraint is that if $b \leq c$ then $a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow c$, even if none of $a_1, a_2, \cdots, a_k$ are comparable with c. From an encoding point of view, a meet constraint is satisfied by deducing new information. We later show how certain cases of meet constraints can be implemented using coreference.

Suppose we have $a_1 \sqcup a_2 \sqcup \cdots \sqcup a_k = b$. Interpreting this logically, if we have at least one of the a_i, we can infer b. The constraint is then $a_1 \vee a_2 \vee \cdots \vee a_k \rightarrow b$. Thus, from the uncertainty associated with a disjunction, we can infer a consequent. Due to the difficulty in implementing join constraints except with intersection, we will rely on previous techniques to satisfy $\Gamma_{\sqcup}$.

Meet and join incompatibility constraints ($\Gamma_{\perp}, \Gamma_{\top}$): Suppose we wish to implement $\perp$ as failure and we have a meet $a_1 \sqcap a_2 \sqcap \cdots \sqcap a_k = \perp$ that is minimal in the sense that any subset of the a_i is meet compatible. This results in k constraints: $a_1 \wedge \cdots \wedge a_{i-1} \wedge a_{i+1} \wedge \cdots \wedge a_k \rightarrow \neg a_i$, $1 \leq i \leq k$. Join incompatibility constraints can be defined dually, although we do not discuss them. The negation of an element a_i is a logical construct, the purpose of which is to cause an inconsistency in case we infer a_i . We show later how these constructs can be used to implement $\perp$ as failure.

As indicated, we only explicitly deal with $\Gamma_{\leq}, \Gamma_{\sqcap}$ and $\Gamma_{\perp}$. Thus, the antecedent of every constraint will be a conjunction (or a singleton). Our only rule of inference is modus ponens: $A, A{\rightarrow}b \vdash b$, where A is a conjunction of one or more elements. This rule enables us to deduce new elements from a given base set. Rather than allowing closure immediately, we provide an incremental inference procedure. This is important for encoding, since we need

[16] The generalization to meet-crest constraints is straightforward: if we have $a_1 \sqcap a_2 \sqcap \cdots \sqcap a_k = \{b_1, b_2, \ldots, b_j\}$, then the resulting constraint is $a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b_1 \vee b_2 \vee \cdots \vee b_j$. To keep our discussion clear, however, we will focus only on meet constraints and lattices.

to bound the number of inference steps in a deduction for the sake of efficiency. The following rules describe this procedure for a given initial set of constraints Γ, where $\vdash_1$ represents one application of modus ponens:

i. $\Gamma^0 = \{A \rightarrow b \mid \Gamma, A \vdash_1 b\}$ $(\supseteq \Gamma)$
ii. $\Gamma^{i+1} = \{A \rightarrow b \mid (\Gamma^i, A \vdash_1 c_1), \cdots, (\Gamma^i, A \vdash_1 c_k)$ and $\Gamma^i, c_1, \cdots, c_k \vdash_1 b\}$
iii. $\Gamma^* = \bigcup_{i=0}^{\infty} \Gamma^i$.

We say $\Gamma \vdash A \rightarrow b$ if there is some $i \geq 0$ for which $A \rightarrow b \in \Gamma^i$. Since Γ is finite, there will be a number $k \geq 0$ for which $\Gamma^{k+1} = \Gamma^k$, giving a fixed-point for this construction and $\Gamma^* = \Gamma^k$. Of course, using the above rules, we could specify a minimal set of constraints from which all others could be obtained (e.g. the entire order relation could be derived from the cover relation), and perform taxonomic operations using inference. However, to satisfy locality, every constraint we wish to satisfy needs to be immediately accessible (i.e. in a constraint set) or derivable in a small number of steps. For the spanning sets we have studied, all constraints are local. We show later how coreference may allow us to derive additional constraints in one inference step.

We will use the diamond lattice in Figure 4.1 to illustrate the specification and use of constraints. The cover constraints are $\Gamma_{\leq'} = \{a{\rightarrow}\top, b{\rightarrow}\top, c{\rightarrow}\top, \bot{\rightarrow}a, \bot{\rightarrow}b, \bot{\rightarrow}c\}$. The meet and join constraints are: $\Gamma_{\sqcap} = \{a{\wedge}b{\rightarrow}\bot, a{\wedge}c{\rightarrow}\bot, b{\wedge}c{\rightarrow}\bot\}$ and $\Gamma_{\sqcup} = \{a{\vee}b{\rightarrow}\top, a{\vee}c{\rightarrow}\top, b{\vee}c{\rightarrow}\top\}$, respectively. Recall that we showed in section 4.2 that no spanning set exists that preserves both meets and joins for this lattice. We later show how $\Gamma_{\sqcap}$ and $\Gamma_{\sqcup}$ may be preserved using coreference.

4.9.2 Augmented spanning sets

Each component of a spanning set S can be viewed as encompassing a set of constraints, and S preserves certain properties that we can infer from these constraints.

A down-set $\downarrow\{a_1, a_2, \cdots, a_k\}$ represents the set of constraints $\forall x \in \downarrow\{a_1, a_2, \cdots, a_k\}$, $x \rightarrow a_1 \vee a_2 \vee \cdots \vee a_k$. That is, given any element in the down-set, we can infer the disjunction of the factors. In case the down-set is principal, $\downarrow a$, we have $\forall x \in \downarrow a$, $x \rightarrow a$. An up-set $\uparrow\{a_1, a_2, \cdots, a_k\}$ embodies the constraints: $\forall x \in \uparrow\{a_1, a_2, \cdots, a_k\}$, $a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow x$. That is, given all of the factors, we can infer any element in the up-set. In case the up-set is principal, $\uparrow a$, we have $\forall x \in \uparrow a$, $a \rightarrow x$. Our analysis focuses on down-sets. We can also view a component itself as a set of constraints: the component s represents $x \rightarrow s$ for all $x \in s$.

Principal down-sets thus include a subset of $\Gamma_{\leq}$ and the spanning set of all principal down-sets induces this entire set. We showed in Theorem 4.3 that the meet irreducible elements embody the essence of joins, so $S_{\mathcal{M}(L)}$ preserves subsumption and joins while retaining only a subset of $\Gamma_{\leq}$. Compound down-sets, however, incorporate ambiguity. By merging the constraints of two or more principal down-sets, uncertainty arises as to which constraint is satisfied. Although we cannot preserve joins with such uncertainty (as we have shown), we can possibly preserve subsumption and meets (sections 4.6 and 4.7). In general, if $\mathcal{C}(x_1) \cup \mathcal{C}(x_2) \cup \cdots \cup \mathcal{C}(x_k) \supseteq \mathcal{C}(y)$ then $x_1 \wedge x_2 \wedge \ldots \wedge x_k \rightarrow y$. We denote the set of constraints of a spanning set S as $\Gamma(S)$. These can be expressed dually in terms of components: if $s_1 \cap s_2 \cap \cdots \cap s_k \subseteq s$ then $s_1 \wedge s_2 \wedge \ldots \wedge s_k \rightarrow s$.

A decomposition $S = \alpha_1 \cup \ldots \cup \alpha_k$ represents additional constraints. A chain decomposition induces the constraints $\forall 1 \leq i \leq k$, if $s_1, s_2 \in \alpha_i$ and $s_1 <_{\alpha_i} s_2$ then $s_1 \rightarrow s_2$. For a meet incompatible decomposition we have: $\forall 1 \leq i \leq k$, if $s_1, s_2 \in \alpha_i$ and $s_1 \neq s_2$ then $s_1 \rightarrow \neg s_2$. For a meet homogeneous decomposition, $\forall 1 \leq i \leq k$, if $s_1, s_2, s_3 \in \alpha_i$ and $s_1 \neq s_2$ then $s_1 \wedge s_2 \rightarrow s_3$.

To integrate constraints and spanning sets, we express constraints in terms of spanning set components. We now discuss how this affects the component mapping and taxonomic operations.

Definition 4.10 *A* component constraint *of an ordered set P is a constraint $s_1 \wedge \ldots \wedge s_{k-1} \rightarrow s_k$, where each of the antecedents and the consequent are subsets of P. A set of component constraints S_Γ of P is called an* augmented spanning set *if the function $\mathcal{C}_\Gamma : L \rightarrow 2^{S_\Gamma}$ defined by $\mathcal{C}_\Gamma(x) = \{s_1 \wedge s_2 \wedge \ldots \wedge s_{k-1} \rightarrow s_k \in S_\Gamma | \exists i, 1 \leq i \leq k, x \in s_i\}$ is one-to-one.*

Ordinary spanning sets are a special case, where $k = 1$ for every constraint. We associate a constraint with every element in its antecedent or consequent. An augmented spanning set for our example is as follows: $S_\Gamma =$

$\{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b \rightarrow \downarrow c,\ \downarrow a \wedge \downarrow c \rightarrow \downarrow b,\ \downarrow b \wedge \downarrow c \rightarrow \downarrow a\}$. We say that S_Γ is an augmented spanning set of down-sets if every antecedent and consequent is a down-set.

Although many constraints can be inferred from a base set, encoding essentially performs all the desired inferences *a priori*, and then represents the consequences of an element in a code. Using this code, we can perform operations locally, which amounts to reducing inferences to one step. We shall see in Chapter 5 one approach to relaxing this to allow inferences with a fixed number of steps. How can we perform a one-step inference? Since we associate constraints with elements, we can perform set operations, as we have previously shown. We can also apply one level of modus ponens (i.e. calculate Γ^1 from Γ^0) using coreference, as we describe later.

We must now redefine property preservation for an augmented spanning set S_Γ of down-sets:

Subsumption: $x \leq y$ if and only if $\mathcal{C}_\Gamma(y) \subseteq \mathcal{C}_\Gamma(x)$.

Meets: $x \sqcap y = z$ if and only if $\mathcal{C}_\Gamma(x), \mathcal{C}_\Gamma(y) \vdash \mathcal{C}_\Gamma(z)$.

Joins: $x \sqcup y = z$ if and only if $\mathcal{C}_\Gamma(x) \cap \mathcal{C}_\Gamma(y) = \mathcal{C}_\Gamma(z)$.

Note that when computing meets, we use the constraints to infer additional components. For efficiency, we will usually only perform one inference step. That is, we only infer components from the given components and inferences, and do not attempt further inferences using inferred components (i.e. we compute $\vdash_1$). After performing the inference step of a meet, we can remove trivial constraints (e.g. if we already have s_2, then $s_1 \rightarrow s_2$ is redundant).

Our example preserves subsumption and meets, but not joins. For elements a, b and $\bot$:

i. $\mathcal{C}_\Gamma(a) = \{\downarrow a,\ \downarrow a \wedge \downarrow b \rightarrow \downarrow c,\ \downarrow a \wedge \downarrow c \rightarrow \downarrow b,\ \downarrow b \wedge \downarrow c \rightarrow \downarrow a\}$
ii. $\mathcal{C}_\Gamma(b) = \{\downarrow b,\ \downarrow a \wedge \downarrow b \rightarrow \downarrow c,\ \downarrow a \wedge \downarrow c \rightarrow \downarrow b,\ \downarrow b \wedge \downarrow c \rightarrow \downarrow a\}$
iii. $\mathcal{C}_\Gamma(\bot) = \{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b \rightarrow \downarrow c,\ \downarrow a \wedge \downarrow c \rightarrow \downarrow b,\ \downarrow b \wedge \downarrow c \rightarrow \downarrow a\}$

We can see that $\mathcal{C}_\Gamma(a) \subseteq \mathcal{C}_\Gamma(\bot)$. To compute $a \sqcap b$, we compute the inference $\mathcal{C}_\Gamma(a), \mathcal{C}_\Gamma(b) \vdash \downarrow c$ using the constraint $\downarrow a \wedge \downarrow b \rightarrow \downarrow c$. We can thus infer $\mathcal{C}_\Gamma(\bot)$, and so $a \sqcap b = \bot$. Simplifying the constraints then yields the set $\{\downarrow a, \downarrow b, \downarrow c\}$. Since $\mathcal{C}_\Gamma(\top) = \emptyset$, but $\mathcal{C}_\Gamma(a)$ and $\mathcal{C}_\Gamma(b)$ are not disjoint, this spanning set does not preserve joins.

4.9.3 Integrating spanning sets and constraints

Suppose we have a set of constraints Γ we wish to satisfy and a spanning set S of down-sets that may satisfy some of these constraints. In order to integrate S and Γ, we need to transform Γ so that the antecedents and consequents are expressed in terms of components.

How do we convert elements to components? This can easily be done for any subsumption preserving spanning set S of down-sets. Using the original set Γ of constraints (we assume that $\Gamma \supseteq \Gamma(S) \supseteq \Gamma_\leq$), we construct an augmented spanning set S_Γ. The next theorem shows not only how these conversions can be accomplished, but also proves that it can always be done in a sound and complete manner. For soundness we require: $S_\Gamma \vdash a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$ implies $\Gamma \vdash a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$ and for completeness we require: $\Gamma \vdash a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$ implies $S_\Gamma \vdash a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$. We need to specify how we can infer a constraint on elements from a set of component constraints: $S_\Gamma \vdash a_1 \wedge a_2 \wedge \cdots \wedge a_k \rightarrow b$ if and only if (i) $Q = \bigcup_{1 \leq i \leq k} \mathcal{C}(a_i)$ and (ii) $S_\Gamma \vdash Q \rightarrow s$ for every $s \in \mathcal{C}(b)$. That is, if we can infer every component of the consequent from the components of the antecedents, then we can infer that the antecedents imply the consequent.

Theorem 4.17 *Let L be a lattice, S a spanning set that preserves subsumption and Γ a set of constraints on L of the form $a_1 \wedge a_2 \wedge \ldots \wedge a_k \rightarrow b$ (which contains $\Gamma(S)$). Then the augmented spanning set $S_\Gamma = S \cup \{Q \rightarrow s | A \rightarrow b \in \Gamma, Q = \bigcup_{a \in A} \mathcal{C}(a), s \in \mathcal{C}(b)\}$ is sound and complete.*

Proof: *Soundness:* Suppose $S_\Gamma \vdash A \rightarrow b$ and $Q = \bigcup_{a \in A} \mathcal{C}(a)$. Then $S_\Gamma \vdash Q \rightarrow Q'$, where $Q' \supseteq \mathcal{C}(b)$. Let the sequence of constraints in S_Γ that were used to derive Q' be $Q_1 \rightarrow q_1, \ldots, Q_m \rightarrow q_m$, where $Q \supseteq Q_1$ and $Q' \subseteq Q \cup \{q_1, \ldots, q_m\}$. Each component constraint $Q_i \rightarrow q_i$ must have come from a constraint $A_i \rightarrow b_i \in \Gamma$, where $A \supseteq A_1$. Thus, $\Gamma, A \vdash b_i$, $1 \leq i \leq m$ (i.e. each inference step is justified). Since $\bigcup_{a \in A} \mathcal{C}(a) \cup \bigcup_{b_i \in \{b_1, \ldots, b_m\}} \mathcal{C}(b_i) \supseteq \mathcal{C}(b)$ and S_Γ preserves subsumption, we have $A \wedge b_1 \wedge \ldots \wedge b_m \rightarrow b \in \Gamma(S) \subseteq \Gamma$. Thus, $\Gamma, A \vdash b$.

Completeness: Suppose $\Gamma, A \vdash b$. Let the sequence of constraints in Γ that were used to derive b be $A_1 {\rightarrow} b_1, \ldots, A_m {\rightarrow} b_m$, where $A \supseteq A_1$ and $b = b_m$. For each constraint $A_i {\rightarrow} b_i$, there is a set of constraints in S_Γ: $Q_i {\rightarrow} s$, where $Q_i = \bigcup_{a \in A_i} \mathcal{C}(a)$ and $s \in \mathcal{C}(b_i)$. Thus, we can derive $S_\Gamma, Q \vdash b_i$, where $Q = \bigcup_{a \in A} \mathcal{C}(a)$. □

We can now convert any constraint to a component constraint and the theorem shows that the resulting set will be sound and complete. The resulting constraints can of course be simplified. Constraints with empty consequences, or for which a component appears as both an antecedent and the consequent, can be eliminated. Continuing with our example, if $S = \{\downarrow a,\ \downarrow b,\ \downarrow c\}$ and $\Gamma = \{a {\rightarrow} \top,\ b {\rightarrow} \top,\ c {\rightarrow} \top,\ a \wedge b {\rightarrow} c,\ a \wedge c {\rightarrow} b,\ b \wedge c {\rightarrow} a,\ \bot {\rightarrow} a,\ \bot {\rightarrow} b,\ \bot {\rightarrow} c\}$, then the augmented spanning set is: $S_\Gamma = \{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b {\rightarrow} \downarrow c,\ \downarrow a \wedge \downarrow c {\rightarrow} \downarrow b,\ \downarrow b \wedge \downarrow c {\rightarrow} \downarrow a\}$. We can achieve a further reduction in this example, and still maintain order and meets, by eliminating the components containing $\downarrow a$ in their consequents. This results in the augmented spanning set $S'_\Gamma = \{\downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b {\rightarrow} \downarrow c,\ \downarrow a \wedge \downarrow c {\rightarrow} \downarrow b\}$. Although it may be difficult to determine a minimal augmented spanning set, approximation algorithms may be developed.

Our analysis above did not consider negated elements resulting from meet incompatibility constraints. For this, we require the notion of a negated component $\neg s$, which represents a logical barrier to the inference of a component s (i.e. $s \wedge \neg s$ is inconsistent). The constraint $a_1 \wedge a_2 \wedge \cdots \wedge a_k {\rightarrow} \neg b$, can be replaced by $\mathcal{C}(a_1) \cup \mathcal{C}(a_2) \cup \cdots \cup \mathcal{C}(a_k) {\rightarrow} \neg s$ provided: (i) $s \in \mathcal{C}(b)$ and (ii) $\forall$ factors f of s, we have $a_1 \wedge a_2 \wedge \cdots \wedge a_k {\rightarrow} \neg f$. Thus, we can replace a negated element by the negation of one of its components provided the antecedents imply the negation of every factor. This is required because incompatibility will be detected by inference failure and we need to be certain that all failures are justified. We can always accomplish this if the negated element is the factor of a principal down-set component. If no component satisfies this constraint, we can add this principal down-set to the spanning set. We later show how coreference and decomposition can be used to implement these constraints.

As an example, a spanning set for the cube lattice in Figure 4.5 is $S_\mathcal{M} = \{\downarrow a, \downarrow b, \downarrow c\}$ and the meet incompatibility constraints are $\{a \sqcap f {=} \bot, b \sqcap e {=} \bot, c \sqcap d {=} \bot\}$. The augmented spanning set is $S_\Gamma = \{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b {\rightarrow} \neg \downarrow c,\ \downarrow a \wedge \downarrow c {\rightarrow} \neg \downarrow b,\ \downarrow b \wedge \downarrow c {\rightarrow} \neg \downarrow a\}$. To take the meet $\sqcap\{a, b, c\}$, we first obtain the entire set above, from which we can derive $\downarrow a, \downarrow b, \downarrow c, \downarrow a \wedge \downarrow b {\rightarrow} \neg \downarrow c \vdash \downarrow c, \neg \downarrow c$, which is inconsistent. We can again reduce the number of components in the augmented spanning set, while still maintaining meets: $S'_\Gamma = \{\downarrow b,\ \downarrow c,\ \downarrow a \wedge \downarrow b {\rightarrow} \neg \downarrow c,\ \downarrow a \wedge \downarrow c {\rightarrow} \neg \downarrow b\}$.

4.9.4 Guarded constraints

Although constraints are global, for efficiency we want to selectively associate constraints with elements. We must do this in a way that ensures satisfaction, yet minimizes the number of constraints associated with each element. A constraint could be affiliated with each of its antecedents and its consequent, but to ensure satisfaction only one antecedent, or the consequent in case there are no antecedents, needs to be linked to it (since the antecedents are conjunctive). This leads to the notion of guarded constraints, which are analogous to Dijkstra's guarded commands.

Definition 4.11 *Let P be an ordered set. A* guarded *constraint for P is a constraint of the form $a : A {\rightarrow} b$, where $A \wedge a {\rightarrow} b$ is a constraint in P[17]. For any element $a \in P$, $a : a$ is a trivial guarded constraint.*

The set of guarded constraints obtained from Γ is denoted as Γ^G. A constraint with k antecedents may result in up to k guarded constraints, but we may not need to retain all of these: it may be possible to eliminate up to $k-1$ of the constraints, although we shall see that this cannot be done arbitrarily. In the diamond lattice example (Figure 4.1), we can guard the meet constraints and still maintain meets as follows: $\Gamma^G = \{a{:}b {\rightarrow} \bot,\ a{:}c {\rightarrow} \bot,\ c{:}b {\rightarrow} \bot\}$.

Modus ponens can be revised to operate on guarded constraints: $a, A, (a{:}A {\rightarrow} b) \vdash b$. Given a starting set of constraints Γ, constraint inference becomes:

i. $\Gamma^0 = \{a{:}A {\rightarrow} b \mid \Gamma, a, A \vdash_1 b\ (\supseteq \Gamma)$
ii. $\Gamma^{i+1} = \{a{:}A {\rightarrow} b \mid (\Gamma^i, a, A \vdash_1 c_1), \cdots, (\Gamma^i, a, A \vdash_1 c_k)$ and $\Gamma^i, c_1, \cdots c_k \vdash_1 b\}$

For encoding, we will guard the constraints in augmented spanning sets. Thus S^G_Γ will be a set of guarded component constraints from S_Γ. We guard an elementary component s as $s{:}s$ (if we write s, this is assuming the implicit

[17] If $A = \emptyset$, we write $a : b$.

form $s{:}s$). The component mapping is modified as follows: $\mathcal{C}_\Gamma(x) = \{s_1 \wedge \ldots \wedge s_{k-1} {\rightarrow} s_k | s^g : s_1 \wedge \ldots \wedge s_{k-1} {\rightarrow} s_k \in S^G_\Gamma, x \in s^g\}$.

Taxonomic operations are performed as before. The reason that we don't include the guard in the result of the *augmented component mapping* is that the guard indicates to which elements a constraint (or *augmented component*) is associated, and the rest of the constraint is conditional on the context of the guard (analogous to conditional probability). Also, in order to implement augmented spanning sets, we require that, for every component of the form $s^g{:}s_1 \wedge \ldots \wedge s_{k-1} {\rightarrow} s_k$, there are elementary components $s_i{:}s_i$ for $1 \leq i \leq k$. Thus, down-sets involved in constraints (but not necessarily guards) must be present as elementary components. We show later how this property can be used to reduce encoding size.

In our example, $S^G_\Gamma = \{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow a{:}\downarrow b {\rightarrow} \downarrow c,\ \downarrow a{:}\downarrow c {\rightarrow} \downarrow b,\ \downarrow b{:}\downarrow c {\rightarrow} \downarrow a\}$. Meets, and now also joins, are preserved. We can also reduce this spanning set to $S'^G_\Gamma = \{\downarrow b,\ \downarrow c,\ \downarrow a{:}\downarrow b {\rightarrow} \downarrow c,\ \downarrow a{:}\downarrow c {\rightarrow} \downarrow b\}$.

4.9.5 Coreference

Logical terms provide coreference through named variables or labels. Two or more positions in a term that corefer must hold identical values, called a coreference constraint. If one is instantiated, then all are identically instantiated. We can characterize coreference as *persistent* or *transient*. Once a coreference point is instantiated, transient coreference disappears (i.e. there is no recollection of the coreferring positions). This is the form provided by Prolog. Although implementations may retain the coreference constraint to reduce storage requirements, the surface form is transient. Persistent coreference, as provided by LIFE [4], maintains the coreference after instantiation.

More generally, coreference is an equivalence relation within a term. That is coreference is (i) symmetric: if it is used to implement $a {\rightarrow} b$, then it also implements $b {\rightarrow} a$, and (ii) transitive (since we can only have one coreference label or variable at a position in a term): if we implement $a {\rightarrow} b$ and $b {\rightarrow} c$, then we are also implementing $a {\rightarrow} c$. By introducing coreference within a specific term, we implement a guarded equivalence relation. For example, if we use coreference to implement the guarded constraint $s{:}s_1 {\rightarrow} s_2$, then the equivalence class $s_1 \leftrightarrow s_2$ is implemented for elements in s. Meet incompatibility constraints (e.g. $s{:}s_1 {\rightarrow} \neg s_2$) require the use of symbols, as discussed in the next subsection. If we can decompose our meet inferences into guarded equivalence classes, we can implement an augmented spanning set using coreference in logical terms, as formalized below.

Theorem 4.18 *Let L be a lattice, and S^G_Γ be a guarded augmented spanning set on L that contains no negated components (i.e. no meet incompatibility constraints). Then there is a logical term implementation (which may use coreference) of S^G_Γ if and only if*

i. If $s^g{:}s_1 \wedge s_2 \wedge \cdots \wedge s_k {\rightarrow} s \in S^G_\Gamma$ then $k \leq 1$
ii. If $S^G_\Gamma \vdash s^g{:}s_1 {\rightarrow} s_2$ then $S^G_\Gamma \vdash s^g{:}s_2 {\rightarrow} s_1$
iii. If $S^G_\Gamma \vdash s^g{:}s_1 {\rightarrow} s_2$ and $S^G_\Gamma \vdash s^g{:}s_2 {\rightarrow} s_3$ then $S^G_\Gamma \vdash s^g{:}s_1 {\rightarrow} s_3$

The proof of this theorem follows from the fact that coreference cannot itself be conditional (condition (i)) and it imposes a set of equivalence classes (conditions (ii) and (iii)). Condition (iii) is actually unnecessary, since it follows from inference. It is possible to take any constraint with more than one antecedent and split it into a number of constraints with two antecedents each. For the constraint $a_1 \wedge a_2 \wedge \cdots \wedge a_k {\rightarrow} b$, we can create $k-2$ additional elements $l_{2,3}, l_{3,4}, \cdots l_{k-1,k}$ and rewrite the constraint as: $a_1 \wedge a_2 {\rightarrow} l_{2,3},\ l_{2,3} \wedge a_3 {\rightarrow} l_{3,4}, \cdots, l_{k-1,k} \wedge a_k {\rightarrow} b$.

Logical terms can be used to implement augmented spanning sets that satisfy the above restrictions. A coreference equivalence class will be introduced by its guard by placing a new variable in the positions assigned to each of the coreferring components. We may also be able to implement coreference using integer vectors equipped with pointers. For non-decomposed spanning sets, we can use the same symbol (e.g. 1) for all components, or just record the presence of the component without a symbol (as is possible with sparse terms). We describe additional restrictions for decomposed spanning sets and meet incompatibility constraints in the next subsection. Using coreference, we can implement our example spanning set for the diamond lattice as shown in the first diagram in Figure 4.20. Meets are preserved with unification and joins with anti-unification.

There are lattices for which we cannot preserve both meets and joins with augmented spanning sets of down-sets. As indicated, problems arise when we cannot establish symmetry or transitivity of constraints. Figure 4.21 shows

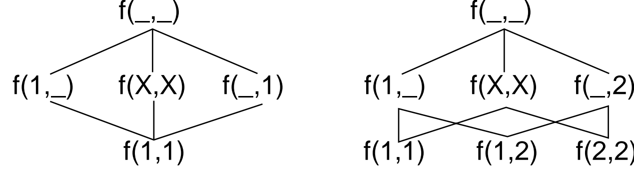

Figure 4.20: Term encoding for diamond and cube lattices

such a lattice. If we are to preserve joins, the component mappings for each of a, b, c must be disjoint. Thus all the down-sets must be principal, and in particular the guards must be principal down-sets. We must preserve the constraint $a \wedge c \rightarrow b$, but neither $a \wedge b \rightarrow c$ nor $b \wedge c \rightarrow a$ holds, so there is no way to guard this constraint for implementation with coreference.

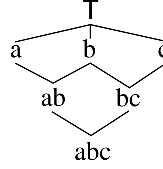

Figure 4.21: Lattice for which no augmented spanning set of down-sets can preserve meets and joins

4.9.6 Coreference, decomposition and meet incompatibility constraints

Decomposition in augmented spanning sets only applies to elementary components (components of the form $s{:}s$); the other components will be implemented as constraints between these. Meet-incompatible decomposition, in addition to reducing space requirements, permits some meet incompatibility to be detected by union failure, and represents incompatibility constraints among pairs of elements. However, when the meet of three or more elements is $\perp$, but every pair is compatible, we cannot ensure incompatibility-as-failure using decomposition alone [102].

Since coreference imposes equality constraints between positions within terms, and each subset in a decomposition is assigned a position within a term, we define a partial function *symbol* that maps elementary component/subset pairs to the symbol used to represent the elementary component within the subset. Thus $symbol(s, \alpha)$, for $s \in \alpha \subseteq S$, returns the symbol used to discriminate component s from other components within subset α. We specify the subset since components may be in multiple subsets. For non-decomposed spanning sets with no meet incompatibility constraints this was unnecessary, as every component could be assigned the same symbol. For a chain decomposition, the symbols must be ordered according to the chain order. For meet homogeneous decomposition, we must have a $\perp_\alpha$ symbol to put in the position of α for elements in every component, but otherwise the restrictions are similar to those for meet incompatible decomposition. We do not consider these cases further. Integrating coreference with meet incompatible decomposition of spanning sets requires different restrictions than in Theorem 4.18:

Theorem 4.19 *Let L be a lattice, S^G_Γ be a guarded augmented spanning set on L, and $\mathcal{A} = \{\alpha_1, \ldots, \alpha_k\}$ be a meet incompatible decomposition of the elementary components of S^G_Γ. Then there is a logical term implementation (which may use coreference) of S^G_Γ if and only if (i) $\forall\ s^g{:}s_1 \wedge s_2 \wedge \cdots \wedge s_k \rightarrow s \in S^G_\Gamma$, $k \leq 1$ and (ii) $\exists$ guarded equivalence relations, $=_{s^g} \subseteq \mathcal{A} \times \mathcal{A}$ for each guard s^g in S^G_Γ, and a symbol mapping that satisfy:*

i. $\forall \alpha \in \mathcal{A}$, *if* $s_1, s_2 \in \alpha$ *and* $s_1 \neq s_2$ *then* $symbol(s_1, \alpha) \neq symbol(s_2, \alpha)$

ii. *If* $s^g{:}s_1 \rightarrow s_2 \in S^G_\Gamma$ *then* $\forall \alpha_2 \in \mathcal{A}$ *for which* $s_2 \in \alpha_2$, $\exists \alpha_1 \in \mathcal{A}$ *for which* $s_1 \in \alpha_1$ *and* $symbol(s_1, \alpha_1) = symbol(s_2, \alpha_2)$ *and* $\alpha_1 =_{s^g} \alpha_2$

iii. *If* $s^g{:}s_1 \rightarrow \neg s_2 \in S^G_\Gamma$ *then* $\exists \alpha_1, \alpha_2 \in \mathcal{A}$ *for which* $s_1 \in \alpha_1, s_2 \in \alpha_2$ *and* $symbol(s_1, \alpha_1) \neq symbol(s_2, \alpha_2)$ *and* $\alpha_1 =_{s^g} \alpha_2$

iv. *If neither* $s^g{:}s_1 \rightarrow s_2 \in S^G_\Gamma$ *nor* $s^g{:}s_1 \rightarrow \neg s_2 \in S^G_\Gamma$, *then*

iv.1. $\forall \alpha_1, \alpha_2 \in \mathcal{A}$ *such that* $s_1 \in \alpha_1$ *and* $s_2 \in \alpha_2$, *if* $symbol(s_1, \alpha_1) \neq symbol(s_2, \alpha_2)$ *then* $\neg(\alpha_1 =_{s^g} \alpha_2)$

iv.2. $\exists \alpha_1, \alpha_2 \in \mathcal{A}$ *such that* $s_1 \in \alpha_1$ *and* $s_2 \in \alpha_2$, *and* $\neg(\alpha_1 =_{s^g} \alpha_2)$

For every constraint $s^g\!:\!s_1 \rightarrow s_2$ we need to establish coreference between each subset containing s_2 and some subset containing s_1. For every constraint $s^g\!:\!s_1 \rightarrow \neg s_2$, we need to establish coreference between one subset containing s_1 and one containing s_2. By ensuring equality or inequality of the symbols, we can satisfy the constraint in the context of s^g. In the former case, we will infer s_2 given s_1; in the latter case unification will fail if we have both s_1 and s_2. Thus, provisions (i), (ii) and (iii) are necessary conditions for implementation of the non trivial constraints in S^G_Γ with coreference. Since coreference forms a guarded equivalence relation among subsets of the decomposition $\mathcal{A}$, not among components, the establishment of coreference constraints must be consistent with other constraints pertaining to the coreferring subsets. Provision (iv) ensures that no unsupported inferences are made. Note that the above conditions can be used when attempting to satisfy meet and meet incompatibility constraints even if our spanning set is not decomposed by giving it the trivial decomposition that puts each component in its own subset.

Given a satisfying set of guarded coreference relations and symbol mapping, we can easily construct the terms as before. Each subset will have a position, as discussed in Section 4.8. When computing the term for an element x, start with the inherited term (i.e. the unification of the parent terms). For each subset α for which x is a factor of a component $s \in \alpha$, put $symbol(s, \alpha)$ in the position for α. For each guard s^g for which x is a factor of s^g, add coreference between all positions α_1, α_2 for which $\alpha_1 =_{s^g} \alpha_2$.

As an example, consider again the lattice in Figure 4.5. Using principal down-sets, we can derive the augmented spanning set $S^G_\Gamma = \{\downarrow a,\ \downarrow b,\ \downarrow c,\ \downarrow b\!:\!\downarrow a \rightarrow \neg\downarrow c,\ \downarrow b\!:\!\downarrow c \rightarrow \neg\downarrow a\}$. As in the previous case, we can notice that the elementary component $\downarrow b$ is unnecessary, so a reduced spanning set is $S1^G_\Gamma = \{\downarrow a,\ \downarrow c,\ \downarrow b\!:\!\downarrow a \rightarrow \neg\downarrow c,\ \downarrow b\!:\!\downarrow c \rightarrow \neg\downarrow a\}$. We can now give the trivial meet-incompatible decomposition, and define the symbol mapping as follows: $symbol(\downarrow a, \{\downarrow a\}) = 1$ $symbol(\downarrow c, \{\downarrow c\}) = 2$. Since the constraints guarded by $\downarrow b$ are equivalent, we can easily implement this spanning set, as shown in the second diagram in Figure 4.20.

4.9.7 Encoding algorithms

In [102] is an exploration of which forms of ordered sets can be encoded using logical terms so that meets are preserved with union (i.e. unification) and meet incompatibility is detected with failure. In [104] this exploration is extended to general DAGs.

Our exploration of the use of constraints and coreference takes a different approach. Mellish fixes on an implementation (e.g. terms or DAGs) and attempts to find the class of ordered sets that can be encoded to preserve $\Gamma_\sqcap$ and $\Gamma_\perp$. In contrast, we take the ordered set P to encode and the constraints to satisfy as input that we cannot control. Our goal is to develop a variety of tools with which we can efficiently encode P regardless of its form (although we assume that P is finite, and Mellish does not). In the above two papers, the form of *encodable* ordered sets is explored, but no encoding algorithms are presented. The only encoding algorithm that exploits coreference that we are aware of is the *brute force* algorithm in [101]. Unfortunately, this algorithm may potentially produce terms that are of exponential size compared to the size of the ordered set to encode.

We have not given any encoding algorithm, although a naive one may be specified:

i. Start with the constraints to satisfy (e.g. a subset of $\Gamma_\leq \cup \Gamma_\sqcap \cup \Gamma_\perp$).
ii. Derive an augmented spanning set S^G_Γ that satisfies these constraints (e.g. the principal down-sets for meet-irreducible elements satisfy this).
iii. Form a meet-incompatible decomposition of the elementary components.
iv. Form guarded coreference relations and a symbol mapping that satisfy as many of the constraints as possible, while obeying provisions (i) and (iv) of Theorem 4.19.
v. Derive the logical term for each element using the component and symbol mappings, and the guarded coreference relations.

Recall that finding a minimal meet-incompatible decomposition is NP-Hard. Thus, it seems likely that encoding algorithms that exploit logical terms and coreference will be approximation algorithms. The above high-level algorithm will find a term encoding that approximates the optimal in terms of space requirements and properties satisfied. An area for future research is to design specific algorithms for term encoding.

4.9.8 Variations

In order to enhance implementations of augmented spanning sets, there are several avenues worth considering. The first involves the preservation of joins. Given a spanning set $S_{\mathcal{M}}$, which preserves joins, when we augment this with constraints we may lose joins because of constraints that are associated with each element of the join, but not with the result. This problem can be avoided by redefining joins to consider only the elementary components.

Although coreference provides an efficient and available implementation of certain forms of constraints, its nature restricts its usage. Since logical inference is transitive, this is a desirable property to implement constraints. Symmetry, on the other hand is not always desired; it does not always hold in a set of constraints. What we require is a way to implement arbitrary guarded constraints. One approach would be to use a constraint logic programming language. This is viable only if the language efficiently implements such constraints. Another possibility is to use a "trigger" mechanism that invokes a constraint when the antecedents are satisfied, but ignores it otherwise. Coreference essentially allows the consequence to trigger the constraint as well as the antecedent. This functionality is developed as *reference constraints* in Chapter 8.

4.10 Discussion and Conclusion

In this chapter, we have characterized encodings as implementations of spanning sets that preserve subsumption and possibly meets and/or joins. We have thus provided a framework in which to compare all approaches to encoding. Although implementations may have a drastic effect on the size and efficiency of encodings, we can abstract the fundamental aspects of a technique to the level of spanning sets.

Throughout our analysis, we classified current encoding techniques within this structure. We showed how the transitive closure and compact encodings in [2], the tree encoding in [77] and a simplified version of a tree term encoding defined in [102] are all implementations (or equivalent to implementations) of spanning sets of principal down-sets or up-sets. The compact hierarchical encoding of [24] implements a spanning set of compound down-sets, which we showed to be an approximation to the NP-Hard optimum. The integer vector encoding of [97] employs chain partitioning. More complex term encodings described in [102] arise from meet incompatible decomposition and coreference constraints induced by logical variables. Table 4.1 summarizes our characterization of these encoding schemes in terms of the operations satisfied, the types of components in the spanning set, whether decomposition is utilized and the implementation of the spanning set. For comparison, we characterize schemes using spanning sets of down-sets, which may be the dual of the actual algorithm described. As can be seen, there are many possibilities open for exploration.

In many of the our inquiries, the complexity of the problem has left open many avenues for continued research. The NP-Hard results for minimal spanning sets of compound down-sets and meet incompatible decomposition warrant further exploration for approximation algorithms. In particular, we have indicated the utility of multiple occurrences of factors in compound down-sets, offering the potential for finding approximation algorithms resulting in more efficient subsumption encodings than in [24, 61, 79]. Another area justifying more research is in the specification and implementation of constraint-based spanning sets. Coreference provides a logical implementation for certain forms of constraints. Mellish [101] provides a brute force method for encoding any finite taxonomy using coreference.

A key factor affecting the design of encoding algorithms is whether the ordered set is dynamic or static (i.e. the degree to which the ordered set may change during run-time). The encoding of a static order can be computed *a priori*. In this case, the speed of the encoding algorithm, and the feasibility of modifying codes is not as important as the efficiency of the codes. For dynamic orders, however, we need encoding schemes that efficiently generate encodings and are not brittle in the face of change. In this case, the modifications required for codes should be local to the change in the ordered set and should not take too long to update. Of course the underlying spanning set will have a great impact on the scope of a change. Compound components and decomposition both magnify the number of elements directly effected. Implementations also have a significant effect on scope. Those which require every element to be of the same length (e.g. bit-vectors and integer vectors), or which require the specification of unfilled positions (e.g. bit-vectors and ordinary logical terms), cause the scope of change to extend beyond those elements directly affected. For the interval encoding in [1], the authors describe how leaving *gaps* between different intervals

Table 4.1: Characterization of encoding schemes in terms of spanning set of down-sets

	type of encoding	spanning set components	decomposition	implementation
transitive closure [2]	join	principal S_1	-	bit vector
compact [2]	join	principal $S_{\mathcal{M}}$	-	bit vector
interval [1]	join	principal S_1	-	integer intervals
virtual time [97]	join	principal S_1	chain	integer vector
tree encoding [77]	meet	prime $S_{\overline{\mathcal{T}}}$	-	bit vector
tree term [102] [102]	meet	prime $S_{\overline{\mathcal{T}}}$	meet incompatible	tree term
term [102]	meet	pseudo-prime[18]	meet incompatible	logical term
compact hierarchical [24]	subsumption	compound	-	bit vector
indexed [61]	join	compound	-	bit vector
co-atomic tree encoding [79]	subsumption	compound	-	bit vector

can reduce the cost of updates (both inserts and deletes). As these gaps fill, it may become necessary to re-encode the ordered set. We argue in Chapter 6 that sparse terms may offer the flexibility required of dynamic environments.

One of the contributions of our analysis is that it may guide the development of new encoding schemes. A given encoding problem may dictate certain constraints, such as structural properties of the ordered sets to encode (e.g. lattice, distributive, bounded width), operations required (order checking, meets, joins), if the order changes dynamically and how (does it grow top-down? are the changes frequent?), and so on. The application and available hardware may also suggest an implementation (e.g. parallel hardware may preclude the use of coreference). The problem parameters will constrain the available techniques and may indicate the availability or absence of existing algorithms to solve the problem. In the latter case, some of our results may assist in the development of new algorithms.

There are several important topics that we did not cover in this chapter. We did not discuss in detail the problem of *decoding* the result of a meet or join operation to obtain the element(s) in the original order. The importance of this depends on the application. Some applications (e.g. [2]) only need to perform a decode operation after many meet operations, and so the efficiency of decoding is less significant. Other applications, however, may need to decode after every operation. There are several options to decode efficiently. Efficient algorithms have been proposed in [61, 77, 78, 114]. The composition of sparse terms may be exploited in decoding. Depending on the implementation, hashing may also be possible. Another area we ignored is *relative complements*, which involves the association of negative, as well as positive, information with elements. We hypothesize that the technique in [2] can be formalized in terms of spanning sets and integrated with the techniques we have discussed.

We have proposed spanning sets as a foundational framework in which taxonomic encoding techniques can be classified. Our analysis exposes connections among existing schemes in terms of the information content of the resulting encodings and the implementation techniques employed. We have also shown some of the limits of encoding, especially our NP-Hardness results. The classification also reveals several avenues for continued research on encoding, particularly for algorithms to approximate the NP-Hard problems (e.g. sections 4.7 and 4.8) and for exploration of

[18] See [102] for a description of pseudo-prime spanning sets.

some of the generalizations and extensions that we have proposed. Additional exploration of the use of constraints (such as coreference constraints provided by logical variables) is also warranted.

We feel that this work provides an important view on the field of taxonomic encoding, summarizing current efforts and giving direction for its continuing development. It is one step forward in the quest for efficiency in taxonomic reasoning.

Chapter 5

Modulated Encoding

"Thinking is sometimes injurious to health"

– Aristotle

In the previous chapter, we considered encoding ordered sets in their entirety. Using the techniques presented, many efficiency gains can be realized. However, if we could decompose our ordered set P into a number of smaller units, dramatic decreases in space may be achieved[1].

In this chapter, we examine ordered sets in terms of intervals. A special type of interval, called a *module*, leads to an efficient form of order partitioning called *modulation* [2] where each partition can be encoded, or further modulated, independently. This allows us to synthesize, with little overhead, different approaches to encoding, by taking advantage of the most efficient techniques for portions of a taxonomy.

Modulation is related to modular decomposition of graphs, particularly comparability graphs [90, 109, 112][2]. Another form of partitioning for distributive lattices is described in [78]. We present a flexible scheme to perform lattice operations on modulated taxonomies, and also lay some groundwork for generalizing modulation. This chapter extends our research in [49], and provides correctness proofs for operations in modulated taxonomies.

5.1 Order Intervals and Modules

Definition 5.1 *Let P be an ordered set. A* closed interval*, denoted as* $[\{a_1, \cdots, a_m\}, \{b_1, \cdots, b_n\}]$*, is a set of elements* $\{x \in P | \exists a_i, b_j \text{ such that } a_i \leq x \leq b_j\}$*.*

We can alternatively define a closed interval as the intersection of a down-set and an up-set: $[\{a_1, \cdots, a_m\}, \{b_1, \cdots, b_k\}] = \uparrow\{a_1, \cdots, a_m\} \cap \downarrow\{b_1, \cdots, b_k\}$. Intervals in ordered sets are analogous to intervals in total orders, such as the integers, and are also known as *convex suborders*. Open and half-open intervals can be similarly defined using non-inclusive subsumption. If $m = 1$ and $n = 1$, then the interval is called *principal*; otherwise it is *compound*. A *canonical* principal interval $[a, b]$ requires $a \leq b$ and represents a unique, non-empty set of elements[3]. If $A, B \subseteq P$ then the compound interval $[A, B]$ can be defined as a union of principal intervals: $[A, B] = \bigcup_{a \in A, b \in B}[a, b]$. The notation for a compound interval must not contain any redundant information: $[A, B]$ is canonical if A and B are anti-chains, and $\forall a \in A, \exists b \in B, a \leq b$ and dually. This ensures that non-empty intervals are uniquely represented with this notation.

Since intervals are a restricted type of subset, a spanning set of intervals is simply a set of intervals for which the component mapping is one-to-one. Rather than using spanning sets of intervals directly, however, we will employ certain forms of intervals to partition the ordered set into more manageable pieces that can then be encoded using approaches described previously. Down and up-sets in these segments correspond to intervals in the original ordered set.

[1] The decomposition techniques described in section 4.8 of Chapter 4 are designed to decompose spanning sets to improve the space efficiency of implementation whereas in this chapter, decomposition is a meta-level technique for subdividing an order to encode into two or more smaller orders.

[2] A graph G_P is the comparability graph of an ordered set P if $G_P = (P, E)$ and $(x, y) \in E$ if and only if $x < y$ or $y < x$.

[3] If $a \leq b$ does not hold, then $[a, b] = \emptyset$.

Intervals are related by two partial orders: containment and subsumption. Since intervals represent subsets of a lattice L, they can be related by set containment: $[a,b] \subseteq [c,d]$ if and only if $c \leq a$ and $b \leq d$ and $[A,B] \subseteq [C,D]$ if and only if $\forall a \in A, b \in B, \exists c \in C, d \in D$ where $c \leq a$ and $b \leq d$. The subsumption ordering on L can also specify subsumption on intervals. We first define (absolute) subsumption: $[a,b] \leq [c,d]$ if and only if $b \leq c$, which is equivalent to: $\forall x \in [a,b], \forall y \in [c,d], x \leq y$. For compound intervals, $[A,B] \leq [C,D]$ if and only if $\forall b \in B, \forall c \in C, b \leq c$. We now define *partial* subsumption among intervals: $[a,b] \leq [c,d]$ if and only if $\forall x \in [a,b], \exists y \in [c,d], x \leq y$. This is equivalent to: $b \leq d$. Absolute subsumption and interval containment can both be seen as special cases of this.

We are particularly interested in certain forms of intervals that permit us to partition an ordered set without incurring a loss of information or unreasonable additional cost to maintain order. Our analysis formalizes and extends an earlier proposal in [2].

Definition 5.2 *Let P be an ordered set, $a \in P$ and $Q \subseteq P$. A* surrogate *for a in Q is an element $b \in P$ for which $\forall x \in Q$, (i) $a \leq x$ if and only if $b \leq x$ and (ii) $a \geq x$ if and only if $b \geq x$.*

An element $b \in P$ that satisfies only the first (second) condition is called an upper (lower) surrogate for a in Q. Also, if a is a surrogate for b in Q, then b must also be a surrogate for a in Q.

Definition 5.3 *Let P be an ordered set. A subset $M \subseteq P$ is called a* module *if $\forall x, y \in M$, x is a surrogate for y in $P \backslash M$.*

That is, $\forall x, y \in M$ and $z \in P \backslash M$, $x \leq z$ if and only if $y \leq z$, and $x \geq z$ if and only if $y \geq z$. Modules are also called *order autonomous sets* [90], and the sets in the comparability graph G_P that correspond to modules are called *modules*, *stable sets*, or *clumps* [109]. We now state some properties of modules.

Theorem 5.1 *Let L be a lattice. Then $M \subseteq L$ is a module if and only if $\sqcup M = b$ is an upper surrogate and $\sqcap M = a$ is a lower surrogate for M in $L \backslash M$.*

Proof: $\Rightarrow$ Suppose M is a module. Let $b = \sqcup M$ and $a = \sqcap M$. Also, let $z \in L \backslash M$ and $x \in M$. If $z \geq b$ then $z \geq x$ (by the definition of join). If $z \geq x$ then $z \geq y$ for all $y \in M$ (by the definition of a module). Then $z \geq b$ (by the definition of join). Thus b is an upper surrogate for M in $L \backslash M$. An analogous proof can show that a is a lower surrogate for M in $L \backslash M$.

$\Leftarrow$ Suppose b is an upper surrogate, and a is a lower surrogate, for M in $L \backslash M$. Consider any $z \in L \backslash M$ and $x, y \in M$. $z \geq x$ if and only if $z \geq b$ if and only if $z \geq y$, and $z \leq x$ if and only if $z \leq a$ if and only if $z \leq y$. Thus every pair of elements in M are surrogates in $L \backslash M$, so M is a module. □

Corollary 5.1 *Let L be a lattice. If a subset $M \subseteq L$ is a module, then*

i. There are no elements between the maximal (minimal) elements of M and the join (meet) of M: $[\lceil M \rceil, \sqcup M] = \emptyset$ and $[\sqcap M, \lfloor M \rfloor] = \emptyset$,

ii. M is a closed interval: $M = [\lfloor M \rfloor, \lceil M \rceil]$ and

iii. The only arcs entering (leaving) M are through the maximal (minimal) elements of M: $M = \downarrow \lceil M \rceil \backslash \downarrow \lfloor M \rfloor \cup \lfloor M \rfloor = \uparrow \lfloor M \rfloor \backslash \uparrow \lceil M \rceil \cup \lceil M \rceil$.

Proof: (i) Suppose $\exists z \in L \backslash M$ for which $z < \sqcup M$ and $z > y$ for some $y \in \lceil M \rceil$. But then $\sqcup M$ is not an upper surrogate for M in $L \backslash M$. By the above theorem, M cannot be a module.

(ii) If $z \in M$, clearly $z \in [\lfloor M \rfloor, \lceil M \rceil]$. Let $z \in [\lfloor M \rfloor, \lceil M \rceil]$. Then $\exists a \in \lceil M \rceil, b \in \lfloor M \rfloor$ such that $b < z < a$. Suppose $z \notin M$. Then a and b cannot be surrogates for each other with respect to z.

(iii) Suppose M is a module. By (ii) above, $M = [\lfloor M \rfloor, \lceil M \rceil]$. Let $x \in M$. Clearly $x \in \downarrow \lceil M \rceil$. $x \in \downarrow \lfloor M \rfloor$ if and only if $x \in \lfloor M \rfloor$. In either case, $x \in \downarrow \lceil M \rceil \backslash \downarrow \lfloor M \rfloor \cup \lfloor M \rfloor$. Let $x \in \downarrow \lceil M \rceil \backslash \downarrow \lfloor M \rfloor \cup \lfloor M \rfloor$. Then either $x \in \downarrow \lceil M \rceil$ and $x \notin \downarrow \lfloor M \rfloor$ or $x \in \lfloor M \rfloor$. In the latter case, $x \in M$. For the former case, $\exists a \in \lceil M \rceil$ for which $x \leq a$ and $\forall b \in \lfloor M \rfloor$, $x \not\leq b$. If $x \notin M$ then M cannot be a module. We can analogously show that $M = \uparrow \lfloor M \rfloor \backslash \uparrow \lceil M \rceil \cup \lceil M \rceil$. □

This corollary shows that a module is a special type of interval (item (ii) above). The general forms of a module are shown in the following figure. In the first and third b is a surrogate. In the first and second, a is a surrogate. In all cases, a is a lower surrogate and b is an upper surrogate.

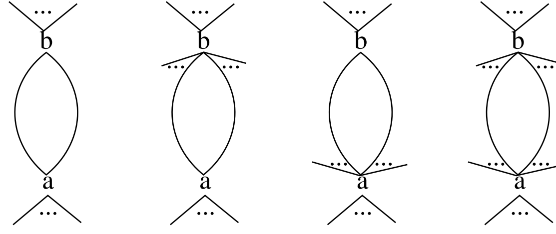

Figure 5.1: Types of modules

5.2 Order partitions

Definition 5.4 *Let P be an ordered set. An* order partition *is a partition of P into two suborders Q and $P\backslash Q$.*

A partition basically loses the subsumption information between elements in Q and $P\backslash Q$. We say that $Q \subseteq P$ induces the partition Q and $P\backslash Q$.

Definition 5.5 *A partition of an ordered set P into Q and $P\backslash Q$* preserves subsumption *if $\exists a, b \in P\backslash Q$ such that a is a lower surrogate, and b an upper surrogate, for Q in $P\backslash Q$.*

Theorem 5.2 *Let L be a lattice and $Q \subseteq L$. Then Q is a module if and only if the partition induced by $Q\backslash\{\sqcup Q, \sqcap Q\}$ preserves subsumption.*

Proof: $\Rightarrow$ Suppose Q is a module. Let $b = \sqcup Q$ and $a = \sqcap Q$. By a previous theorem, b is an upper surrogate, and a a lower surrogate, for Q in $P\backslash Q$. Since $Q\backslash\{\sqcup Q, \sqcap Q\} \subseteq Q$ and $a, b \notin Q\backslash\{\sqcup Q, \sqcap Q\}$, this partition is subsumption preserving.

$\Leftarrow$ Suppose the partition induced by $Q\backslash\{\sqcup Q, \sqcap Q\}$ preserves subsumption.
Then $\exists c, d \in P\backslash(Q\backslash\{\sqcup Q, \sqcap Q\})$ such that c is an upper surrogate, and d a lower surrogate, for Q in $P\backslash(Q\backslash\{\sqcup Q, \sqcap Q\})$. Let $x, y \in Q$ and $z \in P\backslash Q$. Then $x \leq z$ if and only if $c \leq z$ if and only if $y \leq z$ and $x \geq z$ if and only if $d \geq z$ if and only if $y \geq z$. Thus, Q is a module. □

Note that $\sqcup Q$ and $\sqcap Q$ need not be in Q. Both are in Q only for principal modules. In this case, only one of these need be left behind in the partitioning.

Theorem 5.3 *Let L be a lattice and let Q be a module in L. Then the decomposition of L into Q and $L\backslash Q \cup \{\sqcup Q, \sqcap Q\}$ produces two lattices.*

Proof: Clearly Q is a sub-lattice (i.e. it is closed under meets and joins). Consider the meet of any two elements in $L\backslash Q \cup \{\sqcup Q, \sqcap Q\}$: $x \sqcap y$. The only way $x \sqcap y$ could be in Q is if $x \sqcap y = \sqcup Q$, otherwise Q is not a module. □

5.3 Modulation

Modulation involves partitioning a lattice into two sublattices according to a module, and successively repeating until only trivial or small modules remain, essentially constructing a lexicographic decomposition [90]. In the comparability graph, this corresponds to modular, tree or substitution decomposition [90, 109].

At each step, the surrogates for the module inducing the partition are retained and associated with this module, essentially creating the *quotient* graph induced by this module [109]. Due to the partitioning, the containment relation of the final set of modules forms a tree, called the *containment* or *decomposition* tree and denoted as $\mathcal{CT}$. This tree corresponds to the decomposition graph of Gallai [60, 90]. Subsumption, meets and joins in the original lattice are maintained in the modulated lattice through the individual modules, their surrogates and the containment tree. The orders induced by the modules and $\mathcal{CT}$ will be distinguished using subscripts.

Let us define two functions mapping modules to their surrogates: $\mathcal{S}_{upper}(M)$ and $\mathcal{S}_{lower}(M)$. To simplify our procedures for taxonomic operations in modulated lattices, we define, for an element $x \in L$, $\mathcal{S}_{upper}(x) = x$ and $\mathcal{S}_{lower}(x) = x$. Let us also define a function mapping elements to their smallest containing module: $M(x)$. We can now define the taxonomic operations in a modulated lattice L:

Subsumption $x \leq_L y$ if and only if

i. $M_{context} = M(x) \sqcup_{\mathcal{CT}} M(y)$.

ii. $\exists$ children M_x, M_y of $M_{context}$ for which $M(x) \leq_{\mathcal{CT}} M_x$ and $M(y) \leq_{\mathcal{CT}} M_y$[4].

iii. $\mathcal{S}_{upper}(M_x) \leq_{M_{context}} \mathcal{S}_{lower}(M_y)$.

Meet $x \sqcap_L y = z$ if and only if

i. $M_{context}, M_x, M_y$ are as above.

ii. If $x \geq_L y$ then $z = y$. If $x \leq_L y$ then $z = x$.

iii. If $x || y$ then $z = \mathcal{S}_{lower}(M_x) \sqcap_{M_{context}} \mathcal{S}_{lower}(M_y)$.

Join $x \sqcup_L y = z$ if and only if

i. $M_{context}, M_x, M_y$ are as above.

ii. If $x \geq_L y$ then $z = x$. If $x \leq_L y$ then $z = y$.

iii. If $x || y$ then $z = \mathcal{S}_{upper}(M_x) \sqcup_{M_{context}} \mathcal{S}_{upper}(M_y)$.

Theorem 5.4 *Let L be a modulated lattice. Then the above equivalences for subsumption, meets and joins hold.*

Proof:

Subsumption Let $x, y \in L$. M_x (M_y) is the largest module that contains x (y) but not y (x), and $M_{context}$ is the smallest module that contains both x and y. By the definition of surrogates, $x \leq_L y$ if and only if $\mathcal{S}_{upper}(M_x) \leq_{M_{context}} y$. Similarly, $x \leq_L y$ if and only if $x \leq_{M_{context}} \mathcal{S}_{lower}(M_y)$. Putting these together, we arrive at our result.

Meet Let $x, y \in L$ and suppose $x \sqcap_L y = z$. Clearly, if $x \geq_L y$ ($x \leq_L y$) then $z = y$ ($z = x$).

Otherwise, $z <_L x$, $z <_L y$. M_x (M_y) is the largest module that contains x (y) but not y (x), and $M_{context}$ is the smallest module that contains both x and y. Also, by the definition of modules, z must be an element of $M_{context}$.

By the definition of surrogates, $z \leq_L x$ if and only if $z \leq_{M_{context}} \mathcal{S}_{lower}(M_x)$. Similarly, $z \leq_L y$ if and only if $z \leq_{M_{context}} \mathcal{S}_{lower}(M_y)$. Putting these together, gives our result.

Join The proof is the dual of the proof for meets.

□

In an unmodulated encoding, subsumption requires one comparison of codes and meets require one calculation followed by decoding. Here, subsumption requires one calculation in the containment tree to find the context module and one comparison of codes within this module. Meets require the calculation to find the context module, one calculation within this module and decoding. Thus, although the number of comparisons is greater, the size of each code can be drastically reduced, since the size of the resulting modules and the containment tree will be much smaller than the original ordered set P. For the proposal in [2], the number of operations increases linearly with the depth of the containment tree $\mathcal{CT}$. The above operations are simplified if, for each module, upper and lower surrogates are the same element.

To encode a modulated lattice involves encoding each sublattice formed by the partitioning as well as the containment tree. Any of the techniques previously covered can be used, although there are particularly simple and efficient techniques for encoding trees (e.g. [34]). Associated with each element x is its smallest containing module $M(x)$ and its code $\mathcal{C}_M(x)$ within this module. Associated with each module M is a code for the containment tree and the surrogates $\mathcal{S}_{upper}(M)$ and $\mathcal{S}_{lower}(M)$. The spanning set for the entire lattice is the union of the spanning sets for these sublattices plus the intervals defined by the sublattices themselves. The component mapping will compute the components of an element within its smallest containing module plus the defining interval of this module. Thus, the above operations can be efficiently implemented.

Since any technique can be used to encode a module, modulation opens the possibility of *heterogeneous encoding* [49]: different modules can be encoded using techniques that are best suited to the form of the order within the module. For example, modules that are chains may be encoded using integers, while modules that are anti-chains

[4] If $M(x) = M(y)$, then $M_{context}$ will be a leaf of the containment tree. In this case, $M_x = x$ and $M_y = y$. Essentially, this treats elements as (atomic) modules.

may be encoded using logical terms. In both cases, the use of different techniques can lead to optimal encodings. The only additional information required for a module is the type of encoding technique utilized.

Figure 5.2 depicts a modulated lattice, where the modules are encircled by ovals and named for illustrative purposes. In an implementation, they can be replaced by their surrogate elements. The containment tree of this modulation is also shown. In order to determine if $v \leq d$, we first compute $M_{context} = M_8 \sqcup_{\mathcal{CT}} M_5 = M$. $M_v = M_3$ and $M_d = M_1$. Now, $M_3 \leq_M M_1$, so we conclude that $v \leq d$. To compute $c \sqcap d$, we find $M_{context} = M_1$, $M_c = M_4$ and $M_d = M_5$. Then $M_4 \sqcap_{M_1} M_5 = s$. Similarly, $c \sqcap e$ gives us M_3, the surrogate of which is u.

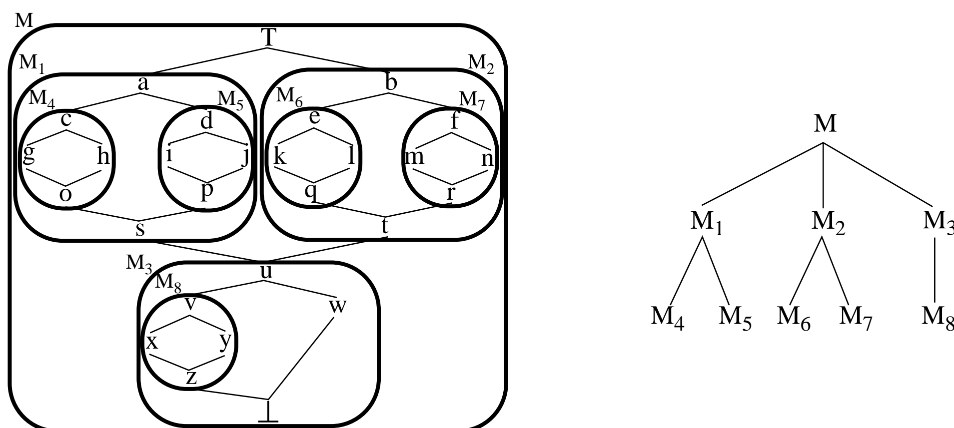

Figure 5.2: A modulated lattice and its containment tree

There still remains the problem of finding modules. Fortunately, we can take advantage of results from comparability graphs. In [109], an algorithm that requires $O(|P|^3)$ time and $O(|P|^2)$ space is described for constructing the entire decomposition tree in a top-down manner. This paper also cites two other algorithms that have time and space complexity of $O(|P|^2)$, the latter of which constructs the decomposition tree incrementally. There exist more recent linear algorithms for producing the entire containment tree [31, 76]. These algorithms may be adaptable to heterogeneous encoding. Also, in [2], an efficient approximation algorithm for modulation is described.

5.4 Extending modulation

The restrictive nature of a module permits efficient partitioning as well as computation of lattice operations. Unfortunately, many lattices cannot take advantage of modulation, particularly very dense lattices. Additionally, in a dynamic environment, modules are fragile and can be breached by the addition of a single arc entering or leaving the middle of the module. We outline below one approach we have developed to make modules more flexible.

5.4.1 Lower and Upper Semi-Modules

Definition 5.6 *Let P be an ordered set. A subset $M \subseteq P$ is called a* lower semi-module *if $\forall x, y \in M$, x is a lower surrogate for y in $P \backslash M$.*

Upper semi-modules are defined dually. For a subset M of a lattice L, we can show that $\sqcap M = a$ is a lower surrogate for M in $M \backslash L$ if and only if M is a lower semi-module. For a lower semi-module, we only obtain a lower surrogate. Elements within the semi-module may have different upper surrogates, but we may still be able to split our lattice on this semi-module, retaining only the surrogates in the original lattice. Thus, instead of an order partition, we end up with an order decomposition and the containment tree becomes a containment order. An example is shown in Figure 5.3. The first diagram is a lower semi-module within the context of our lattice, where element a is a lower surrogate. The second diagram shows a partition of this semi-module (with the grey lines) according to upper surrogates, which are the greatest elements within each partition. Only these elements need be retained in

the original lattice, as we modulate. The set of upper surrogates in a lower semi-module M is the set obtained by the meet closure (within M) of the elements that breach M from above. In this example, elements b, c and e breach M and the meet closure is $\{b, c, e, i\}$, since $e \sqcap c = i$.

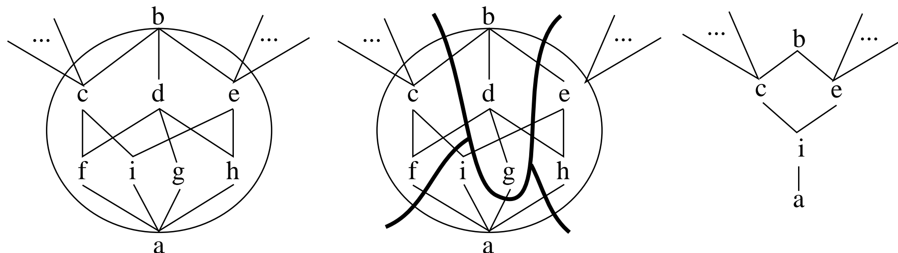

Figure 5.3: Lower semi-modules

Lower modulation incurs some duplication of elements, since the lattice is not partitioned (i.e. the upper surrogates are in both resulting lattices). Each element must now have associated with it not only its smallest containing semi-module, but also its upper surrogate in this semi-module (the lower surrogate is associated with the semi-module). Within the semi-module, the duplicated elements are *ghost* elements - they are no longer treated as other elements, but act as place holders for resolving operations within the semi-module. We may, however, still achieve space savings if we can decompose a lattice using lower semi-modules that do not have too many upper surrogates. Upper semi-modules may be particularly useful for ordered sets that grow dynamically downwards (such as those in [24]). In this case, once an upper semi-module is identified, it will never be breached by later updates, although the number of lower surrogates may change.

5.4.2 Generalized Modules

We can generalize this technique one step further to decompose a lattice based on any interval that is closed under meets and joins. Note that a trivial sublattice of an ordered set P is a singleton set, P itself or the empty set.

Definition 5.7 *A* generalized *module of a lattice L is a non-trivial sublattice of L.*

A number of upper and lower surrogates for the module may need to be left in the parent lattice on decomposition. These elements can be determined as above, where the lower surrogates will be the join closure of elements that breach the module from below. Now, in addition to the smallest containing module M, we need to associate with every element its upper and lower surrogates within this module (as well as its code in M).

There are several consequences of modulation using generalized modules:

i. Modules may overlap: we may have $M_1 \not\subseteq M_2$, $M_2 \not\subseteq M_1$, but $M_1 \cap M_2 \neq \emptyset$. The containment relation is no longer a tree, but a general partial order.

ii. Upper and lower surrogates are no longer associated with modules, but with individual elements.

iii. Ghost elements result in duplication of surrogate elements.

Consider the lattice fragment in Figure 5.4, where we have encircled a potential module M. The left fragment partitions M according to lower surrogates and the right fragment partitions M according to upper surrogates. Modulation on M will remove all the elements that are neither upper nor lower surrogates in M, as shown in the rightmost diagram in Figure 5.4. Each element has a unique upper and lower surrogate in these remaining elements. For elements that are removed (i.e. elements that are neither upper nor lower surrogates for this module), no duplication occurs. Both upper and lower surrogate elements are now duplicated: the element that is in the module is a *ghost* element. We discuss the implications and handling of ghost elements below. Once the decomposition has occurred for a module, we can continue the process of modulation.

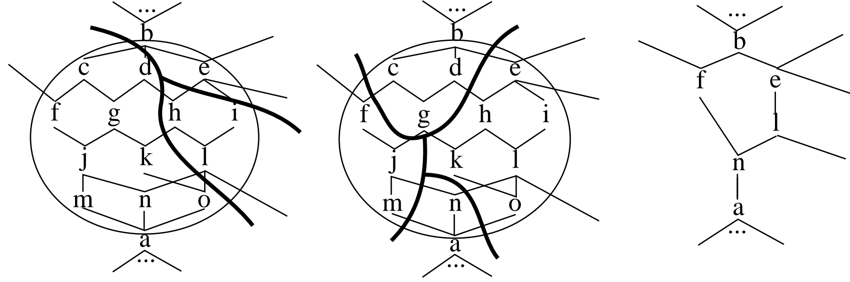

Figure 5.4: Generalized modulation. Lower surrogates (left) are $\{a, e, l\}$ and upper surrogates (centre) are $\{b, e, f, n\}$.

5.4.3 Non-overlapping Modulation

We first consider generalized modulation for modules that do not overlap (i.e. for two modules, either one contains the other, or they share no elements in common). In this case, the containment relation is strictly a tree. In the example in Figure 5.4, further modulation would either contain all or none or $\{a, b, e, f, l, n\}$.

We associate with each module M its code in the containment tree $\mathcal{CT}$. With each element x we associate its least containing module $M(x)$, its upper and lower surrogates $\mathcal{S}_{upper}(x)$ and $\mathcal{S}_{lower}(x)$, and its code in this module.

The procedure for computing subsumption can now be extended with a modification to use the surrogates associated with individual elements rather than modules. A surrogate pathway will need to be followed through the containment tree from the initial elements to the context interval.

Subsumption $x \leq_L y$ if and only if

i. $M_{context} = M(x) \sqcup_{\mathcal{CT}} M(y)$.

ii. $\exists$ elements s_x, s_y in $M_{context}$ that can act as surrogates for x and y (and can be computed as follows, where ":=" denotes assignment.):

$s_x := x$; $s_y := y$;

while $M(s_x) \neq M_{context}$: $s_x := \mathcal{S}_{upper}(s_x)$;

while $M(s_y) \neq M_{context}$: $s_y := \mathcal{S}_{lower}(s_y)$.

iii. $s_x \leq_{M_{context}} s_y$.

A similarly modified procedure can be applied to compute meets. The procedure for joins can easily be derived.

Meet $x \sqcap_L y = z$ if and only if

i. If $x \geq y$ then $z = y$. If $x \leq y$ then $z = x$.

ii. If $x || y$ then

ii.a. $M_{context} = M(x) \sqcup_{\mathcal{CT}} M(y)$.

ii.b. $\exists$ surrogate elements s_x, s_y in $M_{context}$ for x and y (and can be computed as follows):

$s_x := x$; $s_y := y$;

while $M(s_x) \neq M_{context}$: $s_x := \mathcal{S}_{lower}(s_x)$;

while $M(s_y) \neq M_{context}$: $s_y := \mathcal{S}_{lower}(s_y)$.

ii.c. $z = s_x \sqcap_{M_{context}} s_y$.

In strict modulation, surrogates are associated with modules, so once we have found the contextual module $M_{context}$, we can use the surrogates for the appropriate children. Here, we must follow a path of surrogates from the initial elements to the contextual module. Whether upper or lower surrogates are following depends on the operation. The length of these paths depends on the depth of the containment tree, which in turn depends on the sizes of modules.

Note that when performing a meet $x \sqcap y = z$, the result z may be embedded within a module below (in the containment tree) the context $M_{context}$. Due to the way ghost elements are dealt with (i.e. duplicating elements in the meet and join closure of breaching elements), however, this element will be duplicated in both the context and this lower level module (in the latter, it will be a ghost element). Thus, the meet can be performed in $M_{context}$.

Theorem 5.5 *Let L be a lattice that is modulated using generalized modules with no overlapping modules. Then the above equivalences for subsumption and meets hold.*

Proof:

Subsumption Let $x, y \in L$. $M_{context}$ is the smallest module that contains both x and y. Let $s_1 = x, s_2, \ldots, s_k = s_x$ be the path of surrogates followed from x to s_x in the above procedure (i.e. $s_x = \mathcal{S}_{upper}(\mathcal{S}_{upper}(\cdots(\mathcal{S}_{upper}(x)\cdots)))$ and $M(s_x) = M_{context}$). By the definition of surrogates, $x \leq_L y$ if and only if $s_x \leq_{M_{context}} y$. Similarly, $x \leq_L y$ if and only if $x \leq_{M_{context}} s_y$. Putting these together, we arrive at our result.

Meet Let $x, y \in L$ and suppose $x \sqcap_L y = z$. Clearly, if $x \geq_L y$ ($x \leq_L y$) then $z = y$ ($z = x$). Otherwise, $z <_L x$, $z <_L y$. $M_{context}$ is the smallest module that contains both x and y. Also, by the construction used in generalized modulation, z must be an element of $M_{context}$. This is because the meet closure of elements that breach any module are duplicated (one is left in the containing module, and the other is retained as a ghost element in the contained module). By the definition of surrogates, $z \leq_L x$ if and only if $z \leq_{M_{context}} s_x$ (as shown above). Similarly, $z \leq_L y$ if and only if $z \leq_{M_{context}} s_y$. Putting these together, we arrive at our result.

□

An area requiring a closer look is the treatment of ghost elements, which are duplicated upon decomposition. A ghost element x_g is created when an element x is a surrogate for one or more elements in a module M. The element x remains in the parent lattice, and its duplicate x_g remains in the module. This ghost element only needs to be present as an image of x so that operations within the module M which result in x_g can be resolved. Thus, the ghost needs to be encoded in M, but it does not need any other associated information (i.e. the smallest containing module and surrogates). A ghost element x_g can be viewed as a place holder for the portion of the code of x associated with module M.

5.4.4 Overlapping Modulation

In strict modulation, overlapping modules are not possible. In our generalization, this may now occur - this will happen in the example in Figure 5.4 if a new module contains some, but not all of $\{a, b, e, f, l, n\}$. There are two complications that arise from overlapping modules: (i) the containment information is no longer a simple tree, but a general partial order, and (ii) determining the context $M_{context}$ of an operation, and the surrogates in this context, is more difficult.

To deal with these problems, we no longer rely on the containment relation between modules. Instead, we use *surrogate containment* information, and the resulting surrogate containment order $\mathcal{SC}$: For two modules M_1 and M_2, M_2 covers M_1 in $\mathcal{SC}$ if and only if M_2 contains a surrogate for at least one element in M_1 (i.e. iff $\exists x \in M_1$ such that $\mathcal{S}_{upper}(x) \in M_2$ or $\mathcal{S}_{lower}(x) \in M_2$).

Extending the taxonomic operations for overlapping modules requires following surrogate pathways through $\mathcal{SC}$ to find the contextual module. Since we cannot easily identify the contextual module, rather than encoding $\mathcal{SC}$, we associate with each module M a level, $level(M)$, which is the length of the longest path from M to the root of $\mathcal{SC}$. The modified procedures for subsumption (for generalized modules) and meets are given below.

Subsumption $x \leq_L y$ if and only if

i. $\exists$ $M_{context}$ and elements s_x, s_y in $M_{context}$ that can act as surrogates for x and y (and can be computed as follows):

$s_x := x$; $s_y := y$;
$Lev := max(level(M(s_x)), level(M(s_y))) - 1$;
while $M(s_x) \neq M(s_y)$
  while $level(M(s_x)) > Lev$: $s_x := \mathcal{S}_{upper}(s_x)$;
  while $level(M(s_y)) > Lev$: $s_y := \mathcal{S}_{lower}(s_y)$;
  $Lev := max(level(M(s_x)), level(M(s_y))) - 1$;
end while;
$M_{context} := M(s_x)$.

ii. $s_x \leq_{M_{context}} s_y$.

Meet $x \sqcap_L y = z$ if and only if

i. If $x \geq y$ then $z = y$. If $x \leq y$ then $z = x$.

ii. If $x||y$ then

ii.a. $\exists$ $M_{context}$ and elements s_x, s_y in $M_{context}$ that can act as surrogates for x and y (and can be computed as follows):

$s_x := x$; $s_y := y$;
$Lev := max(level(M(s_x)), level(M(s_y))) - 1$;
while $M(s_x) \neq M(s_y)$
  while $level(M(s_x)) > Lev$: $s_x := \mathcal{S}_{lower}(s_x)$;
  while $level(M(s_y)) > Lev$: $s_y := \mathcal{S}_{lower}(s_y)$;
  $Lev := max(level(M(s_x)), level(M(s_y))) - 1$;
end while;
$M_{context} := M(s_x)$.

ii.b. $z = s_x \sqcap_{M_{context}} s_y$.

Theorem 5.6 *Let L be a lattice that is modulated using generalized modules (with possible overlapping modules). Then the above equivalences for subsumption and meets hold.*

Proof:

Subsumption Let $x, y \in L$. We need to find $M_{context}$ as well as surrogates for x and y in $M_{context}$. The level of modules decreases monotonically as we ascend the surrogate containment order $\mathcal{SC}$ searching for $M_{context}, s_x$ and s_y, but it may decrease in steps greater than one.

Initially, we set Lev to one level above the lowest (maximum) level of x and y. This ensures that the lowest of s_x, s_y (or both if they are at the same level in different modules) will move up at least one level in the subsequent two loops. The outer loop continues until we have found $M_{context}$ (i.e. until $M(s_x) = M(s_y)$). The two inner loops each continue until the level of s_x (s_y) is at or above Lev. Since we are following upper (lower) surrogates for s_x (s_y), the subsumption relation between s_x and s_y remains invariant. After both inner loops complete, we set Lev again as above.

At the end of the loops, $M(s_x) = M(s_y) = M_{context}$. $M_{context}$ is the smallest module that contains both an upper surrogate for x and a lower surrogate for y. Since we move up $\mathcal{SC}$ following upper (lower) surrogates for x (y), we find the first module that contains appropriate surrogates for both.

Let $s_1 = x, s_2, \ldots, s_k = s_x$ be the path of surrogates followed from x to s_x in the above procedure (i.e. $s_x = \mathcal{S}_{upper}(\mathcal{S}_{upper}(\cdots(\mathcal{S}_{upper}(x)\cdots)))$ and $M(s_x) = M_{context}$). By the definition of surrogates, $x \leq_L y$ if and only if $s_x \leq_{M_{context}} y$. Similarly, $x \leq_L y$ if and only if $x \leq_{M_{context}} s_y$. Putting these together, we arrive at our result.

Meet Let $x, y \in L$ and suppose $x \sqcap_L y = z$. Clearly, if $x \geq_L y$ ($x \leq_L y$) then $z = y$ ($z = x$).

Otherwise, $z <_L x$, $z <_L y$. We need to find $M_{context}$ as well as surrogates for x and y in $M_{context}$, as above.

Initially, we set Lev to one level above the lowest (maximum) level of x and y. This ensures that the lowest of s_x, s_y (or both if they are at the same level in different modules) will move up at least one level in the subsequent two loops. The outer loop continues until we have found $M_{context}$ (i.e. until $M(s_x) = M(s_y)$). The two inner loops each continue until the level of s_x (s_y) is at or above Lev. Since we are following lower surrogates for s_x (s_y), the subsumption relation between s_x (s_y) and z remains invariant. After both inner loops complete, we set Lev again as above.

At the end of the loops, $M(s_x) = M(s_y) = M_{context}$. $M_{context}$ is the smallest module that contains a lower surrogate for both x and y. Since we move up $\mathcal{SC}$ following lower surrogates for x (y), we find the first module that contains appropriate surrogates for both.

By the construction used in generalized modulation, z must be an element of $M_{context}$. This is because the meet closure of elements that breach any module are duplicated (one is left in the containing module, and the other is retained as a ghost element in the contained module).

By the definition of surrogates, $z \leq_L x$ if and only if $z \leq_{M_{context}} s_x$ (as shown above). Similarly, $z \leq_L y$ if and only if $z \leq_{M_{context}} s_y$. Putting these together, we arrive at our result.

□

5.4.5 Extending Modulation Algorithms

We have outlined the properties and requirements of generalized modulation for encoding purposes, but we need algorithms that can find "good" decompositions. Perhaps some of the algorithms for modulation can be adapted to

decompose an ordered set into lower semi-modules or generalized modules for which the number of surrogates (i.e. the degree of duplication) and the amount of overlap is minimized.

Although generalized modulation may not guarantee encoding efficiency, it does offer many potential benefits. First, the fragility and stringent nature of strict modules makes modulation impractical for many encoding environments, especially for ordered sets that are dense. Although generalized modulation may still be inefficient for very dense lattices, there is the opportunity to expand the utility of decomposition and heterogeneous encoding. Generalized modulation may also be used in conjunction with strict modulation in dynamic environments. Starting with a modulated lattice, updates to the lattice that breach modules may be tolerated, while incurring only a small overhead for updating the encoding. When the decomposition becomes inefficient, the new ordered set can be re-modulated. Another benefit of generalized modulation is in distributed environments, in which a large ordered set may be spread out over a number of sites. The portion of the ordered set at each site can be encoded independently of the others, and duplication of information across sites may only be necessary for the containment order.

5.5 Conclusion

Recent results in taxonomic encoding have identified various taxonomic forms for which efficient encodings exist (e.g. distributive lattices, trees and bounded width lattices). Through order partitioning techniques, a generalized heterogeneous encoding scheme can take advantage of these encoding schemes when such forms are identified as suborders.

In this chapter, we formalized and extended lattice modulation for encoding, introduced in [2]. Modulation partitions a lattice to encode into sublattices and offers the possibility of greatly reducing encoding sizes regardless of implementation, and without undue cost in performance. Generalized modules may increase the applicability of modulation, even for dense, dynamic or distributed lattices. By maintaining (and encoding) the containment information of the decomposition, we provide an efficient framework in which modulated encoding is both feasible and efficient.

For dynamic taxonomies, modulation may confine the extent of change. The strict nature of modules, however, makes them susceptible to violation as a result of change. The generalized modules developed in section 5.4 are more impervious to change. Finally, modulated encoding may aid in decoding, since we know in which partition the result lies, greatly reducing the search space.

Chapter 6

Encoding with Sparse Logical Terms

"Unless you expect the unexpected you will never find truth,
for it is hard to discover and hard to attain"

– Heraclitus

The purpose of the present chapter is to empirically apply the theory of encoding. During our research, we developed *sparse logical terms* as a variant of logical terms that are particularly suitable for encoding [51]. Sparse terms are closely related to directed acyclic graphs (DAGs), which have also been studied for encoding [104]. Our focus, however, is on *developing* an efficient implementation for encoding rather than taking an existing technique. Sparse terms share a number of similarities with Prolog terms, ψ-terms in LIFE [4], feature structures [5, 23, 118], the PATR II formalism [131, 132], etc. However, the focus of sparse terms as an efficient representation for encoding endows them with a number of key distinctions from these other formalisms, as will become clear. Since our aim is to use sparse terms as a contribution to encoding, rather than as a contribution to the suite of logical formalisms, we chose to omit in-depth coverage of these related formalisms.

After motivating our development of sparse terms, we introduce the basic form of sparse term developed in [51]. In section 6.3, we develop extensions that make sparse terms suitable as a universal encoding implementation. We then provide algorithms that implement the transitive closure and compact encoding techniques, which are the first logical term algorithms to be published. Finally, we analyze some theoretical properties of sparse terms in encoding, which we back up with an empirical study of encoding using two taxonomies derived from existing applications.

6.1 Introduction

Compact representations for data structures are commonly used when certain properties can be exploited to significantly reduce the storage space required. As an example, principles of locality are used in data compression techniques. For sparse matrices, the assumption that the majority of elements are zero permits us to retain only the nonzero elements, along with their coordinates. If this assumption holds true, the savings accrued by not explicitly storing the zero elements outweighs the additional cost of storing coordinates for nonzero entries.

We develop a similar representation for logical terms. A sparse term is a term in which the majority of elements (i.e. functors, atoms and variables) are anonymous variables. Named variables provide coreference between term positions, whereas the only purpose of anonymous variables is to reserve positions, and so they do not contribute to the information content of a term.

Applications that work with sparse terms can benefit from sparse terms both in terms of space and time. Unification with an occurs check needs only to examine the named variables. Unification without an occurs check is linear in the sum of the number of atoms, functors and variables of the two terms. This will be more efficient, as our sparse representation eliminates the storage of anonymous variables.

Sparse terms were, however, developed primarily to provide a form of logical term adapted for encoding. In extending the basic sparse term, we incorporate integer sorts (i.e. when unifying two different functors f_1 and f_2, if both are integers, the result is $max(f_1, f_2)$; if at least one is not an integer, then unification fails). Integer sorts come for "free", and can be used to generalize integer vectors: integer sorts provide a form of *sparse integer vector*

that permits the integration of integer vectors and logical terms. This combination is powerful for encoding, since integer sorts are suited for encoding chains, while ordinary functors are suited for encoding anti-chains.

We also integrate more compact and flexible forms of subterm indexing. The basic form of sparse terms are very compact for terms with many anonymous variables. However, as the terms become less sparse, the overhead of explicit subterm indexing surpasses the savings of eliminating anonymous variables. In the expanded form, we permit "relative" indices which denote integer indices that are relative to preceding integer indices in a term. In this more expressive form, as a term becomes more dense, the sparse term representation can remain more compact, up to a point, than the corresponding ordinary terms or integer vectors.

We also permit grouping sequences of indices with identical subterms into intervals. For encoding, this will normally only occur for unspecified subterms. Index intervals in sparse terms provide a generalized implementation of sets of intervals, which have also been used in encoding [1]. Figure 6.1 shows the relation of sparse terms to the encoding implementations of which we are aware.

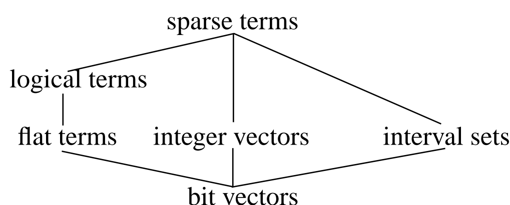

Figure 6.1: Encoding implementations: sparse terms generalize other techniques

6.2 Basic Sparse Terms

Our representation is modeled after that of sparse matrices. An $n \times m$ sparse matrix may be stored as a list of coordinate/value pairs for the non-zero elements rather than as an $n \times m$ array. For example, the following matrix can be stored as [(1,2)-1, (2,4)-5, (4,2)-3, (4,5)-4]:

$$\begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 4 \end{vmatrix}$$

We avoid storing the zeros by using a more space-consuming representation for the non-zero elements. By assuming that most of the elements are zeros we predict a net reduction in storage space.

A sparse term representation relieves us from storing anonymous variables at the expense of a more complex scheme for the named elements (i.e. atoms, predicates, functors and named variables). We focus on the surface form of terms. Although the internal representation may be quite different from this and is implementation dependent, it is the surface form that users manipulate and store outside the system. As for sparse matrices, we need to store the position, or index, of the named elements. Using a rooted graph notation, we can do this by labeling arcs with the index of the named elements and removing the anonymous variables (which are represented by underscores in Prolog). Consider the Prolog term: $a(b(_, c, d, _), _, _, e(_, f(_, _), _))$. The ordinary and sparse forms are shown graphically below. The sparse term can be represented linearly as: $a.[1 - b.[2 - c,\ 3 - d],\ 4 - e.[\ 2 - f]]$, where the argument lists are ordered according to increasing index.

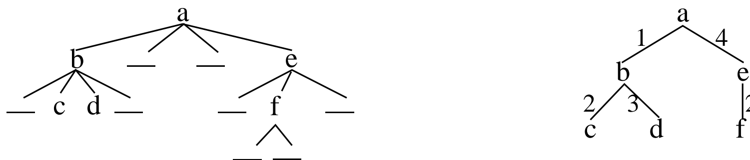

Figure 6.2: Sparse logical terms

To be more precise, we provide the following definition of our representation:

Definition 6.1 *A* basic sparse term *is either (i) an atom (ii) a named variable or (iii) a functor of the form a.L, where a is the functor symbol and L is a sparse argument list. A sparse argument list is a list of elements of the form n-ST, where ST is a sparse term and n is the index of ST in the parent term. This list is ordered by increasing indices with no repetitions.*

6.2.1 Space requirements

Now that we have a sparse representation for logical terms, when is a term considered sparse? That is, when will this representation benefit an application? Since an accurate account of the space required to represent a logical term, for example in Prolog, is implementation dependent, we restrict our analysis to the asymptotic time and space behavior of the surface form.

Consider an ordinary term that has n named elements and m anonymous variables. Since there are $n+1$ symbols, let us assume representing each requires O(log$n$) space. For the sparse representation, O(log$n$) space is also required. Both representations require space for the n named elements, so we do not include this factor in our calculations. For punctuation marks (e.g. commas, parentheses, dashes), ordinary terms require O($n+m$) space whereas sparse terms require O(n) space. Since punctuation may not form part of the internal representation, we do not consider it further.

In addition to the above, ordinary terms require O($m$log$n$) space for anonymous variables, whereas in the worst-case sparse terms require O($n$log($n+m$)) space for indices. Essentially, this means that the space benefits of our sparse representation begin to manifest when the ratio of anonymous variables to named elements is greater than one. Of course, due to the constants not included in this analysis, these benefits may not become evident until this ratio is somewhat greater than this.

6.2.2 Unification and Implementation

Without an occurs check, unification of both ordinary and sparse terms is linear in the number of symbols involved. If the number of named elements in both terms is n and the number of anonymous variables is m, we have O($n+m$) for ordinary terms vs. O(n) for sparse terms. For unification with an occurs check, we avoid needlessly checking the anonymous variables. In both cases, we achieve asymptotically better results. Thus, by using our sparse representation, applications involving sparse terms have potential benefit both in terms of time and space.

The straightforward nature of sparse terms permits a simple implementation of the required algorithms (unification, subsumption, etc.) either in a logic language (e.g Prolog) or as an extension to a logic language (written in, e.g., C). Our representation shares some features with the ψ-terms in LIFE [4], in particular attribute indexing and unbound arity, but it also differs in several respects. Named variables in LIFE use more generalized coreference labels (which can specify coreference between any two locations in the graphical representation, not just between leaves). Although our definition of sparse terms implies the use of Prolog variables, we have also extended our implementation to provide both forms of coreference. Our representation also deviates from ψ-terms in the use of anonymous and disjunctive functors, discussed below. Another significant difference is that our representation is intended as an enhancement to Prolog systems, not as a replacement.

6.2.3 Variations

Our sparse representation removes the burden of explicitly storing anonymous variables. We now explore some variations on this theme. Prolog is capable of expressing uncertainty through variables, only for entire predicates, functors or atoms. We analyze how we may incorporate finer scale uncertainty into logical terms, specifically for arity and functors. We also integrate an extension of argument indexing that permits arbitrary labels, or attributes, rather than just numerical indices. By blending these variations, applications have the ability to incorporate varying degrees of uncertainty and information into logical terms, while remaining concise and efficient.

Binding arity. The representation presented does not provide a one-to-one correspondence between sparse and ordinary terms. For example, the following terms correspond to the sparse term $f.[1-a]$: $f(a)$, $f(a,_)$, $f(a,_,_)$, $f(a(_),_)$, ... Any sparse term has an infinite number of corresponding ordinary terms. The arity of each functor and atom is not bound, so we can always append an arbitrary number of anonymous variables as arguments of functors and atoms.

If we require the arity of terms to be bound, we must specify it explicitly. This can be accomplished by extending part (iii) of our definition to allow functors of the form $a/N.L$ where a is a functor, N is the arity of the functor and L is a sparse argument list. For example, the term $f(_, b(_,_), c, d(e,_),_)$ is completely represented by $f/5.[2-b/2, 3-c/0, 4-d/2.[1-e/0]]$, and graphically as:

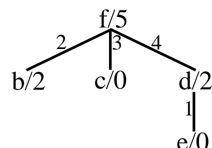

Figure 6.3: Binding arity in sparse terms

Anonymous functors. An interesting variation that we have found useful for encoding allows terms to specify only those argument positions that are occupied, but not record the functor or atom in that position. This information, presumably, would be stored elsewhere. This greatly reduces space requirements for cases when many terms are being formed from one set of data, which is indeed the case for our logical term encodings where each element of a taxonomy is assigned a term that is a subgraph of the taxonomy itself. We can label the original taxonomy with term positions and use it to decode our terms. To provide functorless terms, we simply remove the functor or atom from the elements of the sparse argument list. The term f(_,b(_,_),_,c(d,_,e),_) would thus be represented as the term [2,4-[1,3]] and graphically as:

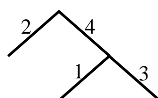

Figure 6.4: Anonymous functors in sparse terms

Attribute-value matrices. Attribute-Value Matrices (AVMs), or Feature Structures, are a tool used in several computational linguistic systems (e.g. [118]). Some implementations of AVMs using ordinary terms require prior knowledge of all the attributes an AVM may contain in order to compile appropriate terms (e.g. [91, 119]). A simple modification to our scheme, allowing atomic, rather than numeric, indices (for the attributes) and omitting functor names (a value is either an atom or another AVM), provides for efficient and dynamic AVMs. A predicate can be provided to access the value of an attribute, or a sequence of attributes. As an example, the sparse term [a_1-v_1, a_2-v_2, a_3-[b_1-x_1, b_2-x_2], a_4-v_4] represents the following AVM (shown in both its matrix and graphic forms):

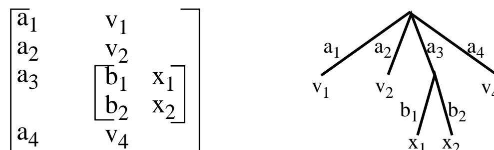

Figure 6.5: Attribute-value matrix using sparse terms

Disjunctive functors. Thus far, we have permitted two levels of certainty regarding a functor symbol: either it is unknown (i.e. it may be any atomic symbol) or it is known. Between these extremes lies a range of increasingly focused information as to the actual functor symbol. That is, we may know that it is one of a

set of possible symbols. When this set has cardinality one, we know which symbol it must be. We name such functors *disjunctive* and represent them with a set notation. For example, the term [model-{MacSE; MacII}, memory-{1;2;4;8}] may be used to represent a computer system whose model type is either a MacSE or a MacII and with either 1, 2, 4 or 8 KB of memory.

Applications that permit and maintain uncertainty may find the flexibility offered by disjunctive functors a valuable property. Examples include computational linguistics, for maintaining the uncertainty of the referent of a pronoun, and automatic system configuration (e.g. [37]).

6.3 Generalizing Sparse Terms for Encoding

Basic sparse terms are based on the observation that anonymous variables only reserve positions and do not contribute to the information content of a logical term. We now extend the basic form to develop a *universal encoding implementation.* In addition to the benefits of eliminating anonymous variables, there are some properties of extended sparse terms that endow them with flexibility and conciseness required for encoding:

Unbound arity A sparse term can represent an infinite number of ordinary terms, since arity is not bound. This permits flexibility for encoding updates since a code may be extended with a subterm without affecting related codes.

Unspecified functors Positions in terms can be specified as filled, but the actual symbol (predicate, atom or functor) occupying the position can be left unspecified. Thus, $[2, 4 - [1, 3]]$ represents a term in which the second position is occupied by an unknown subterm, and the fourth position is occupied by a subterm in which the first and third positions are filled. Of course, unification can only fail if there are different functors at the same location in two terms.

Integer sorts Although sparse terms were designed for encoding, they share a number of similarities with ψ-terms in LIFE [4], including unbound arity. A hierarchy can be specified among functors in LIFE, which is used when two different functors are unified. If α and β are unified, the result will be $\alpha \sqcap \beta$ or failure if this results in $\perp$. One of the most influential papers on encoding was written with the purpose of performing these operations efficiently [2]. However, there is a very simple functor ordering that we can incorporate into sparse terms for free: the total order on integers. Unification of two functors will be as in Prolog, unless both are integers n_1 and n_2, in which case unification will result in $max(n_1, n_2)$. This simple addition generalizes integer vectors, providing a form of *sparse integer vectors* with the hierarchical advantages of logical terms.

Relative Indices As terms become less sparse, the advantages of explicit indexing diminish until the costs outweigh the benefits. To overcome this, some indices may be *relative.* Relative indices can be specified by preceding a positive integer n by the "+" symbol, and represent the previous numerical index to the left plus n. If there is no preceding numerical index, then the index is n. For example, the sparse term $[535, 538, 546, 577, 578]$ could be represented as $[535, +3, +8, +31, +1]$. Although we must still provide an index, if the absolute index is very large, a space saving may be realized.

Interval Indices As terms become even more compact, there may be situations (particularly for encoding) in which we can benefit from denoting a sequence of indices using a set. These *interval indices* provide a generalization of interval sets, which have been used for encoding [1]. To illustrate, the sparse term $[5, 6, 7, 8, 9, 10, 11, 12, 73, 74, 75, 76, 77]$ could be represented as $[(5, 12), (73, 77)]$. Relative indices can also be used in the interval bounds.

As we have mentioned, sparse terms generalize the various implementations that have been used for encoding. The significance of this is that, not only can encoding algorithms be adaptive and selected from existing encoding techniques, but mixtures of techniques can take advantage of structures within taxonomies. The following definition is based on the original definition, but extended with integer sorts, and relative and interval indices. We do not provide any form of coreference, since it is not necessary for our application, although this could be easily integrated.

Definition 6.2 *A* sparse term ST *is defined as:*

$ST \doteq Functor.ArgumentList \mid ArgumentList \mid Functor$
$Functor \doteq Atom \mid NaturalNumber$
$ArgumentList \doteq [Argument|ArgumentList] \mid []$
$Argument \doteq Index{-}ST \mid Index$
$Index \doteq NumericIndex \mid (NumericIndex, NumericIndex) \mid Atom$
$NumericIndex \doteq NaturalNumber \mid +NaturalNumber$

where $NaturalNumber$ *is any natural number. The notation* $[Head|Tail]$ *denotes a list, the first element of which is* $Head$ *and the remainder of which is* $Tail$*, while* $[]$ *denotes an empty list (as in Prolog).*

6.3.1 Explicit and canonical forms for sparse terms

In order to simplify description of a canonical form, and for defining subsumption, unification and anti-unification, we need to describe an explicit form for sparse terms. The explicit form replaces all relative indices by their corresponding absolute values, and all interval indices by their corresponding sequences. We also clarify terms that have empty argument lists or no functors, where explicit sparse terms use anonymous variables ("_") in place of unspecified functors.

Definition 6.3 *An* explicit sparse term ST_x *is defined as:*

$ST_x \doteq Functor_x.ArgumentList_x$
$Functor_x \doteq Atom \mid NaturalNumber \mid _$
$ArgumentList_x \doteq [Index_x{-}ST \mid ArgumentList] \mid []$
$Index_x \doteq NaturalNumber \mid Atom$

Given a sparse term ST, we can construct its explicit form as follows:

Empty Argument Lists If F is a subterm with an empty argument list (i.e. F is just a functor), then replace it by $F.[]$.

Unspecified Functors If AL is a subterm with an unspecified functor (i.e. AL is just an argument list), then replace it by $_.AL$. Note that in sparse terms, the anonymous variable can only be instantiated to a functor.

Relative Indices Suppose $+n$ is the first relative index in an argument list (including those that appear in interval indices): $[\ldots, +n - ST, \ldots]$. If there is no absolute numerical index to the left of this position, then replace $+n - ST$ by $n - ST$. Otherwise, if the first absolute numerical index to the left of this position is m, then replace $+n - ST$ by $n_1 - ST$, where $n_1 = n + m$.

Interval Indices Suppose we have an argument list containing an interval index: $[\ldots, (n_1, n_2){-}ST, \ldots]$. If $n_1 > n_2$, then simply remove $(n_1, n_2) - ST$ from the argument list (i.e. the interval is empty). Otherwise, replace it by the sequence $m_1 - ST, \ldots, m_k - ST$, where $m_1 = n_1$, $m_{i+1} = m_i + 1, 1 < i \leq k$, and $k = n_2 - n_1 + 1$.

Given an arbitrary sparse term, for efficiency we want to define a canonical or normal form. For terms in canonical form, subsumption, unification and anti-unification algorithms can be designed much more efficiently than otherwise possible (i.e. linear in term size). Below we define a canonical form for a term ST in terms of its explicit form. We say that ST is in canonical form, if its explicit form is in canonical form.

Let ST be a sparse term, and ST_x be its explicit form. We define the *canonical form* ST_c of ST as follows:

No duplicate indices If ST_x has a duplicate index I in some argument list: $[\ldots, I - ST_1, \ldots, I - ST_2, \ldots]$, then remove $I - ST_1$ and $I - ST_2$ and add $I - ST_{1,2}$, where $ST_{1,2}$ is the unification of ST_1 and ST_2.

Indices in increasing order For any subterm in ST_x, if index I_1 precedes index I_2 then $I_1 \sqsubseteq I_2$, where $\sqsubseteq$ denotes a lexical ordering on indices.

6.3.2 Sparse term subsumption

We now describe how subsumption ($\preceq$) is computed for explicit canonical sparse terms. Unification and anti-unification can easily be derived in a standard way based on subsumption. All three operations have been implemented in Sicstus Prolog. Converting from an ordinary canonical sparse term to the explicit form can be done easily during processing. First, some general properties are given below:

- [] subsumes everything (i.e. $ST \preceq []$ for any sparse term ST).
- If n_1, n_2 are integers and $n_1 \leq n_2$ then $n_2 \preceq n_1$ (note the role reversal).
- If n is an integer and a is a non-integer atom, then $n||a$ (i.e. n and a are incomparable).
- If a_1, a_2 are non-integer atoms and $a_1 \neq a_2$, then $a_1||a_2$.

Definition 6.4 *If ST_1 and ST_2 are sparse terms, then $ST_1 \preceq ST_2$ if and only if all of the following hold:*

1. $ST_1 = F_1.ArgList_1$ *and* $ST_2 = F_2.ArgList_2$
2. $F_1 \preceq F_2$
3. $ArgList_1 \preceq ArgList_2$

If F_1 and F_2 are functors, then $F_1 \preceq F_2$ if and only if one of the following holds:

1. $F_2 = _$ *(functorless terms)*
2. F_1 *and* F_2 *are non-integer atoms and* $F_1 = F_2$ *(atomic functors)*
3. F_1 *and* F_2 *are integers and* $F_2 \leq F_1$ *(numeric functors)*

If $ArgList_1$ and $ArgList_2$ are argument lists, then $ArgList_1 \preceq ArgList_2$ if and only if one of the following holds:

1. $ArgList_2 = []$
2. $ArgList_1 = [Index_1 - ST_1|Rest_1]$, $ArgList_2 = [Index_2 - ST_2|Rest_2]$ *and one of the following holds:*
   (a) $Index_1 = Index_2$, $ST_1 \preceq ST_2$ *and* $Rest_1 \preceq Rest_2$
   (b) $Index_1 \sqsubseteq Index_2$ *and* $Rest_1 \preceq ArgList_2$

6.4 Encoding with Sparse Terms

The most well-studied implementation for encoding is the bit-vector [2, 24, 61, 79]. The available hardware implementation and minimal requirements for each item of information (one bit) makes them attractive for encoding. However, there are a number of drawbacks to using bit-vectors for encoding very large, dynamic ordered sets:

- Codes in a bit-vector implementation all have the same size, so updates to the encoding that require changing this length affect every code. This problem is shared with integer vectors. Sparse terms, however, do not suffer from this, so the scope of change can be contained.
- Both logical terms and integer vectors generalize bit-vectors in different dimensions (see Chapter 4). A bit-vector s of length k can be represented with a logical term τ of arity k: if position i in s is a 1 (resp. 0), then position i in τ is the functor 1 (resp. an anonymous variable). The translation from bit-vectors to integer vectors is obvious. Thus, any bit-vector encoding can be translated to use sparse terms and exhibit the same asymptotic behaviour; only the asymptotic constant changes. Since we are most concerned with asymptotic behaviour for encoding large taxonomies, bit-vectors do not actually provide any real benefit, although their inflexibility is certainly a drawback. In fact, we show later how the hierarchical structure of sparse terms can provide a significant savings over bit-vectors even for modest taxonomies of only several thousand nodes.

As we showed in Chapter 4, all encoding algorithms we are aware of can be abstracted into two components: (i) the underlying information stored in the encoding (which can be characterized using what we call *spanning sets*) and (ii) the implementation details for storing this information in a computer. Some encoding algorithms require a lot of effort to generate codes. This is understandable, given the complexity of the problem (in [79], evidence for the NP-Hardness of finding optimal encodings is discussed). For static taxonomies, it may be worthwhile spending a lot of energy to construct compact encodings. For dynamic taxonomies, however, this effort may be wasted by changes to the hierarchy. In fact, the changes required for an encoding after updates to the source taxonomy may be more extensive in complex encodings, due to the wider scope of analysis performed.

Encoding algorithms for dynamic taxonomies must be efficient, in addition to generating efficient codes. Two of the earliest and most well-known, encoding algorithms (*transitive closure* and *compact* [2]) satisfy the need for efficient computation of codes. However, the algorithms described directly construct bit-vector implementations. As we showed in Chapter 4, these basic algorithms form the basis of a number of encoding techniques. We describe how sparse terms can implement these simple schemes. This in itself does not contribute significantly, but we show in a subsequent section how sparse terms equal or surpass other implementations for encoding a number of theoretical ordered sets. This is followed empirically, where two ordered sets taken from existing applications are encoded using the transitive closure and compact algorithms. These results are compared with the space requirement for bit-vectors.

Since we are concerned with large taxonomies, we must carefully count space requirements (i.e. an integer of size n takes $logn$, not constant, space). Two common techniques for implementing a graph $G = (P, E)$ are *adjacency matrices*, which take $O(|P|^2)$ space, and *adjacency lists*, which take $O(|E|log|P| + |P|)$ space. Adjacency list representation corresponds to maintaining the list of parents (or children) for each element.

Both the encoding algorithm and the implementation affect these characteristics. Since the requirements of particular taxonomic applications may differ, it is apparent that there may be no *best* encoding algorithm to satisfy all needs. Rather, the designer of an encoding algorithm must take into account the needs of the application, and the form of the taxonomies to encode, in order to determine the relative importance of different characteristics.

Most existing algorithms concentrate on the resulting codes and have not been as concerned with the complexity of the encoding algorithm or of dynamic updates. In addition to the space requirement of the resulting codes, we focus on these two issues.

6.5 Sparse Term Encoding

The simple transitive closure and compact encoding algorithms in [2] satisfy one of our goals: the complexity of the encoding algorithm is minimal. Transitive closure has an additional advantage: decoding (i.e. determining the element(s) denoted by a given code) can be done efficiently in both bit-vector [47, 61] and sparse term implementations. Sparse terms use a spanning tree of the order for decoding in time linear in the depth of a code term. Research on complex encoding algorithms to find optimal encodings (e.g. [79]) is important, but is of limited practical use in dynamic environments. Below we use the abstract versions of these two simple encoding algorithms described in Chapter 4 to specify versions that compute sparse term encodings. Note that we use these algorithms in a *top-down* manner (which preserve joins), while the dual *bottom-up* versions (which preserve meets) were described in [2].

The transitive closure algorithm for sparse term encoding is given below. Several variations were implemented in Sicstus Prolog, and were used to derive the empirical results of section 6.7. A topographic traversal of the ordered set is done so that, when processing an element p, the codes for all parents of p have already been constructed. Associated with each element p is a "path" (a sequence of indices from the root of the code $\tau(p)$ to one of the leaves), and a "label" indicating how to extend $\tau(p)$. The code for an element is built from the unification of the parent codes, plus an extension of the path associated with one of its parents. The subroutine *extend* will select one of the parents to extend, and either increment an integer sort (done through *extend_integer_sort*) or add a new subterm (done through *extend_arglist*). These two straightforward functions are not described.

Algorithm 1 *sparse_term_encoding(input: P; output:τ)*

1. *let* $< p_1, \ldots, p_n >$ *be a (top-down) topographic ordering of* P, *where* $p_1 = \top$
2. $\tau(\top) := [\,]$
3. $path(\top) := [\,]$
4. $label(\top) := 1$
5. *for* $i = 2$ *to* n *do*
6. $\quad \tau(p_i) := \sqcap_{q \in parents(p_i)} \tau(q) \sqcap$ *extend*(p_i)

Algorithm 2 *extend(input: p; output:α)*
Global information: ordered set P ($p \in P$), and path, label and pred information

1. *if* $\exists q \in parents(p)$ *such that* $label(q) > 0$ *then*
2. $\quad \alpha := extend_integer_sort(path(q), label(q))$
3. $\quad path(p) := path(q)$
4. $\quad label(p) := label(q) + 1$
5. $\quad$ *if* $label(q) = 1$ *then*
6. $\quad\quad label(q) := -1$
7. $\quad$ *else*
8. $\quad\quad label(q) \rightarrow label(pred(q))$
9. $\quad$ *endif*
10. *else*
11. $\quad$ *select any* $q \in parents(p)$
12. $\quad n :=$ -$label(q)$
13. $\quad \alpha := extend_arglist(path(q), n)$
14. $\quad path(p) := \alpha$
15. $\quad label(p) := 1$
16. $\quad label(q) := -(n + 1)$
17. *endif*
18. $pred(p) := q$

Note the polymorphic use of the predicate *label*. If *label* is a positive integer n, then term extension is to be accomplished by setting the integer sort at the end of the path specified in the *path* predicate to n. If *label* is a negative integer $-n$, then term extension is to be accomplished by adding a new subterm at the end of *path* with index n. Also note that we used ":=" to denote variable assignment, while the symbol "$\rightarrow$" is used to denote identity (i.e. in line 8, $label(q)$ becomes identical to the label of its predecessor $pred(q)$). Essentially, if any parent q can be extending by incrementing an integer sort, we select that parent (lines 1 to 9). The current element p inherits the path of q (line 3) and increments the next integer sort extension (line 4). If the label for q is 1 then a new subterm list is begun (line 5), otherwise subterm expansion is done using its predecessor's sublist (line 8) so new subterm extensions will be done correctly (since q and its predecessor have the same *path*). In both cases, new extensions will be argument list extensions. If no parent can be extended with integer sorts, we select one to extend by adding a new subterm (lines 11 to 17). The label is the negation of the new subterm index, which is used to extend the path of q, and also becomes the new path of p (lines 12 to 14). Now p can be extended by incrementing (the currently non-existent) integer sort functor (line 15), while the next extension of q is updated (line 16). The last line sets up the predecessor information.

For compact encoding, we need only change line 6 of the sparse term encoding algorithm to the following, so that only the codes for meet irreducible elements are extended. The code for a non-meet irreducible element is simply the unification of the parent codes.

6.1. if p_i is meet irreducible then
6.2. $\tau(p_i) := \sqcap_{q \in parents(p_i)} \tau(q) \sqcap extend(p_i)$
6.3. else
6.3. $\tau(p_i) := \sqcap_{q \in parents(p_i)} \tau(q)$

Postprocessing can optimize codes to use relative and interval indices, where a space saving can be realized. For dynamic updates to the taxonomy, variations of these algorithms can modify existing encodings by updating only codes below the point of change, although we do not describe these here.

6.6 Theoretical Justification

We now justify, using a variety of theoretical taxonomies, that sparse terms provide the necessary flexibility and efficiency required for encoding. This analysis complements an earlier theoretical comparison of various encoding techniques, including flat terms, on theoretical orders [43], where the focus was on comparing different encoding algorithms. We focus on comparing different implementations of two algorithms: transitive closure and compact. There is one deviation, however, for interval sets, where we used the results of the more complicated algorithm described in [1]. Although the underlying information is the same, the resulting interval sets are more compact (at the cost of more encoding effort).

Chains: Integers are well suited for encoding chains. Thus, sparse terms (using integer sorts), integer vectors and interval sets provide optimal encodings. However, bit-vectors require linear space. Since every element is meet irreducible, bit-vectors using the compact encoding algorithm also require linear space. Figure 6.6 shows a sparse term encoding for a chain.

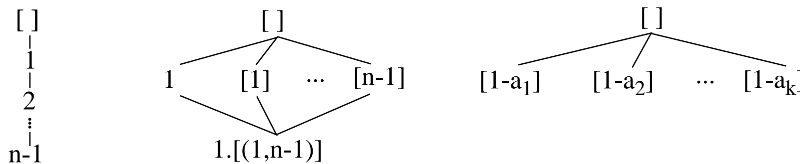

Figure 6.6: Chain and anti-chain encodings

Anti-chains: Terms and interval sets optimally encode anti-chains. Bit-vectors and integer vectors, however, require linear space. Figure 6.6 shows a sparse term encoding for an anti-chain. The second anti-chain encoding shows how $\perp$ could be encoded as unification failure using atomic functors.

Complete Binary Trees: In this case, the combination of integer sorts and logical terms permits optimal encoding using sparse terms (linear with respect to the height of the tree). Integer vectors and sparse terms without integer sorts both require linear code space, as do bit-vectors. With additional processing, bit-vectors can achieve optimal code size, using modulation or other techniques [2, 24, 49]. Figure 6.7 shows a sparse term encoding for a complete binary tree.

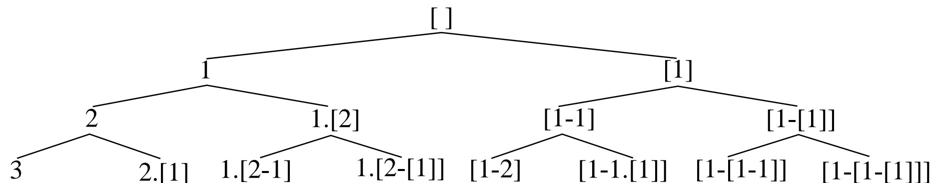

Figure 6.7: Binary tree encoding

If we invert the tree, and add a top element, the space requirement for sparse terms, bit-vectors and integer vectors does not change, but interval sets require $O((log n)^2)$ space.

For arbitrary binary trees, the code size for sparse terms remains linear with respect to the tree height. The worst-case occurs for a right-skewed binary tree (i.e. where the left branch is always a leaf), where the height is asymptotically the same as n. However, all of the other implementations require linear code space, except for interval sets which is optimal using the more complex algorithm. Also, if the tree is flipped left-right, then sparse terms achieve optimal encoding. In general, due to the use integer sorts, sparse terms will perform better if trees are organized so that the leftmost branch of a node has the largest subtree. In case two children have the same size subtree, the deepest should be selected as the leftmost. These selection criteria are closely related to those used in the interval sets approach [1].

For complete k-ary trees, bit and integer vectors remain linear. However, if the tree has height h, then sparse terms require $O(hlogk)$. Since $h \leq log_k n$ this is bounded above by $O(log_k n * logk)$.

Square Lattices: A *square lattice* is a partial order resulting from the product of two chains. An example is shown in Figure 6.8. For two chains of length k, their product has $n = k^2$ elements.

Transitive closure bit-vectors require linear space. Integer vectors, interval sets and sparse terms require $O(\sqrt{n}log(\sqrt{n})) = O(\sqrt{n}logn)$ which is sublinear, although not optimal. This is primarily because the square lattice has width $k = \sqrt{n}$. If additional work is performed to determine that this lattice is a chain product, then space can be improved to $O(2log\sqrt{n}) = O(logn)$. In general, however, finding the minimum number of chains that decompose a partial order is NP-Hard[144].

For compact encoding, there are $2k = 2\sqrt{n}$ meet irreducible elements. Thus, bit vectors require $O(\sqrt{n})$. Compact encoding for sparse terms, integer vectors and interval sets, however, achieve optimal codes. Figure 6.8 shows a transitive closure and compact sparse term encoding for a square lattice.

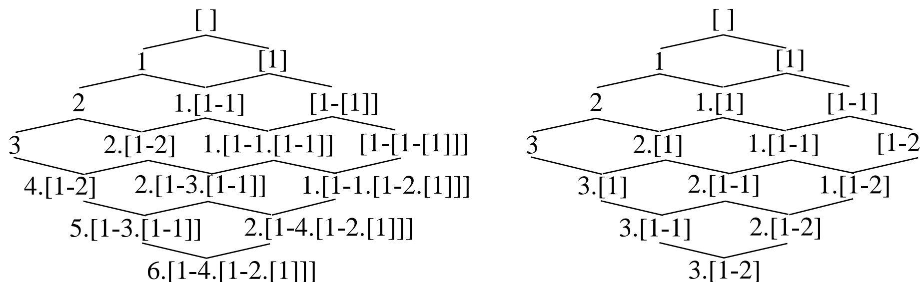

Figure 6.8: Square lattice transitive closure and compact encodings

Consider a product of m chains of length k each (so $n = k^m$). Optimally, if we have an algorithm that can decompose this order, integer vectors require $O(mlogk)$. However, using the transitive closure algorithm, we can only detect that the width of the order is $k^{m-1} = n^{\frac{m-1}{m}}$. Thus, integer vectors, interval sets and sparse terms require $O(k^{m-1}logk) = O(n^{\frac{m-1}{m}}\frac{1}{m}logn)$ which is still sublinear. Using the compact algorithm, we again obtain optimal results.

Generalized Crowns: The preceding example orders are all somewhat sparse (and of low dimension[144]). In lattice theory, generalized crowns are the standard example used for minimal sized partial orders of high dimension. Figure 6.9 shows the generalized crown S_5 of dimension 5. An important property of such orders, is that the minimal size lattice into which the generalized crown S_n of $2n$ elements can be embedded has 2^n elements.

Determining compact encodings for the generalized crown S_n is a challenge. Bit-vectors and integer vectors both require linear space, even for the compact algorithm. Note that even if we can determine the dimension (which is NP-Hard), we cannot improve on these results. However, interval sets and sparse terms can encode S_n using optimal space (also shown in Figure 6.9).

Table 6.1 summarizes these results, where n is the number of elements in the ordered set. Unless indicated, results are for both transitive closure and compact algorithms. Also, recall that the results for interval sets are somewhat

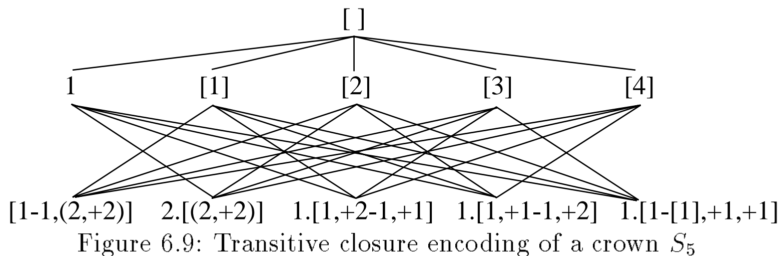

Figure 6.9: Transitive closure encoding of a crown S_5

biased as they are based on the more complex algorithm in [1]; using this algorithm, sparse terms can match or surpass these results, since they generalize interval sets.

Table 6.1: Asymptotic encoding results for theoretical orders

	Sparse Terms	Bit-Vector	Integer Vector	Interval Sets
Chains	$logn$	n	$logn$	$logn$
Anti-Chains	$logn$	n	n	$logn$
Complete Binary Tree	$logn$	n	n	$logn$
(inverted)	$logn$	n	n	$(logn)^2$
Arbitrary Binary Tree	n	n	n	$logn$
Square Lattice				
(transitive closure)	$n^{1/2}logn$	n	$n^{1/2}logn$	$n^{1/2}logn$
(compact)	$logn$	$\sqrt{n}$	$logn$	$logn$
Product of m chains				
(transitive closure)	$n^{\frac{m-1}{m}} * \frac{1}{m}logn$	n	$n^{\frac{m-1}{m}} * \frac{1}{m}logn$	$n^{\frac{m-1}{m}} * \frac{1}{m}logn$
(compact)	$logn$	$k * \sqrt[k]{n}$	$logn$	$logn$
Crown	$logn$	n	n	$logn$

6.7 Empirical Evidence

The above clearly shows the power of sparse terms. However, the partial orders likely to occur in practice are unlikely to possess any of the above forms. Intuitively, a large partial order will probably have some regions that are very sparse while others that are dense; some regions may possess certain properties, while others possess different properties. One technique that can be used to encode such hierarchies is modulation [2, 49], which decomposes a partial order into suborders that can be independently encoded. Modulation can be a powerful technique provided the order is not too dense. Although we generalized modulation to handle denser orders in Chapter 5, and a linear modulation algorithm now exists [76], it may not be appropriate for all dynamic taxonomies.

To demonstrate the power of sparse terms, we encoded two large empirically obtained taxonomies, using transitive closure and compact algorithms. The resulting sparse terms were not optimized in the sense that no relative or interval indices were used. Also, for the compact encoding, no integer sorts were used - this accounts for poorer behaviour in some cases compared with the transitive closure algorithm. If integer sorts are incorporated, more dramatic results may be achieved. We show the resulting space requirement of the encodings, as well as the required space for bit-vector encodings. Here too, the results are skewed against sparse terms. The sparse term space requirement was the actual memory used to store all codes; for bit-vectors, however, the space requirement does not consider memory padding. Still, the improvement that sparse terms offer over bit-vectors is remarkable.

The first taxonomy was obtained from a chess learning program [95], in which each node is a board position. There are 1,815 nodes (590 meet irreducible elements and 1,425 join irreducibles) and 8,227 links in the transitive reduction. As shown in Table 6.2, sparse terms require one quarter of the space for bit-vectors in the top-down transitive closure algorithm, and three quarters for the compact algorithm. Similar space improvements are made for the bottom-up algorithms. Thus, we not only gain the improved flexibility of sparse terms over bit-vectors, but this shows that even for moderate size taxonomies, the asymptotic advantage of sparse terms pays off.

Table 6.2: Empirical results (in bits) for chess learning system [16]

	Top-Down Trans. Closure	Top-Down Compact	Bottom-Up Trans. Closure	Bottom-Up Compact
Bit-Vectors total	3,294,225	1,070,850	3,294,225	2,586,375
bits/code	1,815	590	1,815	1,425
Sparse Terms total	820,872	803,056	966,920	1,007,104
bits/code	452	442	533	555
Sparse Term/ Bit-Vector ratio	0.25	0.75	0.29	0.39

The second taxonomy was obtained from a terminological medical knowledge base[1]. Nodes are medical terms, and the partial order is subsorting. There are 2,717 terms (2,640 meet irreducible elements and 2,187 join irreducibles), and 4,766 links in the transitive reduction. This taxonomy is less dense than the previous one (more nodes, less links), and most of the elements are irreducible. In this situation, compact encoding provides very little benefit for the additional cost. However, the benefits of sparse term encoding are even more marked: about 10 times more efficient than bit-vectors.

Table 6.3: Empirical results (in bits) for medical ontology

	Top-Down Trans. Closure	Top-Down Compact	Bottom-Up Trans. Closure	Bottom-Up Compact
Bit-Vectors total	7,382,089	7,172,880	7,382,089	5,942,079
bits/code	2,717	2,640	2,717	2,187
Sparse Terms total	690,432	812,768	812,064	812,064
bits/code	254	299	299	299
Sparse Term/ Bit-Vector ratio	0.09	0.11	0.11	0.14

6.8 Conclusion

Our goal in this chapter is twofold. First, we presented sparse terms as a universal implementation for encoding, generalizing the basic form of sparse terms [51] and extending previous work on logical term encoding [35]. Second, we argued that for large dynamic taxonomies, simple and fast encoding algorithms are necessary. These two claims are backed up by theoretical and empirical evidence. Furthermore, either claim could be taken independently. In particular, sparse terms could be exploited in any encoding algorithm with a potentially large decrease in space. Finally, although logical term encoding has been extensively studied [35, 43, 47, 102], this chapter presents the first published description of algorithms for encoding with terms. The results presented are important in contexts such as conceptual structures, where taxonomic knowledge is likely to change frequently.

[1] Thanks to Ian Horrocks, Medical Informatics Group at the Univ. of Manchester.

Part II:

Applications and Extensions

of

Reasoning with Taxonomies

"Then he was told: Remember what you have seen,
because everything forgotten returns to the circling winds"

– Lines from a Navajo chant

Chapter 7

Extending Partial Orders for Sort Reasoning

"Reason, alas, does not move mountains. It only tries to walk around them and see what is on the other side"

– G. W. Russel

The mathematical basis of partial orders has been exploited in taxonomic knowledge representation and reasoning, and research on taxonomic encoding has provided techniques for the efficient management of partial orders. Unfortunately, the simple structure of a partial order limits the taxonomic knowledge that can be represented. At the other extreme are description logics (e.g. the KL-ONE family [19, 159]) in which taxonomic relationships among sorts are specified using a formal language, but the taxonomy itself must be derived through *classification* (which may or may not be NP-Hard, depending on the logic). We feel that explicit maintenance of a taxonomy is important for efficiency. In this chapter, we formally extend partial orders to permit incorporation of additional taxonomic information.

7.1 Introduction

Research on integrating additional forms of taxonomic knowledge into partial orders is scarce. Most notable, work by Cohn [28] proposed a generalized form of taxonomic specification within a sorted-logic framework. In [53] we proposed some extensions to partial orders to integrate machine learning [103] and systemic classification [20, 101]. We extend these proposals in this chapter in an attempt to develop a taxonomic knowledge representation system that is both flexible and parsimonious.

We may wish, for example, to define an element to be the intersection (union) of another set of elements (e.g. $woman = human \cap female$). Although this may hold coincidentally through meets (joins), such a restriction ensures that any changes must also respect this constraint. As another example, every element in a taxonomy must normally be specified, but there may be cases when this is both unnecessary and inefficient. Suppose we wish, e.g., to view people along lines of religion (e.g. Catholic, Jewish, Muslim, etc.), nationality (e.g. Canadian, Belgian) and occupation (e.g. student, prof, miner). Currently, we need to specify all possible combinations (i.e. the *cross-product*) of these facets to produce all sorts of people (e.g. a Belgian Catholic student). It would be cleaner if we could specify these lines separately, and infer the cross-product when needed.

After providing some background on sorted logic and sorted logic programming, we formalize sorts and sort hierarchies, and identify the relation between lattice and set operations. We then propose the *sort reasoning problem* as the fundamental problem for a sort reasoner, and discuss how sort relations can be specified in two expressive, but equivalent ways. In section 7.4 we develop a three-valued propositional logic for sort reasoning and introduce the notion of a *sort context*. Using this logic, we show that, although resolution provides a sound and complete mechanism for sort reasoning, it is NP-Complete. The focus of section 7.5 is to identify tractable subcases of sort reasoning. Finally, we discuss some implementation issues.

7.2 Background

First-order logic is *unsorted* in the sense that the domain of discourse (i.e. the universe) is treated as a single undivided set. A *sort* can be viewed as a subset of the domain of discourse, and is generally a group of objects related in some way (e.g. the set of *dogs*). Sorts can be mimicked using special sort predicates, but *many sorted* logics move sorts into the forefront as first-class objects. This allows specification of the non-logical symbols as belonging to certain sorts, and provides a simple syntactic mechanism to state semantic constraints. Thus, in a many-sorted logic, a set of sorts can be specified that divide the domain of discourse. Although in some logics, sorts must be disjoint, most permit overlap between sorts, in which case the subset relation forms an order on sorts.

There are a number of advantages to using sorts in logic, particularly the reduction in the length of certain proofs by eliminating futile branches of the search space. See [27] for specific coverage of the benefits of many-sorted logic.

Sorted logic programming is simply the logic programming analog to sorted logic. Prolog is unsorted, and so the unification to two unequal atoms results in failure. LIFE [4], on the other hand, permits the specification of a sort hierarchy P. In the event of unification of unequal atoms a_1 and a_2, the sort hierarchy is used to determine the result. If $a_1 \sqcap_P a_2 = \bot$ then failure results. If $a_1 \sqcap_P a_2 = b$, then the result of the unification is b. Since the sort hierarchy does not need to be a lattice, $a_1 \sqcap_P a_2$ may be $\{b_1, b_2, \ldots, b_k\}$. In this case, processing proceeds with the result b_1, and subsequent sorts from this set are attempted in turn on backtracking.

7.3 Sort Reasoning

Sorts represent sets of individuals grouped according to common features. Intuitively, a sort p_1 is a *subsort* of p_2 provided that every individual in p_1 is also in p_2 (e.g. *collie* is a subsort of *dog*). We don't require that sorts denote unique sets of individuals, so two sorts p_1 and p_2 may be *aliases* for the same set (e.g. *car* and *automobile*), or that a sort be non-empty (e.g. *unicorn* is an empty sort). As we describe below, subset information on sets of aliases forms a partial order.

- Let $\mathcal{U}$ be the domain of discourse (i.e. the set of individuals).
- Let $\mathcal{P}$ be a set of *base* sorts, notated using letters p and q. $\forall p \in \mathcal{P}$, p represents a subset of $\mathcal{U}$. $\mathcal{P}$ contains an implicit element: $\top_{\mathcal{P}}$, representing $\mathcal{U}$.
- Then $\subseteq$ forms a preorder relation on $\mathcal{P}$ (i.e. $\subseteq$ is reflexive and transitive).

From $\mathcal{P}$ we can specify the *literal* sorts: $\mathcal{P}_{\mathcal{L}} = \{p, \neg p | p \in \mathcal{P}, \neg p = \mathcal{U} \backslash p\}$, notated using greek letters α, β, etc. We can derive an implicit literal sort $\bot_{\mathcal{P}} = \neg \top_{\mathcal{P}}$ that represents $\emptyset$. We can also extract two relations:

- The *sort equivalence relation*, $=_{\mathcal{P}}$: for $p_1, p_2 \in \mathcal{P}$, $p_1 =_{\mathcal{P}} p_2$ if and only if $p_1 \subseteq p_2$ and $p_2 \subseteq p_1$. We denote the set of equivalence classes of $\mathcal{P}$ as $\mathcal{P}_=$, and each equivalence class as $[p]$, where p is a *representative* for the class.
- The *sort (partial) order*, $(\mathcal{P}_=, \leq_{\mathcal{P}})$: for $[p], [q] \in \mathcal{P}_=$, $[p] \leq_{\mathcal{P}} [q]$ if and only if $\forall p_i \in [p], q_j \in [q]$, $p_i \subseteq q_j$. Clearly $\leq_{\mathcal{P}}$ is reflexive and transitive. To show anti-symmetry, consider two classes $[p]$ and $[q]$. If $[p] \leq_{\mathcal{P}} [q]$ and $[q] \leq_{\mathcal{P}} [p]$, and $p_i \in [p], q_j \in [q]$, then $p_i \subseteq q_j$ and $q_j \subseteq p_i$. Thus, $p_i =_{\mathcal{P}} q_j$, so it must be the case that $[p] = [q]$.

For simplicity of notation, we omit the brackets surrounding alias classes. We now describe the relationship between taxonomic and set operations.

- If $p_1 \sqcap p_2 = p_3$, then $p_1 \cap p_2 \supseteq p_3$. For example, if $p_1 \sqcap p_2 = \bot$, we cannot infer that there is no element in $\mathcal{U}$ that is in both p_1 and p_2. We can only infer that there is no known sort that represents such elements. However, if we know that $p_1 \cap p_2 = p_3$ then we can infer $p_1 \sqcap p_2 = p_3$. For non-singleton meet crests, if $p_1 \sqcap p_2 = \{q_1, \cdots, q_k\}$, then $\forall q_i, 1 \leq i \leq k, p_1 \cap p_2 \supseteq q_i$.
- If $p_1 \sqcup p_2 = p_3$, then $p_1 \cup p_2 \subseteq p_3$. However, if we know that $p_1 \cup p_2 = p_3$ then we can infer $p_1 \sqcup p_2 = p_3$. For non-singleton join bases, if $p_1 \sqcup p_2 = \{q_1, \cdots, q_k\}$, then $\forall q_i, 1 \leq i \leq k, p_1 \cup p_2 \subseteq q_i$.

Thus, it is not always possible to perform sort inferences using taxonomic operations. This issue was the focus of the lattice completion proposed in [28]. Figure 7.1 shows the above relationships using Venn diagrams. Our goal is to exploit both the complete and incomplete knowledge in a sort hierarchy for a sort reasoning system. This requires a general means of specifying, maintaining and reasoning with information that relates sorts.

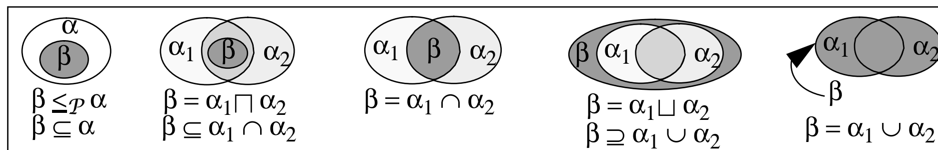

Figure 7.1: Relation between taxonomic and set operations

7.3.1 Generalizing sort reasoning

Definition 7.1 *Suppose we have a set $\mathcal{P}$ of n* base *sorts.*

- *An* atomic *sort is a sort s obtained by intersecting, for every sort $p \in \mathcal{P}$, either p or its complement $\neg p$.*
- *A* derived *sort is a set of atomic sorts.*
- *A* conjunctive *sort is the intersection (conjunction) of a set of literal sorts.*
- *A conjunctive sort s is* consistent *if and only if it does not contain both a base sort and its complement. A consistent conjunctive sort is a derived sort.*

In a Venn diagram of all possible combinations of sorts, each distinct region is an atomic sort of which there are 2^n. Taxonomic information may reduce the number of non-empty atomic sorts (e.g. if $p_1 \leq p_2$ then an atomic sort with p_1 but not p_2 is empty). A derived sort is obtained by selecting 0 or more atomic sorts, and corresponds to the union of distinct regions in a Venn diagram. In the worst case (no taxonomic constraints) there are 2^{2^n} non-empty derived sorts.

To illustrate, consider the specifications: (i) $francophone \leq person$ and (ii) $canadian \leq person$. Although sorts *francophone* and *canadian* are incomparable, there is no information that indicates they are disjoint. Combining them results in the derived sort *canadian_francophone*. In general, conjunctive sorts can be denoted by juxtaposing their constituent sort labels (lexicographically to ensure uniqueness, although any total order on the sort labels could be used). Automatic derivation of conjunctive sorts can be contrasted with LIFE in which the same combination will result in failure, since their coincidental meet is $\perp$.

For conjunctive sorts, we can specify an *intrinsic* ordering ($\preceq$): for two conjunctive sorts s_1 and s_2, we know that $s_1 \subseteq s_2$ if s_1 contains a superset of the literals in s_2. For example, $p_1 \wedge \neg p_2 \wedge p_3 \preceq \neg p_2 \wedge p_3$. Taxonomic information provides further *extrinsic* ordering among conjunctive sorts. Thus, for conjunctive sorts s_1 and s_2, $s_1 \preceq s_2$ implies that $s_1 \subseteq s_2$, but not necessarily the converse.

Clearly there is potential for a combinatorial explosion in the number and size of derived sorts. In [28], completeness in a many-sorted logic setting is required, and so the entire derived sort space must be handled. Unfortunately, this leads to the possibility of a sort structure of exponential size. Our goal is to produce a general sort reasoner that minimally retains polynomial space, and so we choose to restrict the set of derived sorts to conjunctive sorts.

Conjunctive sorts are natural in that they group together individuals in $\mathcal{U}$ that share attributes. They provide for monotonic sort reasoning, since the set of individuals denoted by a partially specified sort cannot increase as new constraints are applied. These are the types of sorts produced in LIFE [4] through unification. Conjunctive sorts have a natural representation using a three-valued logic by selecting for each base sort $p \in \mathcal{P}$ either *true* (include sort p), *false* (exclude sort p) or *uncertain*. Thus, there are at most 3^n different consistent conjunctive sorts, although constraints may reduce this number. Conjunctive sorts have a simple and efficient implementation using logical terms (see section 7.6).

Our problem can now be described succinctly as follows:

Definition 7.2 Sort Reasoning Problem (abstract): *Given a set of base sorts $\mathcal{P}$, a set of assertions $\mathcal{A}$ that specify the emptiness or non-emptiness of zero or more conjunctive sorts, and a conjunctive sort s. Can we infer that s is empty or non-empty?*

We show that interesting sort reasoning problems can be characterized as special cases of this problem, and we describe general methods of specifying the assertions. We develop a *sort logic* (not a sorted-logic, but a logic for sort reasoning) that has a sound and complete reasoning strategy. We also show that this problem is NP-Complete, so we explore tractable subsets of sort reasoning.

The assertions $\mathcal{A}$ partition the conjunctive sorts into three groups: *empty* sorts, *non-empty* sorts and *possibly empty* sorts. If a conjunctive sort s_1 is empty, and $s_2 \preceq s_1$, then s_2 must also be empty. Dually, if s_1 is non-empty, and $s_1 \preceq s_2$, then s_2 must be non-empty. Thus, sort reasoning can be viewed as classifying conjunctive sorts into these groups based on the current set of assertions.

7.3.2 Clausal taxonomic specification

In [28], a suggestion is made for clausal specification of taxonomies: $\forall x, p_1(x) \vee \cdots \vee p_m(x) \vee \neg q_1(x) \vee \cdots \vee \neg q_n(x)$, where the p_i and q_j are base sorts. A number of special cases are worth noting:

1. $m = 0, n = 2$: q_1 and q_2 are incompatible.
2. $m = 0, n > 2$: $q_1, \cdots, q_n$ cannot simultaneously hold.
3. $m = 1, n = 1$: $q_1 \subseteq p_1$.
4. $m > 1, n = 0$: $p_1, \cdots, p_m$ decompose $\top$ (i.e. $\bigcup\{p_1, \cdots, p_m\} = \top$).

The usefulness of these clausal specifications is not explored in [28]. In light of the sort reasoning problem, such a specification can be viewed as asserting that a certain conjunctive sort is empty. The universally quantified form is equivalent to $\nexists x, \neg p_1(x) \wedge \cdots \wedge \neg p_m(x) \wedge q_1(x) \wedge \cdots \wedge q_n(x)$ (i.e. conjunctive sort $\neg p_1 \wedge \cdots \wedge \neg p_m \wedge q_1 \wedge \cdots \wedge q_n$ is empty). We propose to also allow dual specifications: $\exists x, \neg p_1(x) \wedge \cdots \wedge \neg p_m(x) \wedge q_1(x) \wedge \cdots \wedge q_n(x)$, which permit asserting that a certain conjunctive sort is not empty. Duals of the above special cases are:

1. $m = 0, n = 2$: q_1 and q_2 are compatible.
2. $m = 0, n > 2$: $q_1, \cdots, q_n$ can simultaneously hold.
3. $m = 1, n = 1$: $q_1 \not\subseteq p_1$.
4. $m > 1, n = 0$: $p_1, \cdots, p_m$ do not decompose $\top$.

With these two forms, we have the ability to fully specify any instance of the sort reasoning problem, so we can dispense with the quantification, and limit our focus to propositional logic. Universally quantified assertions (or *universal sorts*) are global in that they must all simultaneously hold, but not existentially quantified assertions (or *existential sorts*), which may specify different individuals in $\mathcal{U}$. Figure 7.2 shows the set relationships imposed by these specifications.

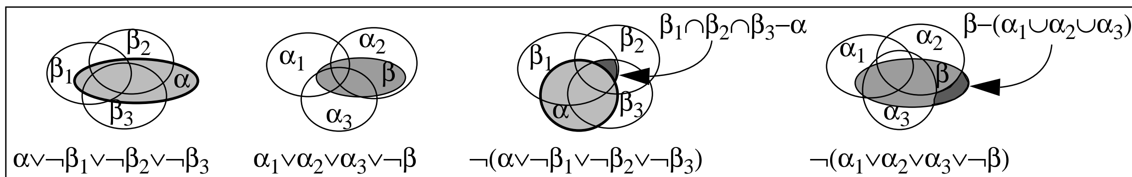

Figure 7.2: Venn diagrams of clausal taxonomy specification

7.3.3 Definitional specifications

As an alternative to clausal specifications, a number of natural relationships can be constructed using sort definitions. Some possibilities are described below and shown in Figure 7.3, and formed the basis of *extended description spaces* [53].

Conjoined Sort Definition: We may want to *define* a sort as precisely the intersection of a set of other sorts. For example, we may want to define *woman* as the intersection of *person* and *female*. We can denote this using set intersection: $p = \alpha_1 \cap \cdots \cap \alpha_k$, where the α_i are sort literals. Such definitions are equivalent to the clauses: (i) $p \vee \neg\alpha_1 \vee \cdots \vee \neg\alpha_k$; and (ii) $\neg p \vee \alpha_i$ for $1 \leq i \leq k$. Partial orders only permit the second set of clauses, and so we may only say: $p \subseteq \alpha_1 \cap \cdots \cap \alpha_k$.

Sort Decomposition: Sometimes we know that a set $\{\alpha_1, \cdots, \alpha_k\}$ of (possibly overlapping) sorts *decomposes* another sort p. That is, $p = \alpha_1 \cup \cdots \cup \alpha_k$. For example, we may wish to define a sort $university_course = grad_course \cup undergrad_course$ (where some courses may be cross-listed as both). Sort decomposition is analogous to generalization in the entity-relationship model [92]. Such a declaration is equivalent to the clausal specifications: (i) $\neg p \vee \alpha_1 \vee \cdots \vee \alpha_k$; and (ii) $p \vee \neg\alpha_i$, for $1 \leq i \leq k$. Every conjoined sort definition $p = \alpha_1 \cap \cdots \cap \alpha_k$ induces a dual sort decomposition $\neg p = \neg\alpha_1 \cup \cdots \cup \neg\alpha_k$, and vice versa.

Sort Partitioning: We may have even stronger information that a set Q decomposes a supersort p and every pair of elements in Q is disjoint. For example, we may want to say that the sort *person* is partitioned into *woman* and *man*. We can denote this using disjoint set union: $p = \alpha_1 + \cdots + \alpha_k$, where $+$ is interpreted as union with the constraint that each pair of sorts on the right-hand side must be disjoint. Such assertions are equivalent to the clauses: (i) $\neg p \vee \alpha_1 \vee \cdots \vee \alpha_k$; (ii) $p \vee \neg\alpha_i$ for $1 \leq i \leq k$; and (iii) $\neg\alpha_i \vee \neg\alpha_j$, for $1 \leq i < j \leq k$.

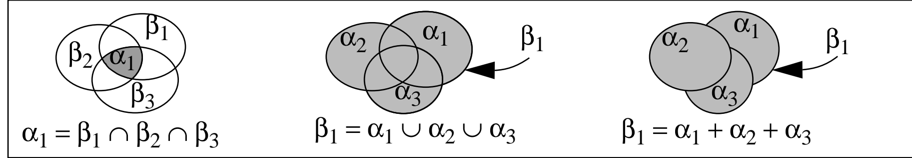

Figure 7.3: Aggregate specifications

We can specify the dual of these assertions, by replacing equal signs by strict subsets. We may, e.g., state that *wild* and *canine* is insufficient to define *wolf* as $wolf \subset wild \cap canine$ (i.e. the sort $\neg wolf \wedge wild \wedge canine$ is non-empty).

Interestingly, definitional and clausal specifications are equivalent. A universal assertion: $p_1 \vee \cdots \vee p_m \vee \neg q_1 \vee \cdots \vee \neg q_n$ can be specified as: (i) $q' = q_1 \cap \cdots \cap q_n$; (ii) $p' = p_1 \cup \cdots \cup p_m$; and (iii) $q' \cap p' = q'$ (or $q' \leq p'$). An existential assertion: $\neg(p_1 \vee \cdots \vee p_m \vee \neg q_1 \vee \cdots \vee \neg q_n)$ can be specified as: (i) $q' = q_1 \cap \cdots \cup q_n$; (ii) $p' = p_1 \cup \cdots \cup p_m$; and (iii) $q' \cap p' \subset q'$ (or $q' \not\leq p'$).

7.4 Sort Logic

Definition 7.3 *A* sort context *is a triple* $\Sigma = (\mathcal{P}, \mathcal{E}, \mathcal{N})$*, where*

- *$\mathcal{P}$ is a set of sort symbols, and $\mathcal{P}_\mathcal{L}$ is the corresponding set of sort literals.*
- *$\mathcal{E}$ is a set of universal sort assertions, where for every $\epsilon \in \mathcal{E}$, $\epsilon = \alpha_1 \vee \cdots \vee \alpha_k$ and each α_i, $1 \leq i \leq k$, is a sort literal. Conjunctive sort $\neg\epsilon$ is in the same sort equivalence class as $\bot_\mathcal{P}$ (i.e. $\neg\epsilon$ is an empty sort).*
- *$\mathcal{N}$ is a set of existential sort assertions, for every $\eta \in \mathcal{N}$, $\eta = \alpha_1 \wedge \cdots \wedge \alpha_k$ and each α_i, $1 \leq i \leq k$, is a sort literal. Conjunctive sort η is in a different sort equivalence class from $\bot_\mathcal{P}$ (i.e. η is a non-empty sort).*

Since existential sort clauses are local (i.e. they implicitly existentially quantify an individual), we cannot use them indiscriminately: we only allow at most one to appear in a proof. Our sort logic has three truth values: T (*true*), F (*false*) and U (*unknown* or *uncertain*). For example, the answer to the query $dog \wedge cat = \emptyset$? may be *true*, whereas the answer to the query $student \wedge plumber = \emptyset$? may be *uncertain*. We also have one rule of inference, resolution, which we can formalize as follows (where the α_i and β_j are sort literals, and $\neg\neg p = p$):

$$(\gamma \vee \alpha_1 \vee \cdots \vee \alpha_j) \wedge (\neg\gamma \vee \beta_1 \vee \cdots \vee \beta_k) \vdash \alpha_1 \vee \cdots \vee \alpha_j \vee \beta_1 \vee \cdots \vee \beta_k$$

Using a standard resolution process, we finish when either the empty clause is derived, or no more resolution is applicable. The empty clause is derived only if both α and $\neg\alpha$ can be derived, which clearly indicates inconsistency.

A sort context Σ is *consistent* if for every conjunctive sort s resulting from $\mathcal{P}_{\mathcal{L}}$, we cannot infer that s is both empty and non-empty. Since resolution is sound and *refutation complete* [72], determining if a sort context is inconsistent using resolution is sound and complete. We do not assume complete knowledge, however, so it may be the case that we cannot infer that s is empty or non-empty. In this case, following Cohn [28], we call s *possibly-empty*.

Queries can be dealt with as follows:

Empty Sorts: To check if a conjunctive sort $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ is empty, we assert that it is not empty by adding s as an existential sort, and attempt to derive the empty clause through resolution. If we derive the empty clause, then s must be empty, and $\neg s$ must be a universal sort (i.e. the sort context $(\mathcal{P}, \mathcal{E}, \{s\})$ is inconsistent). If not, then s may be either non-empty or possibly-empty. Note that we only use elements of $\mathcal{E}$, but not of $\mathcal{N}$, for this.

Inferring Sorts: We may be interested in the sorts that can be inferred from s. These can be produced as a side product of the above resolution process. If s is an empty sort, then every sort is derivable.

Non-empty Sorts: To check if s is non-empty, we assert that it is empty (i.e. add $\neg s$ as a universal sort), and attempt to derive the empty clause through resolution. We do this by finding a non-empty sort $\eta \in \mathcal{N}$ with which we can derive the empty sort (i.e. the sort context $(\mathcal{P}, \mathcal{E} \cup \{\neg s\}, \{\eta\})$ is inconsistent). Note that this is akin to *skolemizing* the existential sort η.

We can now restate the sort reasoning problem in more definite terms.

Definition 7.4 Sort Reasoning Problem (concrete): *Given a sort context* $\Sigma = (\mathcal{P}, \mathcal{E}, \{s\})$. *Is* Σ *consistent?*

The Sort Reasoning Problem is NP-Complete, as we prove formally in the following subsection. This can be demonstrated by modeling an instance of 3-SAT using sort definitions, as shown in Figure 7.4, where a conjunctive normal form formula with ternary clauses $f = c_1 \wedge \cdots \wedge c_k$, where $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, $1 \leq i \leq k$ can be represented using one intersection definition for f and one union definition for each of the clauses. In diagrams, we denote intersection (resp. union) definitions by connecting the parent (resp. child) subsumption arcs with a horizontal line. Answering the query "Is f an empty sort?" is clearly NP-Complete.

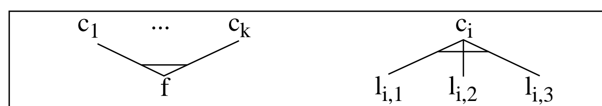

Figure 7.4: Using sort definitions to represent an instance of 3-SAT: $f = c_1 \wedge \cdots \wedge c_k$, where $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, $1 \leq i \leq k$

From a logical standpoint, intractability is of no concern, provided the logic is sound and complete. Also, some systems may prefer to retain expressiveness and assume that the worst-case will rarely, if ever, occur. Even so, there is some sort structure maintenance that we may perform to reduce the cost of sort reasoning. If we determine that a sort s is empty or non-empty, then we can assert this information in the sort context. We refer to this as *sort memoing*, since it is akin to memoing in OLDT resolution [125]. If sort reasoning is performed in localized areas of the sort structure, then this enhancement may result in improved performance at the cost of additional storage (in the worst-case, one conjunctive sort is added to the context for any query).

7.4.1 Complexity of Sort Reasoning

We now prove that sort reasoning is NP-Complete. Note that context $\Sigma = (\mathcal{P}, \mathcal{E}, \{s\})$ is consistent if and only if s is not provably empty.

Lemma 7.1 *If s is an empty conjunctive sort and s' contains a superset of the literals of s (i.e. $s' \preceq s$), then sort resolution can show that s' is empty.*

Proof: Suppose s is an empty sort: $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ (so $\neg\alpha_1 \vee \cdots \vee \neg\alpha_k$ is a universal sort), and s' contains a superset of the components of s: $s' = \alpha_1 \wedge \cdots \wedge \alpha_k \wedge \beta_1 \wedge \ldots \wedge \beta_j$. Further suppose that s' is not empty: assert $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_j$. Clearly, through resolution we can derive the empty clause. Thus, sort resolution can show that s' must be empty. □

Lemma 7.2 *If s is an atomic sort (i.e. $s = \alpha_1 \wedge \cdots \wedge \alpha_n$), then s is provably empty if and only if $\exists \neg s' \in \mathcal{E}$ for which $s \preceq s'$.*

Proof: $\Rightarrow$ Suppose $\nexists \neg s' \in \mathcal{E}$ for which $s \preceq s'$. The only way to infer that s may be empty from $\mathcal{E}$ is to find a decomposition of s, each element of which is provably empty. But since s is atomic, no decompositions exist.

$\Leftarrow$ Suppose $\exists \neg s' \in \mathcal{E}$ for which $s \preceq s'$. By Lemma 7.1, clearly s is provably empty. □

Theorem 7.1 *The Sort Reasoning Problem is NP-Complete.*

Proof: Given a conjunctive sort s, if s is not provably empty, then there exists an atomic sort s' subsumed by s that is not provably empty. By Lemma 7.2, checking if s' is not provably empty and checking if $s' \preceq s$ can both be done in polynomial time. Thus, the sort reasoning problem is in NP.

To show that this problem is NP-Complete, we show a transformation to sort reasoning from 3-SAT [69]. The 3-SAT problem can be specified as follows: Given a set of n variables $v_1, \cdots, v_n$ and a formula F that is a conjunction of k clauses, each of which is a disjunction of precisely 3 literals, is there a truth assignment to the variables for which F is true?

Suppose we have an instance of the 3-SAT problem: $V = \{v_1, \cdots, v_n\}$, $F = C_1 \wedge \cdots \wedge C_k$ and $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$, $1 \leq i \leq k$, where each of the $l_{i,j}$ is either a positive or negated variable from V. Let us define a sort context trivially as $\Sigma = (V \cup \{q\}, \{q, C_1, \cdots, C_k\}, \emptyset)$. Clearly this can be done in polynomial time. Note that the sort q must subsume all the other sorts (i.e. it is in the same sort equivalence class as $\top$). Each atomic sort corresponds to a truth assignment.

Claim: there is a solution to the 3-SAT problem if and only if we cannot infer that q is empty.

$\Rightarrow$ Suppose formula F is satisfiable. Take any satisfying truth assignment, and define an atomic sort s as: $s = \alpha_1 \wedge \cdots \wedge \alpha_n$, where $\alpha_i = v_i$, if $v_i = true$ and $\alpha_i = \neg v_i$ otherwise (for $1 \leq i \leq n$). If s is provably empty, then $\exists$ a clause $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ for which $\neg C_i = \neg l_{i,1} \wedge \neg l_{i,2} \wedge \neg l_{i,3}$ subsumes s by Lemma 7.2. But at least one of $l_{i,1}, l_{i,2}, l_{i,3}$ is $true$, so no such clause exists. Therefore, s is not provably empty, which implies that q is not provably empty. So, if F is satisfiable then q is not provably empty.

$\Leftarrow$ Suppose that q is not provably empty. Then $\exists$ an atomic sort s that is not provably empty. Define a truth assignment as follows: if v_i is a component of s then set $v_i = true$ and if $\neg v_i$ is a component of s then set $v_i = false$. Consider any clause $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ for which none of the literals are true. Then $\neg l_{i,1}$, $\neg l_{i,2}$ and $\neg l_{i,3}$ are all components of s. But then s must be empty, so no such clause exists, and this truth assignment satisfies F. So, if q is not provably empty then F is satisfiable. □

7.5 Tractable subcases

Many knowledge representation systems are concerned with tractable reasoning strategies, so it is important to identify subcases of the sort reasoning problem with polynomial solutions. As intractability results from empty sort assertions (i.e. universal sorts) and queries, there is no need to restrict the form of non-empty sort assertions.

Positive literal sorts. A simple way to achieve tractability is to avoid negated sorts by only allowing assertions that involve positive literals. In LIFE [4], only subsumption (i.e. $p \leq q$) assertions are permitted in specifying a sort hierarchy. However, if the meet crest $p_1 \sqcap \cdots \sqcap p_k$ happens to be $\{q_1, \ldots, q_n\}$, there is an implicit assertion of the form $p_1 \wedge \cdots \wedge p_m = q_1 \vee \cdots \vee q_n$.

Horn sorts. Another possibility is to restrict specification to Horn clauses (clauses with at most one positive literal). This leads to tractable resolution if we restrict each base sort to be a positive literal of at most one clause. This restriction may be relaxed somewhat using the notion of *ORD-Horn clauses* described in [113] for finding a maximal tractable subclass of Allen's Interval Algebra [7] for temporal reasoning.

7.5.1 Containing sort reasoning complexity

Both cases above impose unnecessarily strict limitations on the expression of taxonomic knowledge. To achieve more flexibility while retaining tractability, we can either restrict the form of assertions or the form of queries. We choose a combination. The basic form of universal sort assertions we allow are (i) binary clauses, which can define a partial order among the literal sorts (i.e. $p \vee q, p \vee \neg q$ or $\neg p \vee \neg q$); (ii) intersection (conjoined sort) definitions: $p = \alpha_1 \wedge \cdots \wedge \alpha_m$; and (iii) union definitions (sort decomposition): $p = \alpha_1 \vee \cdots \vee \alpha_m$.

Sort contexts can be described as $(\mathcal{P}, \mathcal{A}, \mathcal{N})$, where $\mathcal{A}$ is a set of definitional assertions that satisfies the above forms. Such assertions could be reduced to clausal form, but these definitional assertions can be maintained in a partial order structure on the literal sorts, augmented with notation for the intersection and union definitions. $\mathcal{N}$ is a set of existential conjunctive sort assertions as before.

Note that asserting a binary clause imposes two constraints: $\alpha \vee \beta$ asserts $\neg\alpha \leq \beta$ and $\neg\beta \leq \alpha$. Asserting an intersection or union definition, also asserts the dual. The intersection definition, $p = \alpha_1 \wedge \cdots \wedge \alpha_m$ also asserts $\neg p = \neg\alpha_1 \vee \cdots \vee \neg\alpha_m$. The union definition $p = \alpha_1 \vee \cdots \vee \alpha_m$ also asserts $\neg p = \neg\alpha_1 \wedge \cdots \wedge \neg\alpha_m$.

Without restrictions, of course, we have full sort reasoning power with the above assertion forms. Even limiting sorts to have at most one definition may lead to intractable behaviour, as shown in Figure 7.4. Our solution is to limit the extent of intractability. First we need to define several notions.

Definition 7.5 *Let $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ be a conjunctive sort. The* expanded *form s^* of s is the fixpoint of the following construction (i.e. there exists a $k \geq 0$ for which $s_{k+1} = s_k = s^*$): (i) $s_0 = \{\alpha_1, \ldots, \alpha_k\}$; (ii) $s_{i+1} = s_i \cup \{\beta \in \mathcal{P}_\mathcal{L} | \exists\gamma \in s_i$ such that $\gamma \leq \beta\} \cup \{\beta \in \mathcal{P}_\mathcal{L} | \beta = \gamma_1 \wedge \cdots \wedge \gamma_m$ is an assertion in $\mathcal{A}$ and $\gamma_j \in s_i, 1 \leq j \leq m\}$*

Thus, given a conjunctive sort s, its expanded form is the set of all sort literals that may be directly inferred from s.

Definition 7.6 *Let $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ be a conjunctive sort, and s^* be its expanded form. The set of* potential conjunctive inferences *$\mathcal{C}(s)$ associated with s is defined recursively as the fixpoint of the following construction (i.e. there exists a $k \geq 0$ for which $s_{k+1} = s_k = \mathcal{C}(s)$): (i) $s_0 = s^*$; (ii) $s_{i+1} = s_i \cup \{\beta | \beta = \gamma_1 \wedge \cdots \wedge \gamma_m$ is an assertion in $\mathcal{A}$, and $\gamma_j \in s_i$ for some $1 \leq j \leq m\}$*

Definition 7.7 *Let $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ be a conjunctive sort, and s^* be its expanded form. The set of* unresolved disjunctions *$\mathcal{D}(s)$ associated with s is defined as: $\mathcal{D}(s) = \{\{\beta_1 \vee \cdots \vee \beta_k\} | (i)\alpha = \beta_1 \vee \cdots \vee \beta_k$ is an assertion in $\mathcal{A}$; (ii) $\alpha \in s^*$; and $(iii) \nexists\gamma \in s^*$ such that $\gamma \leq \beta_i$ for some $i, 1 \leq i \leq k\}$.*

Thus, $\mathcal{D}(s)$ is the set of union definitions for which the left-hand side sort, but none of the right-hand side sorts, is in s^* (so the disjunction is implied but not satisfied by s).

Definition 7.8 *Let $s = \alpha_1 \wedge \cdots \wedge \alpha_k$ be a conjunctive sort. A* locally consistent *selection of literals from the unresolved disjunctions $\mathcal{D}(s)$ is a set $Q = \{\beta_1, \ldots, \beta_m\}$ of at least one sort literal from each disjunction in $\mathcal{D}(s)$, where the expanded sort s_1^* ($s_1 = \alpha_1 \wedge \cdots \wedge \alpha_k \wedge \beta_1 \cdots \wedge \beta_m$) is consistent.*

The existence of a locally consistent selection is necessary but not sufficient to show that sort s is not provably empty. Unresolved disjunctions may *cascade* due to a locally consistent selection - $\mathcal{D}(s_1)$ may contain unresolved disjunctions.

In order to determine if s is provably empty or not (provided s^* is consistent), we need to show that every possible way of resolving the set of disjunctions $\mathcal{D}(s)$ leads to inconsistency. This problem may be intractable in two dimensions. First, even making a locally consistent selection from $\mathcal{D}(s)$ may be NP-Complete (cfr. 3-SAT problem). Second, the potential cascading effect of unresolved disjunctions may lead to an exponential search space, even if determining locally consistent selections can be done in polynomial time. The following set of restrictions attempts to curtail both of these sources of intractability, while retaining a degree of power that makes sort reasoning useful:

1. Positive literal sorts may not subsume negative literal sorts, and no set containing negative literals may imply a positive literal. This is achieved by enforcing the following syntactic constraints on assertions: (i) Subsumption assertions must have the form $p \vee \neg q$ (i.e. $q \leq p$ and $\neg p \leq \neg q$) or $\neg p \vee \neg q$ (i.e. $p \leq \neg q$ and $q \leq \neg p$); (ii) The sorts on the right-hand side of intersection and union definitions must be positive literals.

2. For a given conjunctive sort $s = \alpha_1 \wedge \cdots \wedge \alpha_k$, limit the number of unresolved disjunctions (union definitions) containing positive literals associated with s to a constant $n_\cup$. This ensures that we can determine in polynomial time if there is a locally consistent selection of literals from the unresolved disjunctions $\mathcal{D}(s)$. If $\mathcal{D}(s)$ is empty or contains only disjunctions with negative literals, then a locally consistent selection can be done in linear time.

3. Limit the cascade of unresolved disjunctions by imposing constraints on the relation of positive sorts involved in one union definition $p = q_1 \vee \cdots \vee q_k$ to other union definitions. If $s_i \in \mathcal{C}(q_i)$, $1 \leq i \leq k$, then $\mathcal{D}(s_i)$ can only contain disjunctions with negative literals. Note that if q_i is not subsumed by any sorts on the right-hand side of an intersection definition, then this reduces to the constraint: $\mathcal{D}(q_i)$ can only contain disjunctions with negative literals. This restriction ensures that, for a conjunctive sort s, any locally consistent selection from $\mathcal{D}(s)$ can be checked for global consistency in polynomial time since cascading disjunctions can only contain negative literals (and no selection of negative literals can result in a positive literal being derived).

The first and third restrictions are purely syntactic. The second affects both assertions (i.e. the conjunctive sorts on the right-hand side of intersection definitions) and queries, and depends largely on the current sort structure. It can, however, be checked quickly given any conjunctive sort. If it is not satisfied in a query, we can notify the client and provide the option to attempt a potentially costly answer. Together these restrictions permit us to specify a polynomial time algorithm for determining if a conjunctive sort s is provably empty:

i. Construct s^*. If s^* is inconsistent then s is provably empty.
ii. Determine $\mathcal{D}(s)$ and check if there is a locally consistent selection. If none exists, then s is provably empty.
iii. Attempt to expand each locally consistent selection to a globally consistent selection. If this is not possible, then s is provably empty.

The first step of the algorithm is performed automatically and efficiently using lattice operations and the logical term implementation described in the next section. Due to the second restriction above, step (ii) can be accomplished in polynomial time, and due to the third restriction, checking if there exists at least one globally consistent selection (in which case s is not provably empty) also takes polynomial time.

7.6 Implementing Conjunctive Sorts

For a simple logical term encoding of sort orders, that is fast to compute and flexible to update, we assign terms in which each element has one position and use a variant of top-down transitive closure encoding [2]. For any element $p \in \mathcal{P}$, position i of the code $\tau(p)$ may have one of three values: (i) If $p \leq p_i$ then position i will contain a 1; (ii) If $p \leq \neg p_i$ then position i will contain a 0; (iii) Otherwise position i will contain an anonymous variable (denoted "_").

We can extend our logic and implementation to four values: *true* (1), *false* (0), *uncertain* (_) and *inconsistent* (!). Inconsistency in a sort position could be used as an explanatory feature to identify the base sort at the root of an inconsistency. It could also be used as a basis for extending our sort logic to include default and non-monotonic reasoning - an inconsistent value for a base sort p would indicate that somehow both p and $\neg p$ have been acquired. Our approach does not provide a means of resolving this inconsistency, but does give a framework upon which a default or non-monotonic logic system can be built.

7.7 Conclusion

Taxonomic knowledge representation is a complex, yet intuitive and pervasive problem. By separating sort constraints into a sort reasoner, specialized techniques can be used to manage the sort relations arising in a system. We argued that, although mathematically elegant, partial orders are unwieldy for representing all the relations desired in a system. Although sort reasoning can be plunged into a partial order (in fact, a Boolean lattice), the size of this partial order is extraordinary - given n base sorts, the lattice can be as large as 2^{2^n}. The typical use of partial orders for sort reasoning, in which each base sort is an atom (i.e. plunging the sort structure in a Boolean lattice of size 2^n), leads to either the inability to state certain relations (e.g. sort *woman* is the intersection of sorts *person* and *female*) or to unjustifiable conclusions.

We extended partial orders to more efficiently handle sort processing. By restricting attention to *conjunctive sorts* (sorts that consist of conjunctions of positive and negative base sorts), the scope of the problem is reduced to the interesting case that is most apparent in current logic programming systems (e.g. LIFE [4]). We extended a clausal sort specification notation introduced in [28] to include the specification of existential sort assertions, the dual of universal sort constraints. We also developed a definitional specification notation, in which many important taxonomic relations can be asserted (e.g. sort *university_student* is defined as the union of sorts *grad_student* and *undergrad_student*). Although the two forms are equivalent in power, the latter may be more intuitive for some constraints.

Using the set of base sorts, and the existential and universal sort relations, we defined a *sort context*, and formalized the *sort reasoning problem* as the problem of inferring whether a given conjunctive sort s is provably empty, provably non-empty or neither, given a particular sort context. Sort reasoning is NP-Complete in general, and for many-sorted logics this is of little concern, since sound and complete resolution strategies can be used. A main contribution of this chapter is the identification of a tractable subcase of sort reasoning, which is important for practical many-sorted systems. We identified a number of restrictions that achieve a polynomial-time sort reasoning algorithm, while retaining a relatively high-level of expressive power. This goal is not easily obtained, due to the many ways in which intractability may creep into a sort structure.

Chapter 8

Reference Constraints in Logic Programming

"Man stays wise as long as he searches for wisdom; as soon as he thinks he has found it, he becomes a fool"

– Talmud

Equality constraints that arise through unification partition logical variables into coreference classes, each of which denotes an individual in a domain of discourse. These classes, however, are unrelated to each other. We develop *reference constraints* as a generalization of equality constraints, allowing the specification of a partial ordering among coreference classes. This leads to the notion of *individual level inheritance*, where an individual denoted by a variable may inherit properties from another individual denoted by a subsuming variable in the partial order. A variety of systems, especially systems that reason in ambiguous domains, can benefit from an efficient, formally based implementation of reference constraints.

8.1 Introduction

Sort (or *class*) level inheritance permits the declaration of properties for a sort, which are automatically propagated to all of its sub-sorts. A sort represents a conjunctive set of individuals (the subset of the universe that belongs to the sort), whereas a variable represents a disjunctive set of individuals (the subset of the universe that contains the individual). Each individual (or instance) inherits the combination of properties of its ancestors in the sort hierarchy. For multiple-inheritance hierarchies (i.e. general partial orders, not just trees), research has focused on resolving conflicts among the inherited properties (e.g. [22, 85, 143]).

There are, however, applications in which inheritance among individuals (*instance level inheritance*) is useful. If an individual α inherits from another individual β, then any additional properties acquired by β must also be dynamically acquired by α. Such constraints may have use, for example, in systems that explore alternatives in ambiguous situations. During a line of exploration, we may determine properties of the solution we seek that must be propagated to all lines of exploration. Systems that exhibit such characteristics include natural language processing systems, automatic configuration systems, dynamic programming, and non-monotonic reasoning systems.

An unsatisfactory way of achieving this is to allow instances to be maximally specific (or leaf) sorts. The problems of mixing class and instance (i.e. subset vs. element) links in hierarchies were clearly identified by Woods [158] and Brachman [16]. Another unsatisfactory solution is to create new sorts that denote single elements, because sorts are declarative in nature whereas individuals are assertional. Reference constraints provide a formal means of instance level inheritance.

Logical variables denote individuals. This is true even for a universally quantified variable; it may *range* over a set of individuals, but can only denote one of these at any instant. Although variables may be sorted, the key difference between the sets represented by variables and sorts is that sorts are conjunctive (e.g. every instance in the set denoted by *dog* is a dog) and variables are disjunctive (e.g. X:*dog* denotes some instance in the set denoted by

dog). We show how the symmetric coreference constraints imposed by equality among variables can be decoupled into two asymmetric, unidirectional *reference* constraints. Although individual level inheritance and reference constraints may be applied to a general many-sorted logic setting, we focus on logic programming. We use Prolog and LIFE [4] for examples, and discuss how reference constraints can be efficiently implemented using attributed variables [86].

After providing some background, we describe our decoupling of coreference in logical variables. This includes a discussion of the syntax and semantics of reference constraints, maintenance of the reference order, an extended example, a comparison with sort hierarchies, and how reference constraints may be efficiently implemented in a logic programming language. Section 8.4 develops and justifies instance level inheritance, including a number of potential applications.

8.2 Background

The entity to which a logical variable refers to may be unspecified or partially specified. In logic programming, each variable X has an associated term $\tau(X)$ that contains information regarding the entity that it denotes. In case there is no information, $\tau(X) = _$. When two variables X and Y are unified (i.e. $X = Y$), then we are saying that the entities to which X and Y refer are the same (i.e. X and Y *corefer*). Any change to X is reflected in Y and vice versa (i.e. $\tau(X) = \tau(Y)$). Naturally, to ensure this property, any rational implementation will store only one term for X and Y. Such a constraint is called an *equality* or *coreference* constraint, and is a fundamental basis for some logic programming languages such as Prolog. Equality constraints partition variables into a set of unrelated coreference classes.

8.3 Decoupling Coreference via Reference Constraints

Suppose we decouple coreference and permit *reference* constraints. That is, suppose we can say that X refers to Y without saying the converse. To do this, we add a reference (or *semi-unification* or *subsumption*) operator $\preceq$. The constraint $X \preceq Y$ states that $\tau(X)$ *must be* subsumed by $\tau(Y)$ (but not necessarily the converse). Any property holding for the entity to which Y refers must also hold for the entity to which X refers (i.e. information in $\tau(Y)$ implies that this same information, and possibly more, must be in $\tau(X)$). The pair of constraints $X \preceq Y$ and $Y \preceq X$ is equivalent to coreference/unification (i.e. $X = Y$). Since the term associated with a variable is just an approximation of an entity, $X \preceq Y$ implies differing degrees of knowledge (i.e. the range of variable X is a subset of the range of Y). In Prolog, an entity denoted by a variable is only fully specified when the associated term is ground. LIFE, however, is based upon approximation — terms have unbound arity (i.e. the arity of terms is not fixed), and so the notion of a ground term has no meaning.

What are the consequences of reference constraints? Reference forms a preorder on the set of variables in a clause. That is, reference is transitive and reflexive. However, it also forms a partial order among *coreference equivalence classes*. If $X \preceq Y$ and $Y \preceq X$, then X and Y are in the same equivalence class. Note that in order theory [38], we can always form a partial order from such classes for any preorder. Logical variables in logic programming languages such as Prolog or LIFE create a set of coreference equivalence classes, but there is no connection among these classes. With our treatment of reference constrains, we can construct a relation among these classes.

If $X \preceq Y$ and we further instantiate $\tau(Y)$, then we must similarly update $\tau(X)$ (and the terms for all variables subsumed by the class of X). For example, the output for the code: $X \preceq Y$, $X = f(_, b)$, $Y = f(a, _)$ will be: $X = f(a, b)$, $Y = f(a, _)$.

More formally, we can define a set of reference constraints as a state in a logic program. We sketch the formal details here. We first define some relevant static aspects of a program:

- Let $\mathcal{U}$ be the domain of discourse (i.e. the set of individuals).
- Let $\mathcal{X}$ be a set of variables. This may be infinite, or viewed as the variables mentioned in the logic program.
- Let GAF be the lattice of logical terms, or generalized atomic formulae [121].

We now define the state (relevant to reference constraints) of a logic program:

- Let $\tau{:}\mathcal{X} \to GAF$ be a function mapping variables to terms. Initially, $\forall X \in \mathcal{X}$, $\tau(X) = _$.
- Let the *reference constraints*, $\preceq$, be a preorder relation on $\mathcal{X}$ such that, for $X, Y \in \mathcal{X}$, $X \preceq Y$ implies $\tau(X) \leq_{GAF} \tau(Y)$ (i.e. the term of X is subsumed by the term of Y in GAF).

From $\preceq$ we can extract two relations:

- The *coreference equivalence relation*, $=$, is defined as: for $X, Y \in \mathcal{X}$, $X = Y$ if and only if $X \preceq Y$ and $Y \preceq X$. We denote the set of equivalence classes as P. For each equivalence class in P, we identify one member element X as a *representative* for the class, and denote the equivalence class as $[X]$. We can extend the function τ to reference classes: $\tau([X]) = \tau(X)$.
- The *reference (partial) order*, $(P, \preceq_P)$: for $[X], [Y] \in P$, $[X] \preceq_P [Y]$ if and only if $\forall X_i \in [X], Y_j \in [Y]$, $X_i \preceq Y_j$. Clearly $\preceq_P$ is reflexive and transitive. To show anti-symmetry, consider two coreference classes $[X]$ and $[Y]$. If $[X] \preceq_P [Y]$ and $[Y] \preceq_P [X]$, and $X_i \in [X], Y_j \in [Y]$, then $X_i \preceq Y_j$ and $Y_j \preceq X_i$. Thus, $X_i = Y_j$, so it must be the case that $[X] = [Y]$.

In this framework, we can identify two state changes that may occur during the processing of a logic program: updates to $\preceq$ and updates to τ. These updates are caused by explicit reference and coreference constraints, and through unification, as we discuss in section 8.3.2. We assume initially that both are monotonic (we can only add new reference constraints, and further instantiate terms). That is, suppose $(\preceq_i, \tau_i)$ and $(\preceq_{i+1}, \tau_{i+1})$ are two subsequent states of $\preceq$ and τ in the program. Then $\preceq_i \subseteq \preceq_{i+1}$ and $\forall X \in \mathcal{X}$, $\tau_{i+1}(X) \leq_{GAF} \tau_i(X)$. This condition holds in Prolog, but may be invalidated in LIFE by destructive variable assignment.

8.3.1 Notational considerations

There are two ways in which coreference can be noted in a logic program: explicitly through an equality constraint (e.g. $X = Y$), or implicitly by using the same variable name at two or more locations in a clause (e.g. $f(X, X)$ in Prolog or $person(mother => X{:}person, bestFriend => X)$ in LIFE). Although the implicit notation is important to keep clauses concise and clear, it can be viewed as a convenience; we could replace all occurrences of a variable X by unique names, and explicitly state the coreference constraints among this set of variables.

Reference constraints can be noted in clauses explicitly (e.g. $X \preceq Y$ could be noted using ASCII as `X <~ Y`). Implicit notation for reference constraints may be confusing, and we do not consider this possibility.

8.3.2 Maintaining and satisfying the reference order

In a logic programming language, such as Prolog, the scope of a variable is the clause. Due to the coreference constraints on variables in the head of a clause when a predicate is called, the initial coreference classes may not all be singletons. For example, if we call the predicate $f/2$ with $f(X, X)$, then the two variables in the head will already be in the same coreference class upon entry to the clause. Similarly, a predicate may alter the coreference classes of calling clauses. For example, if the predicate $g/2$ unifies its two head variables (e.g. if the head clause is $g(X, X)$), then the coreference classes of the two variables in any calling clause will be combined. Thus, from the perspective of a clause, we start with a given set of coreference classes containing the variables in the head, which may be modified (monotonically) in either the head or the body of the clause. With reference constraints, the reference order will similarly be modified.

At any stage in the processing of a clause, we have a current reference order $(P, \preceq_P)$, where P is the set of coreference classes. For efficiency, we only maintain the representative for each coreference class in P, and the association of variables with their representative (e.g. via union-find). In this way, reference constraints are constructed on top of standard coreference. There are three situations we need to consider.

Explicit reference constraints Suppose we encounter an explicit reference constraint $X \preceq Y$, where the representatives for X and Y are X' and Y', respectively. If $[X'] \preceq_P [Y']$, then nothing need be done. Otherwise we must update the reference order and propagate changes to new descendants.

If $[Y'] \preceq_P [X']$, then we *collapse* the suborder between $[X']$ and $[Y']$, *completing* the coreference between X and Y: for any class $[Z]$ for which $[Y'] \preceq_P [Z] \preceq_P [X']$, we merge $[Z]$ with $[Y']$. After all such classes have been merged, we propagate the term associated with $[Y']$, which will be at least as instantiated as the term associated with $[X']$, to all new descendants of $[Y']$. These will be the coreference classes $[Q]$ for which, prior to the hierarchy update, $[Q] \preceq_P [X']$, but $[Q] \not\preceq_P [Y']$.

If $[X']$ and $[Y']$ are incomparable, then we simply add this new constraint to the order. Classes below $[X']$ will now also be below $[Y']$, so new descendants of $[Y']$ (including $[X']$) need the term associated with $[Y']$ propagated to them.

Explicit equality constraints Suppose we encounter a variable unification $X = Y$, where the representatives for X and Y are X' and Y', respectively. We could handle this as two separate reference constraints $X \preceq Y$ and $Y \preceq X$, but it may be more efficient to handle the coreference directly. If $X' = Y'$ then nothing need be done. If either $[X'] \preceq_P [Y']$ or $[Y'] \preceq_P [X']$, then we handle the completion of this coreference as above. If, however, $[X']$ and $[Y']$ are incomparable, then we merge these reference classes, and propagate the term associated with $[X']$ to the descendants of $[Y']$ (that are not also descendants of $[X']$) and vice versa.

Term unification Additional coreference class updates and term propagation may result from implicit constraints arising in unification. During the unification of two terms, if we unify a variable X with another variable Y, then the situation is as above.

Suppose, however, we unify a variable X with a term τ_1 (e.g. $X = f(a, Z)$). In this case we find the representative X' for X, unify τ_1 and $\tau(X')$, and propagate this unified term to all descendants of $[X']$ in the reference hierarchy. Although this operation does not directly modify the hierarchy, the unification of τ_1 and $\tau(X')$ may result in further coreference class mergings, as described above.

8.3.3 Example

We now show an example with which we hope to elucidate the nuances of reference constraints. Consider the following predicates:

```
p(G,H,I) :- G <~ H, G <~ J, I <~ H, K <~ G,
            G = f(g(_),_,_), H = f(_,h,_), K = f(_,_,k),
            q(J,K,H).
q(A,B,C) :- A = f(g(a),_,_), C <~ B.
```

Now consider the results of the predicate call `p(X,Y,Z)`. Initially, there are three separate, incomparable coreference classes, as shown in the first reference order in Figure 8.1, where $\top$ represents an implicit top element. The second reference order in the figure results after processing the body of `p` before the call to predicate `q` (where the associated terms are shown below the variables). The structure arises from the reference constraints. For example, the constraints `G <~ H` and `G <~ J` set input variable X (unified with G) to be subsumed by variables J and Y (unified with H). The associated terms arise from the explicit unifications in the predicate and the flow of information in the reference order. For example, the term associated with X is formed from the unification `G = f(g(_),_,_)` and the inheritance of information from J and Y.

The third reference order results after processing the first predicate in the body of `q`. The order itself did not change, but propagation from J to X and K occurs. The next reference order is the final order after variables X, Y and K merge to form one coreference class, with representative Y. The last order shows the returned state after the local variable J is removed.

8.3.4 Comparison with sort hierarchies

There are a number of similarities, but also many important differences between our reference hierarchy and sort hierarchies in many-sorted logics [28] and sorted logic programming languages (e.g. LIFE [4]). The two are compatible, but independent uses of partial orders.

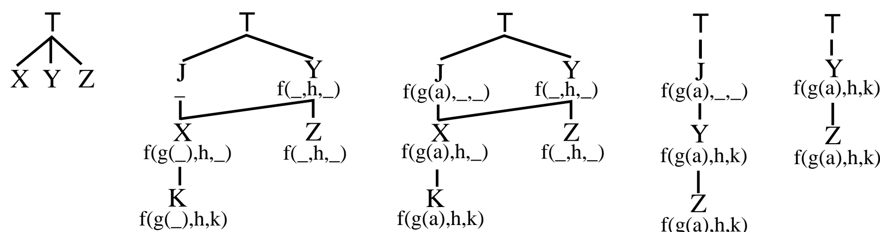

Figure 8.1: State of the reference order at various points in a predicate evaluation

Semantics: As mentioned above, a sort represents a conjunctive set of individuals, whereas a variable represents a disjunctive set of individuals. If the exact individual denoted by a variable is unknown, the set represented by it is neither empty nor a singleton. The distinctions between sorts and individuals (or declarational vs. assertional relations) are described in [16], and the need to distinguish between *subsort* (i.e. *isa subsort of*) relations and *member* (i.e. *isa instance of*) relations is justified. Thus, we cannot intermix the sort hierarchy and individuals (where individuals might be seen as minimal sorts or leaves of the hierarchy). In a sense, reference constraints add another relation "*is more specified than*" among instances.

Scope: There is a fundamental difference between the scopes of sorts and variables. A sort hierarchy is intrinsically global (declarational) in scope. In many systems (e.g. imperative objected-oriented languages such as C++), the sort hierarchy is specified at compile time. In LIFE, the sort hierarchy may be modified during run-time, but in a limited way. New sorts may be added, and sorts may be redefined (e.g. to have new attributes), but these changes are not propagated to existing individuals that are subsorts of those modified.

The scope of a variable in logic is well-defined. In logic programming languages, the scope of a variable is not global to a program, but local to a clause. Thus, all variable changes are during run time, which we would expect to be more frequent than changes to sort hierarchies. In our approach, any change to the reference hierarchy is reflected in the instances represented by the variables affected.

Dynamic Behaviour: A key difference between sort hierarchies and reference constraints is with unification. In sorted logic programming, unification does not modify the hierarchy; rather the unification of two sorts is generally their greatest lower bound. With reference constraints, however, unification may actually change the structure of the reference hierarchy, which in turn may modify terms associated with affected variables. This was exemplified in section 8.3.3.

Thus, we conclude that sort and reference hierarchies share some similarities, but are fundamentally different and independent. However, they are not mutually exclusive, and we feel that systems should provide both features.

8.3.5 Implementation

Can reference constraints be efficiently implemented? If only coreference is used, then the reference order is an anti-chain (i.e. each pair of coreference classes is incomparable). In this case there is little or no overhead when permitting reference constraints. If reference is used, then we must maintain the partial order among coreference classes, and propagate changes in a class to all of its subclasses. This could be achieved efficiently through *attributed variables* [86], where the *cover* (child) relation is stored with variables, and may be implemented at the WAM level. Thus, a modified variable will have knowledge of its immediate descendants in the reference order, and so changes can easily be propagated. Initially, the set of children for a variable will be empty. For changes to the reference order, the only lattice operation that we need to perform is comparability (i.e. $X \preceq_P Y$?). This could be achieved in time linear in size of the descendant cover relation for Y with a (parallelizable) marker passing algorithm. Such an algorithm would be efficient as long as the size of reference order did not become too large, in which case taxonomic encoding techniques could be exploited.

To facilitate backtracking, the state of the reference order would have to be saved, along with the standard trail information, at choice points. Reference constraints also merge well with *memoing* techniques [152]. Instead of tabling only predicate call and return value information, we also need to store the relevant aspects of the reference order prior to the predicate call, and upon return from the call. The relevant portion of the reference order P for a predicate invocation is simply the suborder of P that contains only the variables mentioned in the predicate call. When a look-up matches an entry in the table (i.e. both the predicate call and reference constraints on variables in the call match), then we simply use the result information, which will provide both variable values and updates to the current reference order.

8.4 Individual Level Inheritance

What are the benefits and uses of reference constraints? Ironically, although large reference orders may benefit from taxonomic encoding, it was in the development of our constraint-based view of encoding that the need for reference constraints was first identified [47]. Encoding is, however, a limited domain of utility for this general mechanism. More interesting applications arise with the notion of *individual level inheritance* (inheritance among individuals as opposed to classes). A sort hierarchy provides a partial order among sets of entities, whereas reference constraints construct a partial order among individual entities. Thus sort hierarchies and object-oriented class hierarchies permit class to class and class to individual inheritance.

There are several reasons why we may want individual level inheritance. In an ambiguous domain, we may want to separate the known information about an entity from hypothetical or speculative information. In complex scenarios, we may want to separate information related to an entity in different contexts. We may even want to relate different entities that must share some common, but dynamically changing properties. In all these cases, reference constraints permit the separation of information, while retaining a close structural relation. We now describe some properties of applications that may benefit from individual level inheritance.

In an ambiguous setting, we may have some information regarding an entity that we are certain of, and we may have other information that we are uncertain of. In an exploratory fashion, we can analyze this other information, perhaps in a breadth-first manner. If we discover new information with certainty, we can apply it to the original entity, and it will be propagated down all paths of exploration. Any paths that become inconsistent will be pruned, requiring a different processing strategy than Prolog: instead of backtracking when the term of a variable X becomes inconsistent, we can simply mark X as inconsistent (e.g. $\tau(X) = \bot$) and prune it from the reference order.

Another case arises if we want to retain information for a single entity, but in separate *contexts*. For example, suppose we have a variable $John$ which represents general aspects of a person named John. We may have additional variables $Father_John \preceq John$ and $Pilot_John \preceq John$ which represent fuller information related to John in the context of his being a father or a pilot. This situation is shown in Figure 8.2. We could combine these two contexts with a variable $Father_Pilot_John \preceq Father_John$, $Father_Pilot_John \preceq Pilot_John$. In this way, we maintain the information related to John in a hierarchically structured way; all information is accessible, but the information within any context will not be cluttered by irrelevant information. In addition, any updates at higher levels (e.g. adding general information about John, such as his age) will be propagated to all lower levels. Such a scheme may also be used for analyzing aliases, particularly if we allow information introduced at a descendant to override that introduced at an ancestor (i.e. local information having precedence over inherited information).

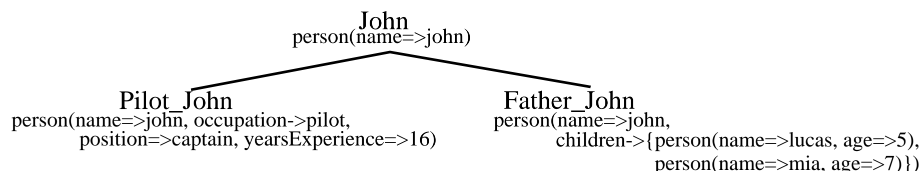

Figure 8.2: Reference order for separating the contexts for a person named John

The above outlines properties of applications that would benefit from individual level inheritance. We next describe some concrete applications.

Automatic configuration: Suppose we have a system that automatically designs a system configuration given a set of constraints among components and a set of specification constraints (e.g. [37]). At any point, we may be certain about some properties of our system $BaseSys$, but uncertain about others. Through exploratory reasoning, we could try a number of possibilities simultaneously, each of which must conform to $BaseSys$.

For each possibility, we could assign a variable, say Sys_i, and make the constraint $Sys_i \preceq BaseSys$. We could then add additional, hypothesized components to Sys_i. Of course, this could be done recursively, creating an entire hierarchy of possibilities, with $BaseSys$ as the root. If we also detect relations among hypothetical systems, then this hierarchy may be a general partial order, not just a tree (e.g. if we detect that Sys_j, where $Sys_j \preceq Sys_i$, is an enhanced system of Sys_k, we can add $Sys_j \preceq Sys_k$).

During processing, we may determine the necessity of components in a higher system, resulting from analysis or additional user input. For example, if we realize the need for a certain component in the base system, we add it to $BaseSys$ (via unification) and it will be automatically propagated to all of its descendants. This propagation may detect inconsistency of one or more hypothetical systems, which will then be pruned from the search space.

Of course, this system may be incorporated as part of a larger constraint solving system, and reference constraints can be viewed as one more form of constraint in constraint logic programming.

Natural Language Processing: Computational linguistics systems must be robust, due to the high level of ambiguity in human languages. As examples, consider phrase parsing and discourse processing. A number of techniques, such as chart parsing [70, 6, 119, 134], have been designed to minimize the effort involved in analyzing a sentence that may have multiple parses.

For a simple example, suppose a variable X represents what is known about a phrase, and variables Y_i (where $Y_i \preceq X$) represent the investigation of various ambiguous parses (i.e. for each Y_i some decision has been made regarding the interpretation of an opaque word or phrase). During the parse, if something becomes known about the entire sentence X (or about some sub-parse higher than the current level), this must be propagated down from X to the Y_i (and recursively to their descendants). This idea can be extended from single sentences to entire discourses.

In the sentence "*Jack saw a dog on his way home*", the prepositional phrase "*on his way home*" may apply to either the dog or to Jack. We may have semantic preference rules that would select the latter reading, but the context of this sentence may override such rules. Thus, we may explore both possibilities, but focus on the most likely reading given the current information available. In either case, we know that Jack saw a dog, so we may assert this as known, and place the two readings in relation to this using reference constraints. Later processing may incorporate additional certain information, which may prune one of the possibilities.

To achieve this using reference constraints, we must use a representation for parsed sentences in which ambiguity can be resolved via further instantiation of terms. Figure 8.3 shows one possibility in which prepositional phrases are stored in a list as the last argument of the main predicate[1]. In the term for variable X, we denote the ambiguity as to whether Jack or the dog is on his way home using the disjunctive set notation $\{Y; Z\}$ (where, for example, $\{jack; dog\}$ unified with dog results in dog). Although Prolog does not support such notation directly, it can be specified in LIFE and with sparse logical terms [51].

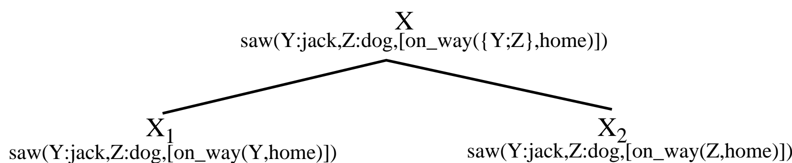

Figure 8.3: Reference order for ambiguous parses of "*Jack saw a dog on his way home*"

[1] More linguistically motivated possibilities also exist, but their development is beyond the scope of this thesis.

As another example, the word "*chair*" is ambiguous in the sentence "*When Sherry saw the chair, she shook her hand*". The default reading may be as a piece of furniture, but it may also refer to the chairperson of a meeting. By maintaining both possibilities, backtracking may be avoided as further information is discovered. Figure 8.4 shows how this may be represented using an interaction between reference constraints and a sort hierarchy. The first diagram in the figure shows a portion of a sort hierarchy for word meanings, in which *furniture_chair* and *meeting_chair* are both subsorts of *chair*, and *meeting_chair* is a subsort of *person*. The second diagram shows the reference order after the sentence has been parsed. The pronouns *she* and *her* have not yet been resolved, and the disjunctive set notation indicates that both must refer to either "Sherry" or "the chair" (although the default may be that "she" refers to "Sherry" and "her" refers to "the chair"). In the interpretation where "the chair" is a piece of furniture, we apply the semantic constraint that hand shaking is done by persons, leading to a parse in which Sherry is shaking her own hand.

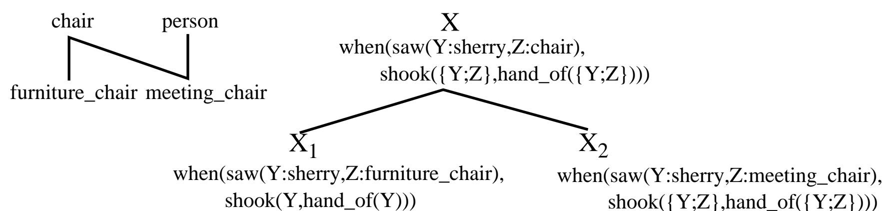

Figure 8.4: Reference order during parse of the sentence "*When Sherry saw the chair, she shook her hand*"

Chart parsing can be viewed as an instance of dynamic programming. It is generally bottom-up in that it starts with words, which coalesce into larger and larger phrases, until one phrase (often a sentence) spans the entire input. The benefit of saving intermediate results is a reduction in redundant processing (which is also the basis of, and motivation for, memoing [152]). Reference constraints can be used as an automatic aid to dynamic programming systems in which information that applies to a node in the search space can be automatically propagated, with inconsistencies corresponding to pruning.

Reference constraints may also aid in the integration of top-down and bottom-up techniques of discourse processing by providing a structure for relating intermediate results. By maintaining ambiguity using reference during top-down parsing, needless backtracking may be avoided. If bottom-up results are stored in a form that is unifiable with the final result, then they too can be coalesced using reference. Thus, both forms of processing create additional entities below existing entities; certainty is added higher up in the reference order, and uncertainty is added at lower levels. When the entire structure coalesces into one coreference class, all ambiguity has been resolved.

Non-monotonic and Default Reasoning: Although default properties are specified in sort hierarchies, reference constraints may be exploited to enhance the efficiency of default reasoning by allowing a clean way of separating known from assumed properties. When a variable X is constrained to be of sort s (e.g. via an assertion of the form $X{:}s$), we can unify X with all the strict properties of s, and create an implicit *default variable* X_d, where $X_d \preceq X$, with which we unify all the default properties of s. In order to maintain the default variable, new properties of X are unified with X_d using what we call *c-unification* [141]. In c-unification, one of the terms is *dominant* and the other is *subordinate*. If a conflict arises during unification, instead of failing, only the information in the dominant term is kept. Thus, when updating X_d after a change to X, we c-unify $\tau(X)$ with $\tau(X_d)$, where $\tau(X)$ dominates $\tau(X_d)$. In this way X_d retains only those default properties that may still be applicable to X. Additional default reasoning strategies (as in e.g. [22, 85, 143]) may be built into c-unification. The importance of using reference constraints in this way is that monotonic aspects of reasoning can be separated from, but still related to, non-monotonic aspects.

To illustrate, we use the standard flying birds example. Suppose that *bird* is a sort with default properties *feathered=>true* and *fly=>true*, and that *penguin* is a subsort of *bird* with a strict property *fly=>false* and a default property *home=>antarctica*. The first diagram in Figure 8.5 shows the situation after initializing

a variable *Opus* to be of sort *bird* (e.g. after an assertion of the form *Opus*:*bird*). The second diagram shows the situation after we specialize *Opus* to be of sort *penguin*.

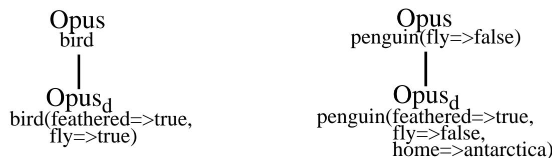

Figure 8.5: Reference constraints for default reasoning

> The hypothetical reasoning systems we described add uncertain assertions as children of a node. In this way, certainty can be incorporated as it is determined, and removal of assertions corresponds to pruning children. However, there may be systems in which assertions must be explicitly withdrawn without pruning the node. In this case, additional work must be performed since branches of the reference order may have been pruned using the information to be withdrawn. One possible solution is to mark, but not prune, inconsistent nodes of the reference order. These nodes would be treated as pruned unless an assertion they contain is removed, in which case they may change state from inconsistent to consistent.

Individual level inheritance is certainly possible without reference constraints, and in fact many systems appear to be already doing this. However, we can apply the same arguments as for sort hierarchies in many-sorted logics, and for inheritance in object-oriented systems. By making this process explicit, declarative and automatic, the programmer (or logician) is freed of the burden of performing this task, and can instead focus on higher-level aspects of the problem. Due to the formal basis of reference constraints as a generalization of equality constraints, we ensure a consistent semantics when individual level inheritance is exploited.

8.5 Conclusion

We have proposed two notions in this chapter: *reference constraints* and *individual level inheritance*. Reference constraints are a generalization of equality constraints among logical variables. Equality constraints form equivalence classes based on coreference. Reference constraints decouple the symmetry of coreference, and permit the construction of a partial order of coreference classes. We have shown that, due to the semantic differences between sorts and variables, the reference order is quite distinct from a sort hierarchy in many-sorted logics and sorted logic programming languages. We believe, however, that both are compatible and desirable in a system, although we did not deeply explore the interaction between the two. In this inquiry, we focused on reference constraints in logic programming languages such as Prolog or LIFE [4]. A full model theoretic analysis in a logic system is required.

Reference constraints lead to individual level inheritance, which permits inheritance from one individual to another. This is distinct from the ordinary notion of inheritance which is from a sort (or class) to another sort or to an individual. Through a general outline of the types of applications that may benefit from automating individual level inheritance, and descriptions of its use in automatic configuration (and constraint logic programming), natural language processing (and dynamic programming) and default reasoning, we investigated the potential benefits of our work in logic programming and artificial intelligence systems.

Chapter 9

Organizing the Hierarchy of Conceptual Graphs

"When nothing is done, nothing is left undone"

– Lao Tsu

"Who really invented nothing"

– Walt Kelly

Conceptual structures is a graphical knowledge representation formalism that is equivalent in expressive power to first order logic. There are two main forms of hierarchies used in the formalism: *defined* and *derived*. Defined (declarative) hierarchies, such as sort and class hierarchies, have an explicit partial order relation. In conceptual structures, the *type* and *relation* lattices are defined. A derived hierarchy is a partial order that is induced by internal structural relations among components. Two conceptual graphs can be compared using the subsumption relation, where graph g_1 subsumes graph g_2 if it contains a subset of the information in g_2. Derived partial orders are employed in other knowledge representation systems, most notably for *classification* in the KL-ONE family of terminological systems [18].

To organize derived hierarchies such as these, which are highly dynamic and expensive to construct, a number of techniques have been proposed, including encoding [42] and multi-level indexing [94]. In this chapter, we develop a novel approach to organizing derived hierarchies using graph normalization and spanning trees.

After providing a brief overview of conceptual structures, we introduce some normalization techniques for conceptual graphs, leading to *spanning tree normal form* (STNF). In [50], we show how an integration of sparse terms and *order-sorted feature terms*, called *sparse feature terms*, can be used to implement graphs in STNF, and how some operations on graphs in STNF can exploit unification and enhance operational efficiency. Starting with graphs in STNF, we develop a generalization hierarchy normal form (GHNF) with which we organize the derived hierarchy of graphs, called the *generalization hierarchy*, into a spanning tree. We show how searches in this hierarchy can be performed efficiently using this spanning tree organization.

9.1 Background and Motivation

Since details of conceptual structures are not necessary for the following, for brevity we choose to limit detailed background on the subject, which can be found in [136]. Essentially, a *conceptual graph* (CG) is a connected bipartite graph consisting of labeled *relation* nodes and *conceptual type* nodes. Conceptual types are standard ontological objects, such as "person", "cat" or "eat", and conceptual relations are basic relations among types, such as "agent" and "object". A standard example graph is shown in Figure 9.1 [136], and represents the declarative statement "a cat sitting on a mat".

For our research, there are three ordered sets that are important: the conceptual types (the *type lattice*), the conceptual relations (the *relation lattice*), and the graphs themselves (the *generalization hierarchy*). The formalism

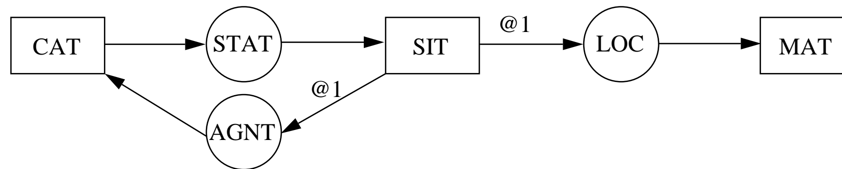

Figure 9.1: Conceptual graph representing "a cat sitting on a mat"

requires both the types and relations to form lattices, which we have argued is overly strict, and that only ordered sets are required [50]. These two ordered sets are definitional, in that the user imposes the partial order relation. Since previous chapters have dealt with encoding defined taxonomies, we omit further discussion of the type and relation lattices.

The generalization hierarchy, on the other hand, is derived using a set of *canonical formation rules* that define how graphs relate. If graph g_1 can be derived from graph g_2 using the canonical formation rules, then g_1 must contain at least as much information as g_2. A conceptual graph system begins with a set of given graphs, called the *canonical basis*. All other valid graphs used by the system must be derivable from the canonical basis.

Spanning trees are a valuable tool for improving the operational efficiency of graphs and the generalization hierarchy. We only deal with *atomic* conceptual graphs in which all relations are both dyadic and invertible. Atomic conceptual graphs contain no logical connectives (i.e. they are connected), no logical quantifiers (other than the implicit existential), and no nesting (i.e. there is only one context) [26, 41]. The *inverse* of a dyadic conceptual relation R is a relation R^{-1} that is semantically identical to R with the direction of the arrows reversed. For example, the inverses of AGNT and PARENT are AGNT_OF and CHILD, respectively. Similar assumptions have been made in [41, 107, 111, 160].

We first discuss the notions of cardinality constraints and functional relations. Although cardinality can be expressed using sets or complex nesting of contexts, it is important to have the ability to express such constraints simply and declaratively. Graph normalization techniques introduced in [107] are expanded upon in section 9.3 to prepare for constructing the *spanning tree normal form* that we introduce in section 9.4. Of particular importance to operational efficiency is the elucidation of functional relations in graphs. We then explore their use in the generalization hierarchy to specify a *generalization hierarchy normal form*, to enhance search operations such as matching and retrieval, and to efficiently perform topological traversals.

9.2 Cardinality Constraints

Although some conceptual relations are functional in character, CG theory provides no simple way to represent these and other forms of cardinality constraints declaratively, without resorting to the use of actors, sets or complex nesting of contexts. Actors imply computation of dependent concepts from independent concepts, while sets do not restrict the number of relations of a particular type, which can be a valuable constraint for normalization and matching. For example, the canonical graph: [EAT]→(AGNT)→[ANIMATE] does not tell us whether an act of eating must have exactly one agent or may have multiple agents (i.e. if AGNT is a *functional* relation of EAT). Another example is: [PERSON]→(SPOUSE)→[PERSON], which says that the spouse of a person must be person, but does not constrain a person to have at most one spouse. For illustration, we assume that both of these cases are functional.

Definition 9.1 *A* cardinality constraint, *@n* $(n \in \mathcal{Z}^+)$, *between a concept c and a relation r states that at most n relations of type r may be connected to c.*

A cardinality constraint is denoted on the arc between the concept and the relation. Thus, the above example becomes: [EAT]-@1→(AGNT)→[ANIMATE]. Restricting a relation to one occurrence for a concept (i.e. $n = 1$) is a *functional cardinality constraint*, and it is these constraints that we focus on. The connection to logic is simple: if the variable representing the independent concept appears in two instances of the relation, then the variables representing the dependent concepts must be equal. This provides a sort of uniqueness constraint. Our example translates to: $\exists x \exists y$ (EAT(x) $\wedge$ ANIMATE(y) $\wedge$ AGNT(x, y) $\wedge \forall z$, AGNT(x, z) $\supset z = y$). We do not suggest that all

functional dependencies can or should be expressed in this way. Rather, we feel that by notating functional relations, normal forms for CGs will be more distinct and easier to determine.

Cardinality constraints blend well with set cardinality [63, 136]. For *set coercion*, a cardinality constraint can be moved into the set notation. On expansion, the set cardinality can be moved out to a cardinality constraint. To ensure set joins, we make concept sets functional. As an example, for: [DANCE]→(AGNT)→[PERSON: Liz], set coercion on PERSON results in: [DANCE]-@1→(AGNT)→[PERSON: {Liz}], whereas set expansion on: [DANCE]-@1→(AGNT)→[PERSON: {Liz,Kirby}@2] results in: [PERSON:Liz]←(AGNT)←@2-[DANCE]-@2→(AGNT)→[PERSON:Kirby].

9.3 Normalization

Normalization is important to enhance the similarity among graphs and can be achieved via transformation rules [107]. We assume that all relations are invertible so, e.g., the inverse of WORKS_FOR is EMPLOYS, whereas the inverse of SPOUSE is itself (i.e. it is symmetric). In [50], we show how our representation automatically performs some simplification, reducing redundancy that can arise during joins.

Explicitly representing functional relations can be exploited to determine a precedence between a relation R and its inverse R^{-1}. Priority is given to functional relations. Thus, assuming a world in which a person has at most one nationality, we would prefer the graph: [PERSON]-@1→(CITIZENSHIP)→[COUNTRY] to: [COUNTRY]→(CITIZEN)-@1→[PERSON]. If both R and R^{-1} are functional, we incorporate both (i.e. we perform *symmetry completion* [107]). By doing this, we can traverse all functional relations in the direction of their arcs. If neither R nor R^{-1} are functional, other preference schemes need to be specified.

Normalization will also incorporate selectional constraints related to the graph, particularly those which add functional relations between concepts. To illustrate, the well-known example in Figure 9.1 shows a normalized version of the CG, in which the concept SIT imposes the selectional constraint that it has exactly one agent.

9.4 Spanning Tree Normal Form

It is easy to specify a spanning tree for any conceptual graph, with coreference linking identical concepts as in the linear form. Any traversal of a graph that visits every concept and relation defines a spanning tree: the first node visited is the root and cycles are broken by introducing coreference. Our goal is to specify a *spanning tree normal form* (STNF) that can be used to improve the efficiency of CG operations, by exposing functional relations, as well as to organize and search the generalization hierarchy. In [160] there is also a proposal for a normal form that is a spanning tree, but the tree is determined in an ad hoc manner (alphabetical order is used to select the root and relations to expand partial trees).

Definition 9.2 *A* spanning tree T *for a conceptual graph* G *is a connected acyclic subgraph of* G *containing all the concepts of* G *(but not necessarily all the relations). For each spanning tree, one concept is designated the* root.

In the linear form [136], concepts and relations form the nodes of a spanning tree, and arcs are labeled with directional arrows. For STNF, only concepts are nodes while relations are arc labels. The direction of arcs is implicitly downward. Although this format is suitable for binary relations, which form the majority of conceptual relations [123], it may be possible to accommodate monadic and higher-order relations; we do not explore this here. We assume that our graph is normalized as described in section 9.3 and that we have linear extensions τ and ρ of the type and relation hierarchies, respectively. Since some graphs may require multiple root elements, we actually construct a spanning forest. We maintain the individual trees in a list ordered by the type of the root concepts (according to τ). When drawing forests, we add an untyped dummy root to connect the trees together.

We give below an algorithm that takes as input a normalized conceptual graph G, and outputs a spanning forest F that represents G in STNF. The concepts and relations of G are the ordered lists C and R, respectively. Each node in the forest is a concept c to which a (possibly empty) list of children is associated (via *children*(c)). Each

child contains a pair: the child concept and the connecting relation. The root of the tree containing a concept c is obtained by calling $tree(c, F)$.

Algorithm 3 *STNF(input:* $G = < C, R >$*; output:* F*)*

1. $F := C$
2. *for each concept* $c \in C, children(c) := \emptyset$
3. *for each relation* $r(c_i, c_j) \in R$ (taken in order)
4. *if* $(tree(c_j, F) = c_j$ *AND* $tree(c_i, F) \neq tree(c_j, F))$ *then*
5. $children(c_i) := children(c_i) \cup \{< r, c_j >\}$
6. $F := F - \{c_j\}$
7. *else*
8. $children(c_i) := children(c_i) \cup \{< r, coref(c_j) >\}$
9. *end*

First, we start with a forest consisting of each concept in the graph G as a tree (lines 1 and 2). We consider relations one at a time and update the forest as necessary. A node is always placed below the entering concept c_i, labeled with the relation type. If the exiting concept, c_j, is the root of a different tree in the forest from c_i simply connect this tree below c_i (lines 5 and 6). We do this by adding the relation/concept pair to the children list of c_i and removing the tree rooted at c_j from the forest. If, however, c_j is not a root or is in the same tree as c_i, the node below c_i will contain a coreference label linking to c_j (line 8). Once we have visited all relations, we have a spanning forest for our graph. The time complexity of this algorithm is near linear in the number of concepts and relations in the input graph if the *tree* function is implemented using a union-find algorithm.

The order in which we visit relations (line 3) is important. We consider all functional relations, before any non-functional ones. Within these groups, the order depends on the types of the relation and two incident concepts. The order of precedence is the relation, followed by the entering concept and lastly the exiting concept. Exploring the consequences of choosing different precedence orderings is a topic for further research. It may still be possible for there to be two or more arcs with precisely the same relation and incident concept types. In this case, contextual information may be needed for selection. In this preliminary analysis, we simply select one arbitrarily, and this is the only place where non-uniqueness can enter into the process. Thus, our construction computes a spanning tree normal form that is nearly unique for normalized graphs. As an example of this construction, Fig. 9.2 shows the STNF of the graph in Fig. 9.1. Note that both AGNT and LOC are functional relations of SIT. The last relation visited is STAT, which is added using coreference. In diagrams, we notate functional relations using thick lines and non-functional ones with thin lines.

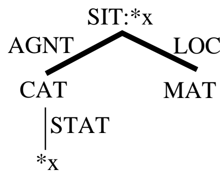

Figure 9.2: Spanning tree normal form

Another well-known example, with a cycle, is: *a monkey eating a walnut using the walnut's shell as a spoon* [136]. Figure 9.3 shows the normalized graph as well as its STNF. For illustrative purposes, we assume that an entity can only be (intransitively) a part of at most one other entity, and that an instance of eating has one agent and one object. Thus the relation PART is inverted to PART_OF. We assume that the linear ordering of relations is AGNT $<$ OBJ $<$ PART_OF $<$ INST $<$ MATR. We first add MONKEY and WALNUT as children of EAT, then a coreference link to WALNUT as a child of SHELL, and finally we add the non-functional relations INST and MATR in the tree rooted at EAT.

For a more complicated example, consider the statement: *a woman eating a dinner cooked by her husband*, which is shown in Figure 9.4. In this case, we end up with two trees since both EAT and COOK only have exiting relations

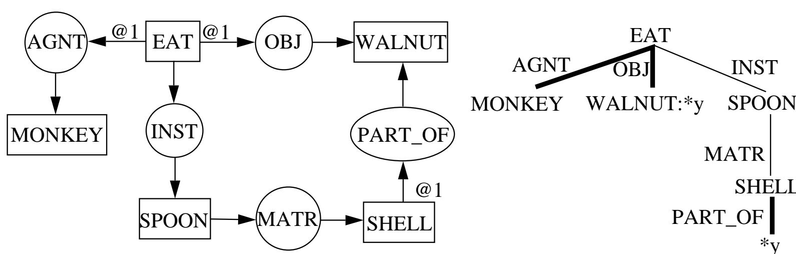

Figure 9.3: A cyclic graph and a tree representation

in the normalized form. Assuming the types are ordered by COOK < EAT < WOMAN < MAN, we obtain the STNF as shown.

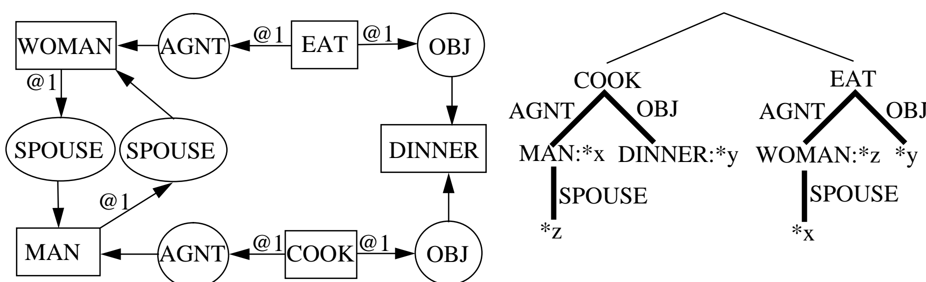

Figure 9.4: A woman eating a dinner cooked by her husband

In [50], we describe more fully the advantages of STNF. Graphs in STNF admit a direct implementation using order sorted feature structures [4, 5]; we developed a variant of sparse terms for this purpose. We demonstrate how the canonical formation rules can be performed on graphs in STNF, in particular how unification can be exploited to efficiently implement these rules by observing the constraints imposed by functional relations. Since these issues are outside the scope of this thesis, we choose to omit details.

9.4.1 Pivoting

Given a graph in STNF, we may need a certain concept to be the root of one of the trees in the forest in order to perform graph matching, to obtain different viewpoints of a graph, or to further normalize the spanning tree for storage in the knowledge base. We call this process *pivoting*. Although there are several possibilities for pivoting, we have chosen one that is particularly simple, yet useful for organizing the knowledge base. We call the node of a concept in a spanning forest that maintains the type information (and possibly has a subtree) the *dominant* node. All other, coreferring nodes are called *subordinate*. Basically, to pivot a concept that is not already a root is accomplished by replacing the dominant node for the concept by a subordinate node and adding the subtree rooted at this node as a top level tree in the forest. Pivoting can easily be carried out, as shown in the following figure which shows pivoting of the STNF form of the graph in Fig. 9.3 on the concepts "WALNUT" and "SHELL".

9.5 Representing the Generalization Hierarchy

A CG database contains of some of the (infinitely many) canonical graphs that can be obtained from the canonical basis B using the canonical formation rules. The generalization hierarchy organizes graphs into a partially ordered set of equivalence classes [41, 111], where each graph in a class is canonically derivable from all others in the class, and

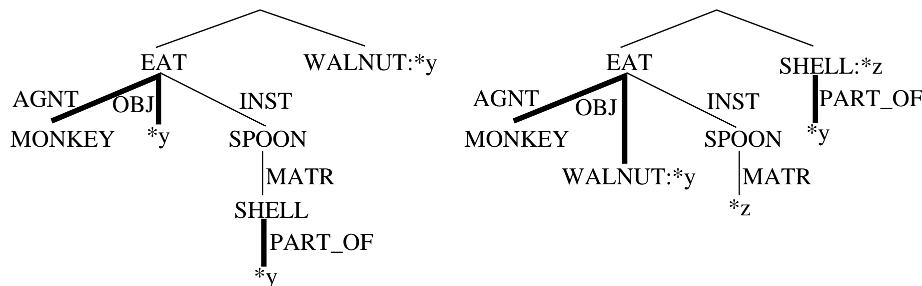

Figure 9.5: Examples of pivoting the graph in Figure 3

one class subsumes another if each graph in the latter is derivable from each graph in the former. The generalization hierarchy consists of both the canonical basis (which represents things that could exist) and the database graphs (which represent things that do exist). Although B may not form an anti-chain, there is a subset B_0 of B that forms the initial level, or co-atoms, of the generalization hierarchy. Our goal is to use STNF to assist in the organization and search of this hierarchy. The advantages of explicitly maintaining the generalization hierarchy are described more fully in [42]. This hierarchy can be encoded so that many operations among graphs in the hierarchy can be performed efficiently using only taxonomic operations, avoiding matching altogether. In our case, we maintain the full hierarchy, but mark one parent of each graph as dominant, to identify a spanning tree.

We first describe the process of constructing the spanning tree for the generalization hierarchy incrementally, leading to another normal form. We start with an empty generalization hierarchy consisting only of $[\top]$ and $[\bot]$. We need to order the children of any element, so we define a total order on graphs (perhaps based on the linear extensions of the type and relation hierarchies, and the form of the graphs). The method used to specify this ordering is not important to the following discussion.

Suppose we have a generalization hierarchy organized with an underlying spanning tree T_G and we wish to add a graph Q in STNF. We essentially use the algorithm of [42] to search the hierarchy and find the immediate predecessors (IP) and immediate successors (IS) of Q. We store graphs so that every STNF graph G is a simple specialization of its parent G' in T_G. That is, G and G' have a direct matching (i.e. their feature term implementations are unifiable, and the term of G' subsumes that of G). This cannot be achieved for all ancestors of G, but if it holds for all ancestors in T_G (i.e. graphs on the path from G to the root $[\top]$), then we can improve search and matching operations. The position of Q in T_G is below the leftmost IP.

As we find each predecessor C of Q in T_G, we modify the form of Q. Since both C and Q are in STNF, the spanning trees in the forest C will be contained in the trees of Q (modulo symmetric relations and coreference). For each tree of C whose root is not a root of Q, we pivot. Pivoting does not destroy the STNF properties, but creates additional trees, so we essentially flatten Q until C is more evident in its forest. When all the ancestors of Q in T_G have been processed, Q will be in *generalization hierarchy normal form* (GHNF). The advantage of a storing graphs in GHNF is that if we have graphs Q and Q' for which Q subsumes Q' in T_G, then Q and Q' have a direct and simple matching. That is, not only is Q' a specialization of Q, the feature terms representing Q and Q' are related by term subsumption.

9.5.1 Depth-first topological traversals

The spanning tree T_G underlying the generalization hierarchy can be viewed as representing a left-to-right (LR) depth first (DF) traversal of the generalization hierarchy. We show here a relation between LR-DF traversals and DF topological traversals, where a topological traversal is any traversal that obeys the topological property that a node cannot be visited until all of its parents have been visited. In [42], the advantages of searching the hierarchy for IP and IS topologically are described.

We make the distinction between breadth first (BF) and depth first topological traversals. In BF traversals, we visit nodes by level. The level in an ordinary BF traversal is the length of the shortest path to the root, since we place an element in the search queue when it is first accessible. The level for a topological BF traversal, however, is the

length of the longest path to the root because we place an element in the search queue only when last accessible (when the last parent has been visited). DF traversals, on the other hand, select the next candidate node to visit with the longest leftmost path to the root (in a LR traversal), where conflicts are resolved by choosing the leftmost element. For ordinary DF traversal, a candidate is any unvisited node that is connected by an arc to the tree traversed so far. When observing the topological property, the only candidates are those whose parents have all been visited.

It should be clear that BF and DF topological traversals are implemented differently (using a queue in the former and a stack in the latter) and may visit nodes in different orders. The proposal in [42] performs a BF topological search of the generalization hierarchy to perform updates and retrievals. We feel that it is interesting to explore DF topological searches for several reasons. First, such a search would result in finding the first member of IP earlier than a BF topological search. Second, we show how the spanning tree T_G can be used to perform a DF topological traversal without needing to mark elements as visited. Third, we can utilize GHNF more fully to improve the efficiency of graph comparisons.

Although we cannot use the LR-DF traversal suggested by T_G in the search algorithm, there is an interesting connection between DF traversals and DF topological traversals. If T_G represents a LR-DF traversal of a hierarchy P, then a right-to-left (RL) DF traversal of T_G is a RL-DF topological traversal of P.

Theorem 9.1 *Suppose G is a rooted directed acyclic graph and T_G is the tree resulting from a LR-DF traversal of G. Then a RL-DF traversal of T_G is a RL-DF topological traversal of G.*

Proof: Consider any point in a traversal of T_G. Suppose the next node to visit, v, with parent p in T_G, has an unvisited parent p_1. Since p_1 is unvisited, it must be to the left of p in T_G, but then during the initial DF traversal p_1 would have been visited before p, and so v would be below p_1 not p in T_G. □

Thus, a simple RL-DF traversal of T_G performs a DF topological traversal of the ordered set without the overhead of checking when all parents have been visited.

In order to fully utilize the spanning tree structure of the generalization hierarchy and the GHNF form of graphs, we describe a modification of the search algorithm of [42]. The problem is to find the immediate predecessors (IP) and then the immediate successors (IS) of a graph Q, which may or may not be in the hierarchy. We assume that after a comparison between Q and a graph u in which $u > Q$, it is desirable to compare the children of u with Q so that we can benefit from the result of the match (while still obeying the topological property). By following the depth first topological traversal described above, this can be achieved with very little effort: we don't even need to mark elements as visited. By marking only those which successfully match Q, we can perform the search with a minimum amount of administration. Furthermore, since graphs are in GHNF, we will successively compare graphs whose GHNF forms most closely match until a subtree is traversed or until a graph is found which doesn't match Q. Another advantage of this approach is that by performing a DF topological search, the focus (as described in [42]) becomes restricted more quickly, providing a more constrained target for guiding the search.

9.6 Conclusion

We have explored the use of spanning tree representations of graphs and the generalization hierarchy in conceptual structures. We first proposed a means of declaratively representing cardinality constraints. Of particular interest are functional relations, which restrict the number of occurrences of a particular relation type to one. These constraints are important for improving the efficiency of matching and other graph operations. We extended and refined CG normalization, as introduced in [107], through the use of functional relations. We developed a spanning tree representation of CGs, leading to a spanning tree normal form (STNF) that is based on semantic content and is less ad hoc than some previous proposals. Graphs represented in STNF have a natural implementation using a variation of order-sorted feature structures, providing a scheme in which graph operations can benefit from the efficiency of feature term unification. Finally, we showed how identifying an underlying spanning tree for the generalization hierarchy can benefit both storage and traversals. A spanning tree can assist in a further refinement of STNF to generalization hierarchy normal form (GHNF) in which all graphs on the same path to the root are unifiable. Furthermore, by

traversing this left-to-right depth first tree in a right-to-left depth first manner, we achieve a depth first topological traversal that can be used as an alternative search procedure of [42]. An advantage of this search, in addition to its efficiency and simplicity, is that graphs which are closely related have a higher chance of being compared successively, so we can take advantage of the results of previous matches.

Chapter 10

A Hierarchical Organization of Landscape Models

"No man can reveal to you aught but that which already lies half asleep in the dawning of your knowledge"

– Kahlil Gibran

Due to the spatial scale at which most empirical landscape studies are performed, replication is rarely feasible, and experimenters may require artificial replication through the use of landscape models that are synthetically generated. In our view, a landscape is a heterogeneous region on the surface of the earth, and a landscape model is a simplified representation (e.g. as a digital map) of a landscape of interest. A generator of landscape models is a procedure for producing landscape models.

Artificial generation of landscape models is becoming increasingly prevalent in landscape ecology and is useful for a variety of purposes, including comparison with real data, testing general theoretical hypotheses, and providing input to simulation models. However, the number of generators of landscape models is increasing and there is no framework within which generators can be analyzed, compared and organized. In this chapter, we propose a hierarchical framework that unifies landscape models within a formal organizational system. A landscape model that is artificially generated using a simple random process is called a *neutral model.* Generators of neutral models produce *instances* of landscape models with two or more patch types, and constrain the patterns generated by specifying the proportion of the model covered by each patch type. We develop a generalization of neutral models, where landscape models are generated according to a *set of constraints* on possible patterns. A set of constraints is a *landscape model prototype.*

Different landscape model prototypes can be compared according to the number and type of restrictions, where a prototype is considered "less neutral" or "more restricted" than another if the former has a superset of the constraints of the latter. This relation produces a hierarchy that captures *gradients of neutrality* among prototypes. The hierarchy thus formalizes, in a mathematically elegant manner, a multi-dimensional transition from neutral models that impose few restrictions on pattern generation to predictive models that impose a variety of more ecologically motivated constraints on the generation of landscape models. In a more practical context, this hierarchy may be used to guide the development of landscape model generators, to aid selection of appropriate existing generators, and to assist in the analysis of models derived from real landscapes through the use of landscape model prototypes.

10.1 Introduction

A landscape is a heterogeneous region of the earth that is composed of a mosaic of different patches, and generally contains a few interacting ecosystems [10]. Landscapes may be defined from the viewpoint of a particular organism, although a common viewpoint is from the human perspective, where a landscape is generally in the range of 10^3 to 10^6 ha (e.g. [150]). A landscape model is a simplified representation (e.g. as a digital raster map) of a landscape of interest, either real or theoretical, and is produced from natural (e.g. remote sensing) or artificial (e.g. simulation

modeling) sources. We must distinguish between three things, each of which may be viewed as a model: an *instance* of a landscape model refers to a particular map that represents a landscape, a *prototype* of a landscape model refers to a set of constraints on the generation of landscape models, and a *generator* of landscape models is a procedure for synthetically producing model instances from model prototypes.

The spatial scale of many landscape studies limits the ability to perform experiments in a traditional way: it is difficult to exert the required control for manipulative experiments, and hard, if not impossible, to find true replicates. With the increase in modeling related technology and techniques, many studies have used computer-generated landscapes both for artificial replication and for studying theoretical properties of idealized landscapes.

Research on the generation of landscape models can be classified in two main groups. The goal of one group has been to produce accurate prediction or duplication of the patterns seen in real landscapes (e.g. [59]). We refer to such model generators as *predictive*. The goal of the other group has been to generate landscape patterns that exhibit a simplified, but known, structure, and are generated by a random process. These types of generators have been termed *neutral models* since they are neutral with respect to ecological processes responsible for patterns observed in real landscapes [66]. The patterns that emerge in neutral models are the patterns expected in the absence of any ecological effects. Thus, neutral models can form a null hypothesis for testing for the effect of ecological processes on natural landscape patterns. A potential focus for hypotheses that relate ecological process and pattern is to explain the difference between neutral model patterns and patterns observed in real landscapes.

Work on neutral models has proceeded steadily over the last few years (e.g. [25, 66, 67, 148]), but is now rapidly expanding, as the number of presentations that focused on neutral models at a recent landscape ecology symposium testifies (e.g. [64, 73, 83, 100, 157]). However, although the development and use of neutral models and neutral model generators has proliferated, no unifying framework for organizing and categorizing models has emerged. Even the notion of a neutral model is becoming vague as neutral model generators are enriched with new features (e.g. [64, 65]).

We develop a general, and formal, view for artificial generation of landscape models. We define a *landscape model prototype* to be a set of constraints that restricts the generation of landscape models. Intuitively, a landscape model prototype is an abstract ideal of a landscape model, and can be viewed as specifying some characteristics of landscape models that are generated using this prototype. For example, a prototype may include restrictions to landscape indices (e.g. *richness* or *contagion*) or may be more complex, involving non-trivial spatial or temporal relations. Specialized generators must be developed to produce landscape models for different types of constraints. A variety of such generators already exist, and more are continually being developed.

Prototypes separate processes on landscapes into those aspects that account for the resulting pattern (i.e. the processes embodied in the constraints) from those that are not considered. The patterns that emerge from landscape model prototypes are the expected patterns in the absence of all ecological effects *not incorporated into the set of constraints*. Landscape model prototypes also form a null hypothesis for landscape patterns, and can be used for testing the effect of ecological processes acting on patterns in natural landscapes that are not accounted for in the constraints. Hypotheses may attempt to explain the difference between the patterns observed in the prototype instances and real landscapes.

A given set of constraints will generate a distribution of landscape models with expected characteristics, and may be deterministic or stochastically distributed. As the number of constraints increase, the expected pattern generated becomes more restricted, providing a gradient from simple models to more complex, predictive models. This relation forms a hierarchy, or partial order [38], on landscape model prototypes. The highest element of the hierarchy imposes no constraints on landscape structure and hence all landscape patterns have equal probability. We develop a framework within which this hierarchy of landscape models can be constructed, and describe its utility to landscape ecology for managing and analyzing sets of landscape models, landscape model prototypes and model generators.

Our framework provides a number of significant contributions to landscape ecology. First, by formalizing the abstract notion of a prototype, we provide a common ground upon which different generators can be compared. This not only may avoid re-developing existing generators, but provides a structure within which generators can be contrasted, and gaps identified. In addition, the resulting hierarchy provides a means for a common organization of landscape model generators, producing a structure for access to existing generators. Finally, the prototype hierarchy

can be used to guide the analysis of data sets of landscape models, assisting the identification of characteristics for which the data set deviates from random.

The next section develops the notion of neutral models, as introduced by Garder *et al.* [66]. This is followed by a definition of landscape model prototypes. Section 10.4 uses this formal description to construct a hierarchy of prototypes. Finally, we describe the potential uses of landscape model prototypes and the prototype hierarchy for landscape ecology.

10.2 Background: Neutral models

Landscape patterns may be represented using a two-dimensional array of cells, where each cell is occupied by some value, which we call a *landscape feature*. A patch is formed where adjacent cells are occupied by the same landscape feature. The neutral models introduced in Gardner *et al.* [66] are *whole mosaic models* [10] that are constructed using methods derived from percolation theory [137]. In their simplest form, each cell in the model is occupied by one of two distinct landscape features, which may differentiate, for example, community types that are susceptible or unsusceptible to disturbance. These models are specified by two parameters:

p : the fraction of the landscape occupied by one of the features
m : the linear dimension of the map (i.e. the length of one side)

By a simple random process, cells are occupied by feature 1 with a probability p, and feature 2 with a probability of $(1-p)$. These models are similar to landscape maps that have been classified into two categories, but are "neutral" with regard to the physical and biological processes that create real landscape patterns. Figure 10.1 shows three example neutral models for various values of p.

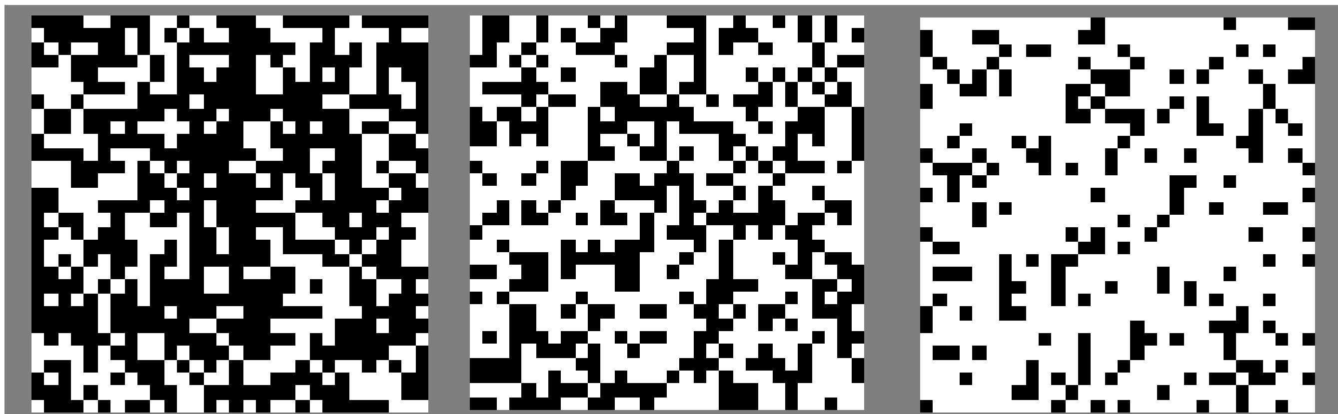

Figure 10.1: Example neutral models. Each instance was generated on a 30×30 grid (m=30), with varying proportions of the white feature (p = 0.4, 0.6 and 0.8).

Gardner *et al.* [66] used such simple neutral models to examine the effect of varying model size on patch size and shape in order to define appropriate scales for landscape analysis, and later Gardner *et al.* [68] examined effects on animal movements. Turner *et al.* [148] simulated disturbances on neutral landscapes with different proportions of susceptible habitat. The disturbances were modeled as random events that occur with a given frequency (probability of initiating) and intensity (probability of spreading to neighboring cells). They showed that the disturbance characteristic (frequency vs. intensity) primarily responsible for the propagation and extent of a disturbance depends on landscape connectivity (i.e. the value of p). In this last study, significant changes in model behaviour were detected near the *percolation threshold* (i.e. the value of p at which a patch of type feature 1 traverses the landscape model). In these simple neutral models, the percolation threshold occurs at a value of $p = 0.5928$ for very large models.

Gardner and O'Neill [67] introduced a contagion factor (see section 10.3) that can be used to create landscape models with larger contiguous patches while retaining the same relative proportion of features in the model. They

used these contagious landscapes to study the potential for movement and resource use by species living in patchy landscapes. They found that the percolation threshold varies inversely with contagion. Turner *et al.* [149] compared the results of simulating natural disturbance on real landscape models (Yellowstone National Park) with results from the same simulations run on neutral models that have an equivalent proportion of the fire susceptible community type. A number of these studies propose that significant departures by real landscapes from the expected patterns generated by a neutral model may be used to form and test hypotheses about the relationship between the observed patterns and ecological processes [66, 149].

Neutral models have a number of important uses in landscape ecology, some of which are mentioned below.

Comparison with real data. This is the main use endorsed by Gardner *et al.* [66] and Caswell [25]. Here, a neutral model is used as an ideal against which to compare real landscape data. Using a landscape statistics tool such as FRAGSTAT [99, 126], we can compute statistics that may differentiate between landscape patterns (e.g. average patch size, number of patches, patch adjacency, fractal dimension, contagion, etc.) [146]. Deviations from the neutral model permit an estimate of the effect of ecological interactions on the pattern observed in nature, and may lead to hypotheses regarding ecological processes responsible for these differences in pattern.

Testing broad-scale landscape hypotheses. Neutral models can be used to test hypotheses about landscape phenomena, such as the spread of disturbance and animal movements. The simplified structure of neutral models permit a clear analysis of how changing the parameter p effects the characteristics of interest. This is how neutral models were exploited in [68, 148]. Another use in this context is to analyze properties of neutral models themselves, using tools such as FRAGSTAT [99], in order to determine how the value of p affects the value of different landscape indices, such as average patch size.

Comparison with output from predictive models. Since we know the characteristics of neutral models, they are useful for comparison with the output from predictive models of landscapes. The difference between real landscape data and the predictions of a model are one measure of a model's ability to predict landscape patterns [66]. Neutral models provide a baseline that can be used to measure the improvement in predictability that is achieved by modeling geomorphological, climatic, biotic and other ecological effects.

Input to simulation models. Replication of landscapes is a difficult problem in landscape ecology. By specifying certain constraints, generation using neutral models provides a means of approximating replicates of landscapes with some specific characteristics (e.g. a fixed contagion). These artificial replicates can be used as input to landscape simulation models that generate new landscape models from a given input model (e.g. SELES [56]).

10.3 Landscape Model Prototypes

Our objective is to extend the core ideas of neutral models into a general framework for reasoning with landscape models that are artificially generated. The loose definition of a neutral model given by Caswell [25] is: "a neutral model is an expected pattern in the absence of specific ecological processes". Rather than focus on the *absence* (i.e. neutrality) of specific processes, we feel that models should be defined in terms of the *presence* of specific processes. That is, "a landscape model prototype is an expected pattern in the presence of specific constraints on that pattern". These *pattern constraints*, which we describe in detail below, dictate the expected pattern. We now give a formal definition:

Definition 10.1 *A* landscape model prototype *is a set of pattern constraints that restrict the possible generation of landscape models. An* instance *of a prototype is a landscape model generated under the set of constraints.*

Thus a landscape model prototype describes the expected pattern of a landscape and in essence gives a distribution of possible instances, which are particular landscape patterns generated using the given set of constraints.

10.3.1 Pattern constraints

There are many ways in which ecological information may be incorporated into landscape model prototypes. We have already seen two pattern constraints, as used in the simplest neutral models [66]: the *model size* m and the *landscape area ratio* (where landscape feature 1 had a relative distribution of p, and feature 2 had a distribution of $1-p$). In addition, these models restrict *richness* to the interval $[1,2]$. Thus, these models are random with respect to pattern, but always have a maximum richness of 2 and a landscape area ratio (LAR) for feature 1 normally distributed around p. The RULE program [65] permits the generation of models that precisely satisfy p. Richness, model size and LAR can be viewed as constraints on the patterns generated by these neutral models (i.e. they are not completely random). That is, a neutral model with $p = 0.4$ and $m = 30$ can be represented as a landscape model prototype with the constraints: $\{LAR = (0.4, 0.6),\ size = 30 \times 30,\ richness \in [1,2]\}$.

Additional constraints may be specified by restricting values of other landscape indices (e.g. *contagion* or *average patch size*), or by incorporating feature responses to spatially explicit landscape parameters such as elevation or soil type. We now discuss a number of constraints that can be imposed on the generation of landscape pattern. This list is not intended to be exhaustive. The example instances were generated using the spatially explicit landscape dynamics simulator SELES [56].

Constraints on bounds: Since a landscape model must be represented in a finite amount of memory, bounds on the grid size and maximum number of cell values are important. Restricting the grid size (i.e. the number of cells) is a fundamental constraint, and is related to the *extent* (i.e. the physical area represented by the entire model) and the *grain* (i.e. the physical area represented by each cell in the model) of the landscape of interest, where $extent = number\ of\ cells \times grain$.

Normally, each cell is represented by an integer, and so the number of potential cell values is bounded by the maximum size of integer that can be represented. In the case of the neutral models of Gardner *et al.* [66], each cell could be represented by a single bit, limiting the number of cell values to two (0 and 1). For instances generated from prototypes that specify only bound constraints, there will be no expected pattern; the feature in each cell is completely independent of all other cells, and hence no expected value (or expected distribution) can be predicted.

Constraints on landscape indices: In the literature to date, neutral models have been restricted to two landscape features (i.e. *patch type richness* is ≤ 2). We can extend this to any number of features, permitting richness in a range of values (e.g. $richness \in [1,5]$). For a particular application, each feature can be assigned different characteristics (e.g. to describe differential effects of a particular disturbance). In the context of percolation theory [137], instead of restricting each cell to either percolate or not percolate, varying degrees of percolation properties can be assigned to different cell types. For studies of the spread of disturbance in neutral models (e.g. [148]), this corresponds to permitting varying susceptibility to disturbance spread (e.g. fires or insect outbreaks) for each feature, as opposed to the simple binary properties of susceptible vs. unsusceptible. In the absence of contagion, this is very simple: for k features, we need to specify k relative abundance probabilities (which must sum to 1). A model containing at most k features can easily be generated.

We mentioned above that Gardner and O'Neill [67] propose contagion as a landscape index that may be used to constrain pattern generation for the two feature neutral models. However, when combined with an arbitrary richness constraint, the notion of contagion becomes more complex. In the two feature model, only one number was needed to represent contagion: an index indicating the probability that two adjacent cells will have the same feature. Now, in addition, we can specify contagion among different features.

To take a more concrete example, suppose our features are tree species. For a cell of type Douglas-fir *(Pseudotsuga menziesii)*, we may specify not only the probability that an adjacent cell is Douglas-fir, but also the probability that it is Western hemlock *(Tsuga heterophylla)*, Red alder *(Alnus rubra)*, etc. Thus we have k^2 contagion values to specify. In some situations, it may be difficult to have precise ecological data to specify this accurately. We can simplify matters by only requiring one contagion value c that specifies the probability that adjacent cells will have the same feature. That is, for each pair of identical features (e.g. Douglas-fir next to Douglas-fir), the contagion value is c, and for each pair of different features, no contagion is specified.

Simultaneously preserving the probability distribution (i.e. LAR) and contagion is not trivial, but can be accomplished by a formal generalization of contagion, which we develop in the appendix at the end of this chapter. Examples of landscape instances generated using different values for contagion are shown in Figure 10.2. All three models have four features with equal relative proportions (0.25).

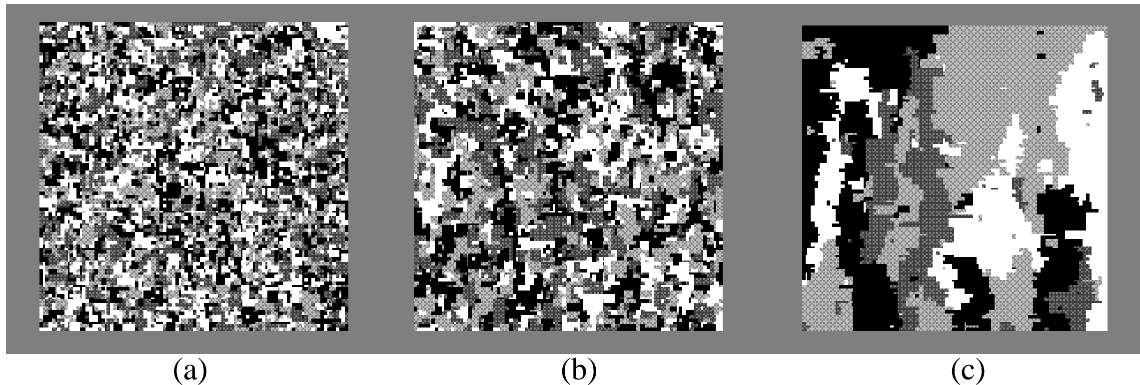

Figure 10.2: Instances of landscape model prototypes produced on a 100×100 grid. Each model has four features with equal landscape area ratios (i.e. equal relative proportions). The value of contagion differs for each model instance, taking on the values 0.6, 0.8 and 0.99, respectively. The prototype for instance (a) is therefore $\{LAR = (0.25, 0.25, 0.25, 0.25)$, $size = 100 \times 100$, $richness \in [1, 4]$, $contagion = 0.6\}$.

Although contagion is an intuitive and common index for landscapes, there is nothing ecologically inherent that distinguishes it from other indices. We could, in theory, restrict the value of any landscape index to constrain possible landscapes. For example, we could set *Shannon's diversity index* or *edge fractal dimension*, and only generate landscapes that have a particular expected value for these indices. Furthermore, we could specify restrictions to more than one landscape index simultaneously, and generate landscapes that satisfy all the values of these indices. In this way, we view landscape model prototypes as models that are *not neutral* with respect to a given set of explicit constraints (landscape indices in this case), but neutral with respect to everything else.

Spatial constraints: There is no mechanism in the models of Gardner *et al.* [66] to incorporate the effects of physiography when generating landscape models. The distribution of real landscape features may be strongly influenced by some physical characteristics of the landscape, and we may want to integrate them into model generation. We can incorporate responses to spatial parameters (e.g. topography, soil type, slope, etc.) as constraints on the probability distributions of features, providing a spatial context for pattern generation. Such parameters affect both the relative proportion and spatial distribution of the features in the model. We call such models *site specific* due to the local effect of parameter values at a given site. This use of spatial parameters essentially replaces a statistical approach to spatial distribution with a more empirical based, process oriented approach.

These parameters can be derived from real data, or can themselves be artificially generated. For example, a topography parameter can be derived from a real landscape through cartographic techniques, or it may represent a theoretical topography derived through *fractal model* generation (e.g. [56, 116, 117]). Spatial parameters are matched to the landscape model, so that each cell in the landscape model has a corresponding value in the parameter model.

Generating a site specific model involves calculating, for each cell, the relative probability of occurrence for each feature. This is akin to deriving a local LAR. This information is then used to either randomly determine a feature for the cell, based on this distribution or it can be further constrained (e.g. with contagion). Note that as prototypes become "less" neutral, the significance of contagion in forming patches decreases. Contagion can be viewed as the aggregation of ecological processes that explain why features are often grouped into patches. As these ecological processes are integrated into a model through spatial constraints, the need for a contagion factor decreases, since features will become more naturally aggregated.

These site specific models can range from more neutral models (i.e. site independent, aspatial distributions of landscape features) to complex models that specify relationships for many parameters. This extends our notion of gradients of neutrality, from prototypes that specify aspatial constraints, to prototypes that incorporate a spatial

context that influences pattern generation, taking one more step towards predictive models.

Figure 10.3 shows an instance of a site specific model for which features vary with altitude. Each of the five features differs in its response to elevation. The darker features respond "better" to lower elevation, while lighter features respond "better" to higher elevation. That is, at low elevations, the relative probability of darker features is higher than lighter features, and vice versa at high elevations.

The model instance is draped over the elevation map that was used to create it, providing a contextual view of the instance. Note that using the same set of constraints, but a different elevation map, would produce a different model instance. In this example, no contagion was used.

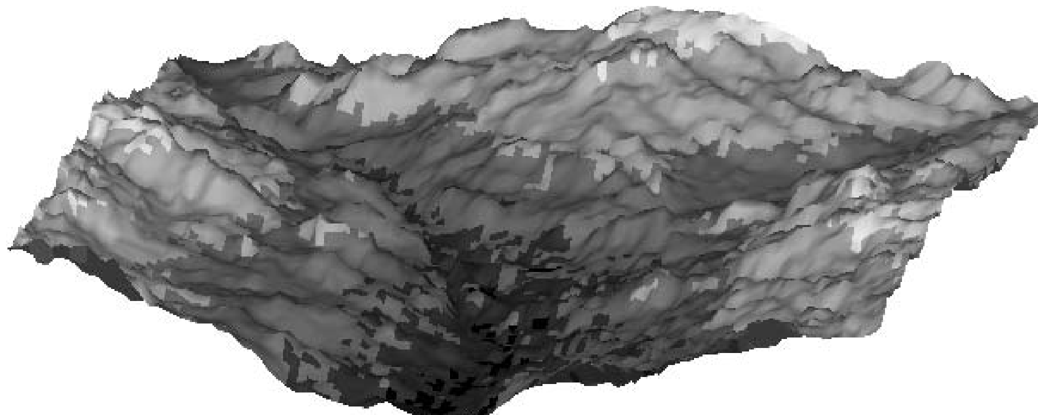

Figure 10.3: Geometric view of an instance of a landscape model prototype with spatial constraints. The instance is overlaid on the elevation model used to create it. The model size of this instance is 100×100, and the number of features is 5. The underlying elevation model provides a context in which spatial constraints, in the form of elevation responses, affect pattern generation. Thus, the prototype for instance (a) is $\{size = 100 \times 100$, $richness \in [1, 5]$, *spatial responses to elevation*$\}$.

Temporal constraints: We can also constrain pattern generation temporally through the use of an existing model instance. If we view the existing instance as a previous state of the landscape, this creates a temporal context for pattern generation. Using a combination of the input landscape model, and temporal change sequences (e.g. modeling succession or disturbance), a landscape simulator may attempt to mimic ecological and/or abiotic processes in the production of landscape pattern in the output model.

Specifying temporal constraints may be as simple as providing a Markov chain [10] (i.e. a *transition* matrix, where entry (i, j) specifies the probability that a cell with feature i in the input model will have feature j in the output model). At the other extreme, temporal constraints may determine the features of the output model based on an analysis of the input pattern, and possibly other information such as spatial parameters. Depending on the complexity of the constraints on temporal sequences, these prototypes may also provide a gradient from models that are a small step beyond neutral models to more predictive models.

Figure 10.4 shows an instance of a prototype (pattern (b)) generated using temporal constraints and an input model (pattern (a)). The temporal sequence is stochastic, and most of the cells obtained their feature from the previous state; some of the cells (most notably in the centre left of the pattern) obtained different values. In general, such sequences may model a successional trajectory, the effect of a disturbance event, or some other dynamic landscape process. The specification of temporal constraints, and the generation of sequences of models based on these constraints is the heart of landscape dynamics simulators, such as SELES [56]. Note that the only constraints involved in the generation of this model instance are *richness*, *model size* and *temporal responses*; the resulting pattern is largely dependent on the input landscape.

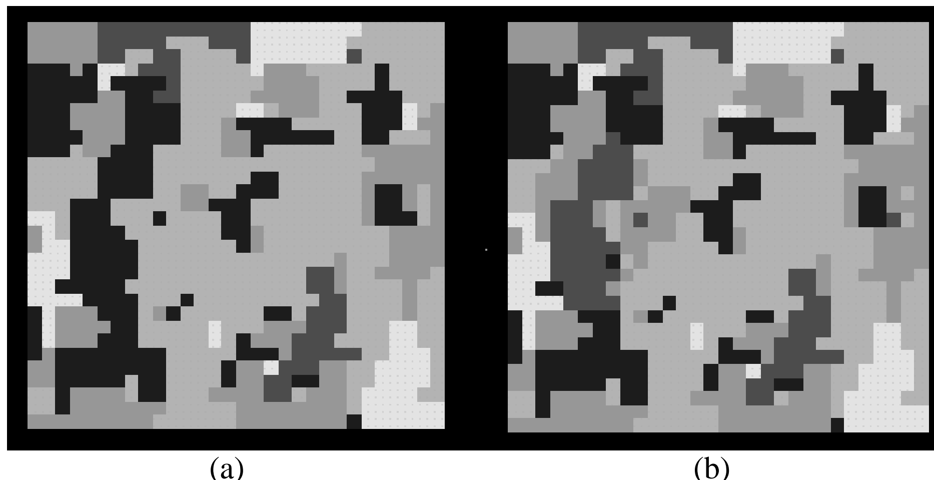

Figure 10.4: Instance of a landscape model prototype (b) generated using stochastic temporal constraints and input pattern (a). The model size is 30×30, and richness is 4. The prototype for instance (b) is therefore $\{size = 30 \times 30$, $richness = 4$, *temporal responses*$\}$.

10.4 A Hierarchy of Landscape Model Prototypes

Different combinations of constraints lead to different landscape model prototypes, and the relation among these prototypes forms a hierarchy. For two prototypes, $\mathcal{P}_1$ and $\mathcal{P}_2$, if $\mathcal{P}_1$ is defined by a superset of the constraints of $\mathcal{P}_2$, then instances generated by $\mathcal{P}_1$ are more restricted than those generated by $\mathcal{P}_2$. In this case, we place $\mathcal{P}_1$ "lower" in the hierarchy than $\mathcal{P}_2$. The most general prototype, denoted $\top$, is the one that imposes no constraints on pattern generation. Although such a prototype may have limited practical utility, it does serve as a common starting point for all other prototypes. The prototype hierarchy forms a general partial order not just a tree shaped hierarchy, since a prototype may have multiple parents.

This hierarchy provides a framework for systematically cataloging and analyzing landscape pattern. A prototype can be used to generate a set of instances with an expected pattern under known constraints. Deviations from this expected pattern in real landscapes, or simulation results, can help us identify components of pattern not explained by the constraints of the prototype.

Figure 10.5 shows a sample fragment from this hierarchy. Each node in the hierarchy includes the set of constraints imposed by all nodes above it. Thus, the lowest node represents the prototype with the constraints: $\{$*richness = 4, model size = 100, LAR = (0.1,0.2,0.3,0.4), contagion = 0.8, spatial responses to elevational data*$\}$. The other nodes in the example contain various subsets of these constraints.

The prototype hierarchy organizes work on neutral models and landscape model prototypes both for developers and users of model generators. Some of the potential applications of the hierarchy are described below.

Development of landscape model generators: Landscape model generators are procedures for the synthetic production of instances of landscape models. In general, they permit the specification of prototypes via parameter values. Once a set of parameters (constraints) has been provided, landscape instances satisfying those constraints can be produced. Thus, generators are more abstract than prototypes in that they only restrict *which* constraints may be specified, while prototypes also restrict the *value* of the constraints. Our framework provides a structure within which landscape model generators can be systematically developed and compared. Not only can two generators be contrasted as to which constraints may be specified, but gaps in the suite of existing generators can be identified. In this viewpoint, the hierarchy does not specify values for constraints. The constraints that may be imposed by a generator determine its position in the hierarchy, and its relation to other generators.

A sample fragment of the generator hierarchy is shown in Figure 10.6. Each node represents a generator that allows specification of the constraints attached to that node and all nodes above it in the hierarchy. For example,

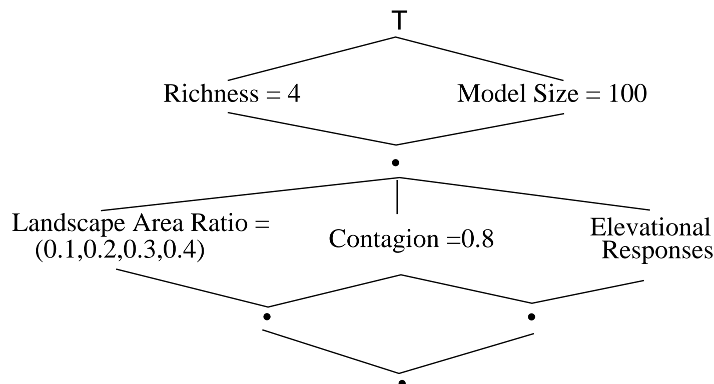

Figure 10.5: Sample fragment of the hierarchy of landscape model prototypes. Each node represents a prototype that consists of the constraints labeling the node and all higher nodes in the hierarchy.

the node labeled *Edge Fractal Dimension* permits specification of *richness*, *model size*, *landscape area ratio*, and *edge fractal dimension*. The node below *Richness* and *Model Size* represents a "totally neutral model", where only bound constraints are specified. Note this fragment is incomplete, and is not intended to suggest any particular relations among constraints. Thus, for example, there may be another node above the one labeled *Edge Fractal Dimension* that permit specification of *edge fractal dimension*, but does not require *landscape area ratio*.

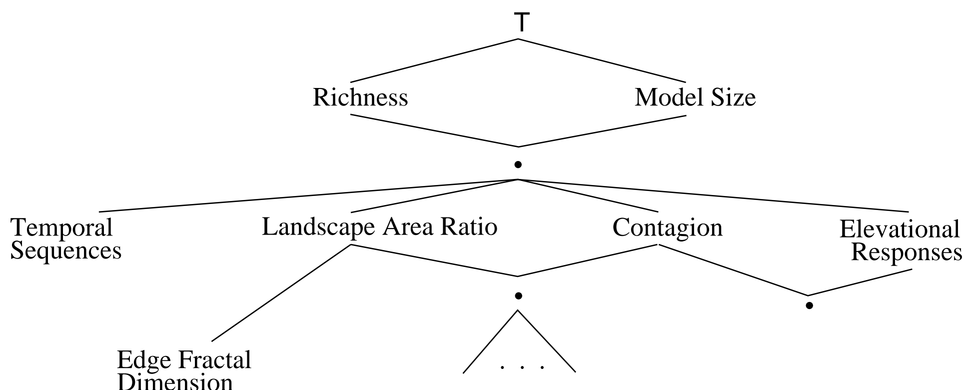

Figure 10.6: Sample fragment of the hierarchy of landscape model generators. Each node represents a generator that permits specification of the constraints labeling the node and all higher nodes in the hierarchy.

A common organization of landscape model generators: Access to existing tools is a prevalent problem. Currently, developers of landscape model generators have no source of information as to which generators already exist, and so run the risk of re-inventing the wheel. Similarly, potential users of generators have no systematized way of searching for generators. The prototype hierarchy has the potential to alleviate these problems as a common organization of model generators. A site on the Internet could be established to maintain the hierarchy, and nodes could have links to sites from which the corresponding generator can be accessed. Thus, once the desired set of constraints has been identified, the hierarchy could be traversed, and if the node corresponding to these constraints could be found, then the landscape model generator exists and can be accessed.

With time, this hierarchy may potentially grow to a size where access becomes cumbersome. In this case, the encoding techniques developed previously in this thesis for efficiently storing and traversing hierarchies could be utilized. A user could enter the desired set of constraints, and the system would automatically find the desired node if it exists. If no such generator exists, then the set of most closely related nodes could be returned.

We envision the prototype hierarchy as providing a cooperative resource for landscape ecologists to share landscape model generators, to find desired generators, and to identify gaps in the current state of landscape model generation.

Analysis of landscape pattern: Landscape ecologists benefit directly from the hierarchy of landscape model prototypes. Given a data set of one or more landscapes, the hierarchy can guide hypothesis testing to determine the level of neutrality of the data set. That is, we can find the node $\mathcal{P}$ in the hierarchy for which the models generated by prototype $\mathcal{P}$ are not significantly different from the models in the data set. We provide below a theoretical example of how this can be accomplished. The process of arriving at $\mathcal{P}$ may identify deviations from random, or neutral, characteristics. This in turn may lead to hypotheses to explain these differences. The node itself is also of interest, since it is the most general prototype that captures the pattern exhibited in the data set, establishing the "level of neutrality" of the data set. That is, this prototype serves as a predictive model for the data set, and is the most general such prototype.

For example, suppose we have a data set $\mathcal{X}$ of landscape models with size m and richness k. Starting near the top of the hierarchy, we can take a basic prototype with constraints only on model size and maximum richness. Using this prototype, we can generate a number of model instances, which can be used as a random sample of the prototype. Now we can compare an attribute of the data set, such as the average contagion, with that of the sample. Note that the contagion for the sample provides an expected value for contagion in the absence of additional ecological information.

If no significant difference can be detected between the contagion of the data set and that of the sample from the prototype, then the data set has a contagion value that is indistinguishable from random. This isn't to say that there is no process in these landscapes acting on this attribute, but rather that we cannot distinguish from pattern that is random with respect to this attribute. We can continue by selecting another attribute, such as LAR or elevation responses.

If, however, we find that the attribute value for $\mathcal{X}$ differs significantly from the expected value of the attribute, then there is some process responsible for this divergence. The identification of this deviation may lead to hypotheses for explaining the difference. For example, if the average contagion for the data set $\mathcal{X}$ is greater than the average contagion for the sample from the prototype, then this indicates that there is some ecological process responsible for the higher degree of aggregation in the data set than is expected from random. This may lead to a hypothesis to explain the aggregation seen in the data set.

We can now continue this process by taking a more constrained landscape model prototype that restricts model size, maximum richness and contagion. In this way we are able to systematically exploit the model generators available in order to classify a landscape on the neutrality gradient, and generate hypotheses to explain deviations from random. If we find a prototype $\mathcal{P}$ for which all attributes of the data set are indistinguishable from the instances produced by $\mathcal{P}$, then this prototype not only identifies the level of neutrality for the data set, but it can also serve as a predictive generator for the data set (at least for the attributes tested during this analysis).

10.5 Conclusion

We have formalized landscape model generators using the notion of a landscape model prototype, which is a set of constraints that restricts the generation of pattern. These prototypes induce a hierarchy that provides a formal framework within which model generators can be constructed, compared and accessed. This hierarchy can be used to guide the analysis of pattern from a data set of landscape models, and captures the idea of "gradients of neutrality". That is, prototypes provide some measure of distance from neutrality, and the hierarchy embodies the variety of ways in which models can diverge from random in a multi-dimensional space of possible constraints on pattern generation. Analysis of data sets of landscape models can exploit this hierarchy to guide identification of differences between the data set and random. In addition, we described how one can determine the node in the hierarchy for which there are no significant differences between the models generated by the prototype represented by this node and the models in the data set. This not only establishes the level of neutrality for the data set, but also the prototype at this node acts as a predictive model for the data set.

Chapter Appendix: Formal Basis for Landscape Model Generators that Permit General Richness, LAR and Contagion Constraints

In this appendix, we provide a mathematical derivation for landscape model generators that can satisfy general constraints on richness, landscape area ratio and contagion. Gardner and O'Neill [67] provided the mathematical basis for combining *landscape area ratio* (LAR) and contagion for models with a richness of 2. However, their results do not permit a direct generalization to an arbitrary number of landscape features. Our goal is to provide a means of generating landscape models that satisfy constraints on richness, LAR and contagion. Clearly, not all possible combinations of constraints are satisfiable. For example, the constraints $richness = 2$ and $LAR = (1.0,0.0)$ imply that *contagion* must be 1. Even though these constraints are not completely independent, we can attempt to satisfy the contagion constraint while maintaining the richness and LAR constraints. Here, we provide a formal derivation for this landscape model generator.

In general, for k features, there can be up to k^2 contagion factors, where contagion factor c_{ij} can be viewed as a probability index that a cell of feature i is next to a cell of feature j. This can be specified using a $k \times k$ array C_{kk}. Each contagion factor c_{ij} may take on any value in $[-1, 1]$, where a value greater (less than) than 0 denotes that a cell of feature i is more (less) likely to be next to one of feature j than random. A value of 0 denotes that a cells of feature i and j are juxtaposed randomly. That is, the probability that feature j is next to feature i is the same as the relative abundance of feature j in the entire model. We minimally require one contagion factor c that is assumed to be the contagion for adjacent cells of the same type. This is the situation we used in section 10.3 for examples. We generalize this somewhat, and permit a vector of k contagion factors C_k, where c_i denotes the probability index that a cell of feature i will be next to another cell of feature i. The other contagion values (i.e. $c_{ij}, i \neq j$) are assumed to be 0.

The relative abundance vector P_k (i.e. LAR for each of the k features) must clearly sum to 1 (i.e. $\Sigma_{i=1}^{k} P_k[i] = 1$). Our algorithm for constructing the contagion matrix takes as input the relative abundance vector P_k and a contagion factor vector C_k. Our goal is to generate a contagion matrix Q_{kk}, where each element q_{ij} is the probability of feature j being adjacent to feature i, and Q_{kk} somehow satisfies the LAR P_k. In the case of no contagion, each row of Q_{kk} will be identical to P_k. As contagion is changed (either increasing or decreasing clumping), we must change the entries of Q_{kk} to reflect this change while still satisfying the relative abundance requirements in P_k over the entire landscape. Note that if $q_{ii} = p_i$, then feature i will not be directly affected by contagion. If $q_{ii} > p_i$, then feature i will be more clumped than random and if $q_{ii} < p_i$, feature i will be less clumped.

In the two feature case, changing contagion while maintaining P_k was simple to achieve, and the mathematics is given in [67]. Their specification of the problem was difficult to generalize, so we look at it slightly differently. First, we need to formalize what we mean when we say that a contagion matrix Q_{kk} "satisfies" P_k. Our algorithm for constructing a contagion matrix Q_{kk} starts with each row of Q_{kk} identical to P_k. Clearly, using this matrix to generate a landscape will be the same as using P_k alone. We then transform Q_{kk} so that P_k is always satisfied and at the end, Q_{kk} reflects the desired contagion factors.

Definition 10.2 *Suppose we have k features and a relative abundance vector P_k. Then a contagion matrix Q_{kk}* satisfies *P_k if and only if $\Sigma_{j=1}^{k} p_j * q_{ji} = p_i$.*

If this equation is satisfied, then the overall probability that a cell will have feature i (i.e. p_i) will be the same as the sum of the probabilities that an adjacent cell will have feature j times the probability that feature i will be next to feature j. One property that we require of any contagion matrix (as we do for the relative abundance vector) is that the sum of the probabilities in any row must be 1 and that all probabilities must be non-negative.

Lemma 10.1 *Suppose we have we have k features and a relative abundance vector P_k. Then a contagion matrix Q_{kk} satisfies P_k if, for all $1 \leq i, j \leq k$, $p_i * q_{ij} = p_j * q_{ji}$.*

Proof: Suppose the above property is satisfied. Consider any feature i. Then $\Sigma_{j=1}^{k} p_j * q_{ji} = \Sigma_{j=1}^{k} p_i * q_{ij} = p_i * \Sigma_{j=1}^{k} q_{ij} = p_i$, since any row of Q_{kk} must always sum to 1. □

In the initial state $q_{ij} = p_j$, so this property is satisfied. We now show that we can perform transformations on Q_{kk} that preserve this property.

Theorem 10.1 *Suppose we have we have k features, a relative abundance vector P_k and a contagion matrix Q_{kk} that satisfies P_k. Given some $1 \leq i, j \leq k$ and a factor α such that $max(-q_{ii}/q_{ij}, -q_{jj}/q_{ji}) \leq \alpha \leq 1$, then after the following transformation, Q_{kk} still satisfies P_k:*

$q_{ii} = q_{ii} + \alpha * q_{ij}$
$q_{ij} = q_{ij} - \alpha * q_{ij} = (1 - \alpha) * q_{ij}$
$q_{jj} = q_{jj} + \alpha * q_{ji}$
$q_{ji} = q_{ji} - \alpha * q_{ji} = (1 - \alpha) * q_{ji}$

Proof: Since only the above four entries are modified, we need only ensure that the property of the above lemma is satisfied. For the diagonal elements, this is trivially satisfied: $p_i * q_{ii} = p_i * q_{ii}$ and $p_j * q_{jj} = p_j * q_{jj}$.

For the other two elements, we must satisfy: $p_i * q_{ij} = p_j * q_{ji}$. By our assumption, this property holds before the transformation. After the transformation, we have: $p_i * ((1 - \alpha) * q_{ij}) = p_j * ((1 - \alpha) * q_{ji})$. Dividing both sides by $(1 - \alpha)$ yields the desired result. □

The proof does not depend on the restriction to the value of α. This restriction ensures that the entries in Q_{kk} remain non-negative. If $\alpha > 1$, then q_{ij} and q_{ji} become negative and if $\alpha < -q_{ii}/q_{ij}$ or $\alpha < -q_{jj}/q_{ji}$ then one of q_{ii} or q_{jj} becomes negative.

Given a relative abundance vector P_k and a contagion factor vector C_k (both of size k), the contagion matrix Q_{kk} can be computed as follows: Start with each row of Q_{kk} equal to P_k. For each contagion factor, c_i perform the above transformation on Q_{kk} (where α becomes c_i). Once we have Q_{kk}, the landscape model instance N_{mm} can be easily generated as follows:

1. For the first cell n_{00}, select a feature randomly using P_k.
2. For each cell n_{i0} in rest of the first row, select a feature randomly using the row of Q_{kk} corresponding to the left neighbour.
3. For each subsequent row:
   (a) For the first cell n_{0j}, select a feature randomly using the row of Q_{kk} corresponding to the neighbour above in the map.
   (b) For each remaining cell n_{ij} in the row, using the average of the rows of Q_{kk} corresponding to the neighbour left and above.

This algorithm will tend to have a diagonal bias, which can be partially alleviated by alternately traversing rows left and right. This will still leave a slight vertical bias, but not very pronounced except at high values of contagion. Other generation techniques may be possible to generate maps using the contagion matrix, but without any bias. The model instances shown in [67] have a clear horizontal bias, and must have been computed without considering the vertical neighbours.

Chapter 11

Conclusion

"There is in nature what is within reach and what is beyond reach"

– Goethe

Reasoning is a fundamental problem in a variety of human intellectual endeavors. Taxonomies assist the reasoning process by clarifying and categorizing knowledge. This thesis is an attempt to bring taxonomic reasoning to centre stage, and to push forth some of the frontiers of research. From a pragmatic viewpoint, we have formalized research on managing large taxonomies, a task known as *taxonomic encoding*. Our formal framework encapsulates the essence of encoding and we are able to characterize all known encoding techniques within it.

During our analysis of encoding, we developed sparse logical terms as a universal implementation for encoding. We explored the utility of sparse terms for encoding, both theoretically and empirically.

Although partial orders are an elegant and mathematically formal basis for representing taxonomic knowledge, we became dissatisfied with their limited expressive ability. Rather than shift to the other extreme, where taxonomic information is hidden within a description logic (such as KL-ONE) and can only be extracted via classification, we feel that explicit maintenance of taxonomic knowledge is essential for taxonomic reasoning. To pursue this line of thought, we formally extended partial orders to incorporate additional information, and developed a sort logic for reasoning within this more expressive framework. To maintain tractable reasoning, we also derived a restricted form of the logic.

In the course of this thesis, it became apparent that taxonomies were prevalent in almost every field. We followed shallow explorations of a number of applications, such as natural language processing, and delved deeper into three of the fields that are rich with possibilities.

Research on using logical terms for encoding led to a viewpoint that coreference in logical variables imposes requirements that are too strict. By viewing the symmetry of coreference as the product of two asymmetric *reference* constraints, a taxonomy may be constructed, where each node represents an equivalence class of variables (i.e. variables that corefer). In current logic programming systems, variable coreference classes are constructed, but cannot be related to one another.

Conceptual structures was the first field to which our initial research on encoding was applied. It became apparent that encoding has a great potential impact on the field due to the variety of (potentially large) taxonomies that are used in the formalism. In addition, our research led us to further application of sparse terms to implement normalized conceptual graphs.

The final area of application for this thesis is ecological modeling. Although hierarchies have been used in a number of domains, we applied taxonomic reasoning to unbroken ground in landscape ecology. By formalizing a hierarchy of landscape models, we have been able to bridge the gap between predictive and theoretical models of landscapes, to provide a framework within which generators of landscape models can be designed, compared and accessed, and to guide analysis of sets of landscape data.

11.1 Significance of Research

The overall goal of this thesis was to forge ahead with research on reasoning with taxonomies, to develop a formal foundation upon which systems that use taxonomies can rest, and to apply the theory to a variety of applications. The research that comprises this thesis has had a number of impacts on several fields, as outlined below:

1. The theoretical work on encoding has provided a foundation on which different encoding algorithms and techniques can be compared and critiqued. Prior to this development, encoding research was somewhat ad hoc, with no context or means to critically evaluate advances in the field. The notion of a spanning set for separating the information content of an encoding from the implementational details provides a yardstick for the addition of new techniques, and avoids the potential problem of re-inventing the wheel.

2. Our contributions to modulation provide the potential to improve further the efficiency gained from using this technique. Furthermore, our generalization of modulation extends the elegance of modulated encoding into the realm of practical encoding with dynamic and irregular taxonomies. By relaxing the notion of a module, the effort involved in modulation can degrade gracefully over time, rather that break in brittle mathematical precision. We have also provided proven algorithms that permit the computation of taxonomic operations in generalized modules.

3. Our constraint based view of encoding provides a guideline for the use of coreference (i.e. logical variables) in encoding. By providing a formal analysis of encoding in terms of constraints, we have shed light on the advantages and pitfalls of going beyond tree terms for logical term encodings.

4. The theoretical and empirical results of sparse term encodings place sparse terms as a universal encoding implementation. The general form of sparse term developed for encoding directly subsumes most other encoding implementations (e.g. integer vectors, logical terms, interval sets), with the exception of bit-vectors. The empirical evidence provided by encoding two medium size taxonomies from existing applications, however, shows how sparse terms let us have our cake and eat it too. Sparse terms used significantly less space than bit-vectors, while providing the flexibility required for dynamic updates to encodings (i.e. partial re-encoding).

5. Our work on extending partial orders separates the task of taxonomic, or sort, reasoning from applications that use taxonomic information. The sort reasoner is provided with taxonomic knowledge in the form of assertions, and can be called upon to answer queries regarding the taxonomic structure specified. We developed a sound and complete sort logic as a logic for reasoning *about* sorts (as contrasted with sorted logic for reasoning *with* sorts). To find utility in practical systems, sort reasoning must be efficient. One of our main contributions is the development of a tractable restriction of the sort reasoning problem that retains enough expressive power to capture many common forms of taxonomic knowledge.

6. Our development of reference constraints as a generalization of equality constraints in logic and logic programming is a novel application of reasoning with taxonomies. Although equality constraints form equivalence classes of logical variables, reference constraints induce a partial order among these coreference classes. We provided a formal description of how reference constraints may be specified in a logic program, and how the resulting reference order can be maintained and satisfied.

   Since variables denote individuals, reference constraints lead to the notion of *individual level inheritance*, where an individual denoted by a variable may inherit properties from another individual which is denoted by a subsuming variable in the partial order. A variety of systems, especially systems reasoning in ambiguous domains, can potentially benefit from an efficient, formally based implementation of reference constraints and individual level inheritance.

7. The issues involved in maintaining derived hierarchies, such as the generalization hierarchy of conceptual graphs, differ from encoding issues for defined hierarchies, such as class or sort hierarchies. Derived hierarchies may be induced by the set of data (graphs) in a knowledge base; they are highly dynamic and expensive to compute. Focusing on the field of conceptual structures, we developed an approach to normalize graph knowledge bases

and store the graphs in a spanning tree of the underlying partial order. The advantages of normalizing within this spanning tree are twofold: (i) the normalization of a graph can depend on its parent in the tree, so that traversals within the tree can be much more efficient than traversals in the general partial order; (ii) there are a number of benefits of traversing such hierarchies in a topological fashion (e.g. more rapid retrieval of a target graph), as covered in [42]. However, there are a variety of topological traversals; the one described in [42] is breadth-first. We argued that there are benefits to depth-first topological traversals, and we showed that if the spanning tree is formed as a left-to-right depth-first traversal of the original partial order, then a right-to-left depth-first traversal of this tree corresponds to a right-to-left depth-first topological traversal of the partial order.

8. Artificial generation of landscape models is becoming increasingly prevalent in landscape ecology. Due to the spatial scale at which most landscape studies are performed, replication is rarely feasible and experimenters may require artificial replication. Artificial generation of landscape models can be used for a variety of purposes, including comparison with real data, testing general theoretical hypotheses, and providing inputs to simulation models. However, the number of generators is increasing and there is no framework within which generators can be analyzed, compared and organized. We proposed a hierarchical framework that unifies landscape models within a formal organizational system. By generalizing neutral landscape models, we proposed landscape model prototypes that induce a hierarchy that represents gradients of neutrality. We described how this hierarchy may be used to guide the development of landscape model generators, to aid selection of appropriate existing model generators, and to assist in the analysis of models derived from real landscapes through the use of landscape model prototypes.

11.2 Future Research Directions

"The solution to every problem is another problem"

– Goethe

The research presented in this thesis has contributed to a number of disciplines and made a variety of connections among fields. It has also opened many doors and identified unexplored pathways which were beyond the scope of a single thesis. This final section of the thesis identifies some promising areas in which research can be continued.

Encoding. Using our notion of spanning sets, further theoretical work should be carried out on the limits of taxonomic encoding. Research continues to push the frontiers in the quest for minimal size encodings (e.g. [79]), and we maintain that the framework provided in this thesis is an appropriate common ground on which new techniques should be evaluated. More empirical testing of different encoding algorithms and implementations should be done. As more taxonomies from real applications become available, this will become easier to perform.

Modulation. Although the advantages of modulation are intuitive, there is a real need for empirical testing of its actual benefit, and for determining at what size of taxonomy should modulation be attempted. We expect that the benefits of modulation will not show up until taxonomies are quite large, but that this technique will address issues of scaling encoding up to much larger taxonomies than are currently encountered. Finally, to address issues of efficiency, there is a need to integrate the linear time modulation algorithm of [76] with our techniques, which may require changes to this fast algorithm to accommodate our generalized forms of modules.

Sparse Term Encoding. Further theoretical and empirical testing of different encoding techniques is required to provide a strong basis for comparison of sparse term encoding with other implementation schemes. Also, additional work on sparse term encoding should be researched to implement and test the utility of encoding in highly dynamic environments.

In the theoretical arena, there are a number of dimensions along which comparisons can be made. We selected two techniques that we felt appropriate for encoding dynamically changing taxonomies (transitive closure and compact), and compared the effects of different implementations on these techniques. One advantage of

our framework for encoding is that it makes possible such comparisons. Another approach, taken in [43], is to compare different algorithms (that mix technique with implementation). There is a great need for more comparisons of these kinds, to identify the types of taxonomies that are best suited for different approaches to encoding.

Extending Partial Orders. Although we have developed a theoretical foundation for tractable sort reasoning in Chapter 7, this work needs to be implemented, and empirical testing can identify the utility of our restrictions to obtain tractability. Other sets of restrictions can also be developed and contrasted with our proposal to develop an efficient sort reasoner.

Also, more efficient encoding techniques that take advantage of the structure of extended partial orders should be developed. For example, two incompatible sorts can share the same position within a term, leading to unification failure if an object is postulated to belong to both sorts. This opens a whole area of research for generalizing our spanning set framework for encoding extended partial orders.

Data Mining. Tree-shaped conceptual hierarchies have been proposed for use in data mining [13, 81, 82]. There exists a great potential for generalizing these techniques to use partial orders, and even extended partial orders.

Reference Constraints. To fully demonstrate the utility of individual-level inheritance, reference constraints must be implemented in a logic programming system. Possibilities include implementation in sparse terms or another logic programming language, such as LIFE [4] or Bin Prolog [140]. A variant of sparse terms has been implemented that includes coreference akin to that in LIFE [4]. This variant could be extended in a straightforward manner to handle reference constraints. In addition, the effects and advantages of different control strategies as mentioned in Chapter 8 should be explored.

Also, applications of hypothetical reasoning such as those outlined in this thesis need to be more thoroughly developed and implemented. The application of individual-level inheritance as a means to integrate top-down hypothetical analysis and bottom-up chart parsing in discourse processing appears to be a promising area to pursue in this direction. In addition, the incorporation of reference constraints into Assumption Grammars [142] for natural language processing should be studied.

Conceptual Structures. As implementation of the Peirce workbench [44] and other systems for reasoning with conceptual graphs proceeds, there will be opportunities to implement and compare the various approaches to handling taxonomies of complex and dynamically changing information, such as graph knowledge bases. Empirical testing of the advantages of the spanning tree organization for the generalization hierarchy compared to other organizations of complex data (e.g. [42]) must be performed.

Landscape Model Prototypes. Using the hierarchy of landscape model prototypes, existing model generators can be placed in relation to each other. The next step is to use this hierarchy to provide a common organization for model generators, and to organize existing and future generators for simple access by users. The internet is a natural location to place such a hierarchy; a proposal in this direction is in progress.

Landscape studies need to attempt to use the hierarchical techniques proposed to guide the analysis of data sets of landscape models. Studies that compare data sets against landscape prototypes will identify gaps in the suite of available generators.

Analysis of Landscape Models using Formal Concept Analysis. The hierarchy of landscape model prototypes developed in Chapter 10 permits analysis of the properties of an entire data set in comparison with artificially generated models. Other techniques are necessary for the analysis of the properties of individual models in comparison with other models in a given data set. The issues addressed here are quite different, and focus more on how the models in a data set can be differentiated and/or grouped. Such analysis is complex, and researchers have proposed a multitude of indices for the comparison of landscape models in a data set [126]. An attempt to select a core subset from this array of indices has been explored in Riitters *et al.* [122]. However, attempts to derive a core set of indices that is independent from a data set fail to recognize that different sets of landscapes have inherently different properties.

We propose an alternate approach for reducing the set of potential indices through the use of *formal concept analysis* [153]. Formal concept analysis is based on a mathematical, set-theoretic model of *concepts* and *conceptual hierarchies* [62, 155]. It was developed as a new approach to data analysis that permits structural analysis of data without reducing the data. Concept analysis provides a formal, objective, data-driven technique for automatically constructing a hierarchy of relationships from a set of *objects* (e.g. landscape models) and a set of *attributes* (e.g. landscape indices). This hierarchy, known as the *formal concept lattice*, elucidates relationships inherent in the data, and can aid in the selection of key indices for a given set of landscape models. Formal concept analysis has been applied to a variety of domains with many nice results (e.g. analysis of Rembrant paintings [155], comparison of recreation opportunities in national parks [139], and information retrieval [29]).

In general, a concept lattice provides a hierarchical conceptual clustering of the objects, and also represents all the implications among the attributes [155]. Using the techniques of formal concept analysis, we can automatically generate a concept lattice that illuminates subtle dependencies contained in the data such as: dependencies among landscape indices; index groupings that cluster or differentiate subsets of landscape models; and gradients of complexity within the data set. The concept lattice, if properly drawn, elucidates many of the nuances and implications contained in the data set that are not apparent by inspecting the data only. Producing good diagrams of concept lattices is an art in itself, although some progress in automating this task has been made [154].

Concept analysis is related to *cluster analysis* [46, 88, 89], although it differs in its ability to graphically illustrate subtle properties of the data. A primary distinction between traditional cluster analysis and formal concept analysis is that the former produces a tree of clusters grouped according to similarity criteria [127], while the latter forms a lattice. This not only involves a novel application of reasoning with taxonomies, but permits the detection of subtle relationships as well as general trends in the data. A wide avenue for future research is to pursue the use of formal concept analysis in landscape ecology by studying its utility for the analysis of one or more sets of landscape models.

Bibliography

[1] R. Agrawal, A. Borgida, and H. Jagadish. Efficient management of transitive relationships in large data bases, including is-a hierarchies. In *Proceedings of ACM SIGMOD*, 1989.

[2] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages*, 11(1):115–146, 1989.

[3] H. Aït-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[4] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3/4):195, 1993.

[5] H. Aït-Kaci, A. Podelski, and S. C. Goldstein. Order-sorted feature theory unification. Technical Report 32, Digital Paris Research Lab, Paris, France, May 1993.

[6] J. Allen. *Natural Language Understanding.* Benjamin/Cummings Pub. Co, Redwood City, CA, 2nd edition, 1995.

[7] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[8] T. F. H. Allen and E. P. Wyleto. A hierachical model for the complexity of plant communities. *Journal of Theoretical Biology*, 101:529–540, 1983.

[9] N. Asher. *Reference to Abstract Objects in Discourse*, volume 50 of *Studies in Linguistics and Philosophy.* Kluwer, 1993.

[10] W. L. Baker. A review of models of landscape change. *Landscape Ecology*, 2(2):111–133, 1989.

[11] G. L. Ball and R. Gimblett. Spatial dynamic emergent hierarchies simulation and assessment system. *Ecological Modelling*, 62:107–121, 1992.

[12] B. Banaschewski and G. Bruns. The fundamental duality of partially ordered sets. *Order*, 5:61–74, 1988.

[13] D. B. Barber and H. J. Hamilton. Attribute selection strategies for attribute-oriented generalization. In *Proc. of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 429–441, Toronto, Canada, 1996. Springer-Verlag.

[14] J. M. Baveco and R. Lingeman. An object-oriented tool for individual-oriented simulation: Host-parasitoid system application. *Ecological Modelling*, 61:267–286, 1992.

[15] G. Birkhoff. *Lattice Theory.* Volume 25 of Colloquium Publications. American Mathematical Society, Providence, RI, 3rd edition, 1979.

[16] R. J. Brachman. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16:30–36, 1983.

[17] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of American Association of Artificial Intelligence*, pages 34–37, Austin, TX, 1984.

[18] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[19] P. Bresciani, E. Franconi, and S. Tessaris. Implementing and testing expressive description logics: A preliminary report. In *Proc. First International Symposium on Knowledge Representation, Use and Storage for Efficiency (KRUSE'95)*, Santa Cruz, CA, 1995.

[20] C. Brew. Systemic classification and its efficiency. *Computational Linguistics*, 17(4):375–408, 1991.

[21] A. Bundy, L. Byrd, and C. Mellish. Special purpose, but domain independent, inference mechanisms. In *Proc. European Conference on Artificial Intelligence*, pages 67–74, Orsay, France, 1982.

[22] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*. Springer Verlag, Berlin, 1984.

[23] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, London, England, 1992.

[24] Y. Caseau. Efficient handling of multiple inheritance hierarchies. *ACM SIGPLAN Notices*, 8(28):271, October 1993.

[25] H. Caswell. Community structure: A neutral model analysis. *Ecological Monographs*, 46:327–354, 1976.

[26] M. Chein and M. Mugnier. Specialization: Where do the difficulties occur? In H. Pfeiffer and T. Nagle, editors, *Conceptual Structures: Theory and Implementation. Proc. 7th Annual Workshop*, Las Cruces, NM, 1992. Springer-Verlag.

[27] A. G. Cohn. Many sorted logic = unsorted logic + control? In M. Bramer, editor, *Research and Development in Expert Systems III*, pages 184–194. Cambridge University Press, New York, 1987.

[28] A. G. Cohn. Completing sort hierarchies. *Computers and Mathematics with Applications*, 23(2-9):477–491, 1992. Reprinted in Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergamon Press, Oxford, 1992.

[29] R. J. Cole and P. W. Eklund. Application of formal concept analysis to information retrieval using a hierarchically structured thesaurus. In *Proc. Fourth International Conference on Conceptual Structures (to appear)*, Sydney, Australia, 1996. Springer-Verlag.

[30] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[31] A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In *Proc. CAAAP'94, Lecture Notes in Computer Science, No. 787*, pages 68–84, 1994.

[32] V. Dahl. Translating spanish into logic through logic. *American Journal of Computational Linguistics*, 13:149–164, 1981.

[33] V. Dahl. On database systems development through logic. *ACM Transactions on Database Systems*, 7(1), 1982.

[34] V. Dahl. Incomplete types for logic databases. *Applied Math. Letters*, 4(3):25–28, 1991.

[35] V. Dahl and A. Fall. Logical encoding of conceptual graph type lattices. In *First International Conference on Conceptual Structures*, pages 216–224, Quebec, Canada, 1993. Also available as SFU CSS/LCCR Technical Report 93-3.

[36] V. Dahl, A. Fall, S. Rochefort, and P. Tarau. A hypothetical reasoning framework for natural language processing. In *8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'96)*, Toulouse, France, 1996.

[37] V. Dahl, G. Sidebottom, and J. Ueberla. Expert systems for automatic configuration. *International Journal of Expert Systems*, 6(4):561–579, 1993.

[38] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, England, 1990.

[39] R. Dawkins. Hierarchical organisation: a candidate for ethology. In P. P. G. Bateson and R. A. Hinde, editors, *Growing Points in Ethology*. Cambridge University Press, Cambridge, 1976.

[40] J. Dunning, D. Stewart, B. Danielson, B. Noon, T. Root, R. Lamberson, and E. Stevens. Spatially explicit population models: Current forms and future uses. *Ecological Applications*, 5(1):3–11, 1995.

[41] G. Ellis. Compiled hierarchical retrieval. In T. Nagle, J. Nagle, L. Gerholz, and P. Eklund, editors, *Conceptual Structures: Current Research and Practice*. Ellis Horwood, New York, 1992.

[42] G. Ellis. Efficient retrieval from hierarchies of objects using lattice operations. In *Conceptual Graphs for Knowledge Representation. Proc. First International Conference on Conceptual Structures*, Quebec, Canada, 1993. Springer-Verlag.

[43] G. Ellis. *Managing Complex Objects*. PhD thesis, The University of Queensland, Queensland, Australia, 1995.

[44] G. Ellis and R. Levinson. The birth of peirce: A conceptual graph workbench. In H. Pfeiffer and T. Nagle, editors, *Conceptual Structures: Theory and Implementation. Proceedings of Seventh Annual Workshop*, Las Cruces, New Mexico, 1992. Springer-Verlag.

[45] J. Eusterbrock. Efficient knowledge base reasoning with transitive dags. In *Proc. First International Symposium on Knowledge Representation, Use and Storage for Efficiency (KRUSE'95)*, Santa Cruz, CA, 1995.

[46] B. S. Everitt. *Cluster Analysis*. Halsted Press, New York, 1993.

[47] A. Fall. The foundations of taxonomic encoding. Technical Report 94-20, Simon Fraser University CSS/LCCR, 1994.

[48] A. Fall. An abstract framework for taxonomic encoding. In *Proc. First International Symposium on Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz, CA, 1995.

[49] A. Fall. Heterogeneous encoding. In *Proc. First International Symposium on Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz, CA, 1995.

[50] A. Fall. Spanning tree representations of graphs and orders in conceptual structures. In *Proc. Third International Conference on Conceptual Structures*, pages 232–246, Santa Cruz, CA, 1995. Springer-Verlag.

[51] A. Fall. Sparse logical terms. *Applied Mathematics Letters*, 8(5):11–16, 1995.

[52] A. Fall. Sparse term encoding for dynamic taxonomies. In *Fourth International Conference on Conceptual Structures*, Sydney, Australia, 1996. Springer-Verlag.

[53] A. Fall and V. Dahl. Integrating description identification and systemic classification. Technical Report 93-12, Simon Fraser University CSS/LCCR, 1993.

[54] A. Fall, V. Dahl, and P. Tarau. Resolving co-specification in contexts. In *Proc. Workshop on Context in Natural Language Processing*, Montreal, Canada, 1995.

[55] A. Fall and J. Fall. A hierarchical organization of neutral landscape models. In *Proc. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[56] J. Fall and A. Fall. SELES: A spatially explicit landscape event simulator. In *Proc. GIS/Environmental Modeling Conference*, Santa Fe, New Mexico, 1996. National Center for Geographic Information and Analysis, Santa Barbara. Available on CD and the Internet at: //www.ncgia.ucsb.edu/conf/santa_fe.html.

[57] L. J. Folse, J. M. Packard, and W. E. Grant. AI modelling of animal movements in a heterogeneous habitat. *Ecological Modelling*, 46:57–72, 1989.

[58] R. T. T. Forman and M. Gordon. *Landscape Ecology*. John Wiley and Sons, New York, 1986.

[59] J. S. Fralish. Predicting potential stand composition from site characteristics in the Shawnee Hills forest of Illinois. *The American Midland Naturalist*, 120:79–101, 1988.

[60] T. Gallai. Transitiv orientierbare graphen. In *Acta Math, Tom 18*, pages 25–66. Acad. Sci. Hung, 1967.

[61] D. Ganguly, C. Mohan, and S. Ranka. A space-and-time-efficient coding algorithm for lattice computations. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):819–829, Oct 1994.

[62] B. Ganter and R. Wille. Conceptual scaling. In F. Roberts, editor, *Applications of Combinatorics and Graph Theory to the Biological Sciences*, volume 17, pages 139–167. Springer-Verlag, New York, 1989.

[63] D. Gardiner, B. Tjan, and J. Slagle. Extending conceptual structures: Representation issues and reasoning operations. In T. Nagle, J. Nagle, L. Gerholz, and P. Eklund, editors, *Conceptual Structures: Current Research and Practice*. Ellis Horwood, New York, 1992.

[64] R. H. Gardner. The generation and analysis of neutral models. In *Spatial Analysis Techniques Workshop. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[65] R. H. Gardner. RULE: A program for the generation and analysis of landscape patterns, Unpublished draft report, 1996.

[66] R. H. Gardner, B. T. Milne, M. G. Turner, and R. V. O'Neill. Neutral models for the analysis of broad-scale landscape pattern. *Landscape Ecology*, 1(1):19–28, 1987.

[67] R. H. Gardner and R. V. O'Neill. Pattern, process and predictability: the use of neutral models for landscape analysis. In M. G. Turner and R. H. Gardner, editors, *Quantitative Methods in Landscape Ecology*, Ecological Studies 82, pages 289–307, New York, 1991. Springer-Verlag.

[68] R. H. Gardner, R. V. O'Neill, M. G. Turner, and V. H. Dale. Quantifying scale-dependent effects of animal movement with simple percolation models. *Landscape Ecology*, 3(3/4):217–227, 1989.

[69] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA, 1979.

[70] G. Gazdar and C. Mellish. *Natural Language Processing in Prolog: An Introduction to Computational Linguistics*. Addison-Weslel Publishing Company, Menlo Park, CA, 1989.

[71] G. Gazdar, G. Pullum, R. Carpenter, E. Klein, T. Hukari, and R. Levine. Category structures. *Computational Linguistics*, 14(1), 1988.

[72] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Palo Alto, CA, 1987.

[73] S. M. Glenn and S. L. Collins. Modelling the effects of competition on species percolating through landscapes. In *Proc. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[74] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press Inc., San Diego, CA, 1980.

[75] W. E. Grant and N. R. French. Response of alpine tundra to a changing climate: a hierarchical simulation model. *Ecological Modelling*, 49:205–227, 1990.

[76] M. Habib, M. Huchard, and J. Spinrad. A linear algorithm to decompose inheritance graphs. *Algorithmica (to appear)*, 1995.

[77] M. Habib and L. Nourine. Bit-vector encoding for partially ordered sets. In *Proceedings of ORDAL. Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[78] M. Habib and L. Nourine. Tree structure for distributive lattices and its applications. Technical Report R.R. LIRMM 94036, Université de Montpellier II, Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, 1994.

[79] M. Habib and L. Nourine. Embedding partially ordered sets into product of chains. In *Proc. First International Symposium on Knowledge Representation, Use and Storage for Efficiency (KRUSE'95)*, Santa Cruz, CA, 1995.

[80] M. A. K. Halliday and J. R. Martin, editors. *Readings in Systemic Linguistics.* Batsford Academic and Educational Press, London, 1981.

[81] J. Han and Y. Fu. Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases. In *AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, pages 157–168, Seattle, WA, 1994.

[82] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB'95)*, pages 420–431, Zürich, Switzerland, 1995.

[83] G. M. Henebry. A spatio-temporal neutral model for ecological dynamics. In *Proc. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[84] J. Hobbs. Resolving pronoun references. In *Readings in Natural Language Processing*, pages 339–352. Morgan Kaufmann Publishers, Inc., 1986.

[85] J. F. Horty, R. H. Thomason, and D. S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. *Artificial Intelligence*, 42:311–348, 1990.

[86] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In *International Workshop on Programming Language Implementation and Logic Programming (PLILP90)*, LNCS 456, 1990.

[87] T. Imielinski. Intelligent query answering in rule based systems. *Logic Programming Journal*, 4(1), 1987.

[88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, Englewood Cliffs, N.J., 1988.

[89] R. H. G. Jongman, C. J. F. ter Braak, and O. F. R. van Tongeren. *Data Analysis in Community and Landscape Ecology.* Cambridge University Press, Cambridge, 1995.

[90] D. Kelly. Comparability graphs. In I. Rival, editor, *Graphs and Order.* D. Reidel Publishing Co., Dordrecht, 1985.

[91] S. Kodric, F. Popowich, and C. Vogel. The HPSG-PL system. version 1.2. Technical Report CSS-IS TR 92-05, SFU, 1992.

[92] H. Korth and A. Silberschatz, editors. *Database System Concepts.* McGraw-Hill, New York, 1991.

[93] H. Krieger. Classification and representation of types in TDL. In *Proc. First International Symposium on Knowledge Representation, Use and Storage for Efficiency (KRUSE'95)*, Santa Cruz, CA, 1995.

[94] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23(2-9):573–600, 1992. Reprinted in Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergamon Press, Oxford, 1992.

[95] R. Levinson. Towards domain independent machine intelligence. In *Conceptual Graphs for Knowledge Representation. Proc. First International Conference on Conceptual Structures*, Quebec, Canada, 1993. Springer-Verlag.

[96] P. Massicotte and V. Dahl. Handling concept-type hierarchies through logic programming. In *Proceedings of the Third Annual Workshop on Conceptual Graphs*, St. Paul, MN, 1988.

[97] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier/North-Holland, 1989.

[98] M. C. McCord. Design of a Prolog-based machine translation system. In *Proceedings of the Third International Conference on Logic Programming*. Springer Verlag, 1986.

[99] K. McGarigal and B. Marks. Fragstat: A spatial pattern analysis program for quantifying landscape structure, Unpublished software, Oregon State University, Department of Forest Sciences, Corvelis, Oregon, 1993.

[100] J. E. Meisel and M. G. Turner. Application of semivariogram analysis to simulated and real landscapes. In *Proc. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[101] C. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

[102] C. Mellish. Term-encodable description spaces. In *Logic Programming 1990 Pre-Conference Proceedings*, pages 1–15. Association of Logic Programming, UK Branch, 1990.

[103] C. Mellish. The description identification problem. *Artificial Intelligence*, 52(2):151–167, 1991.

[104] C. Mellish. Graph-encodable description spaces. Technical Report ESPRIT Basic Research Action DYANA Deliverable R3.2.B, University of Edinburgh, Scotland, 1991.

[105] G. V. Merkuryeva and Y. A. Merkuryev. Knowledge based simulation systems - a review. *Simulation*, 62(2):74–89, 1994.

[106] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17, 1978.

[107] G. Mineau. Normalizing conceptual graphs. In T. Nagle, J. Nagle, L. Gerholz, and P. Eklund, editors, *Conceptual Structures: Current Research and Practice*. Ellis Horwood, New York, 1992.

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

[109] R. H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Order*. D. Reidel Publishing Co., Dordrecht, 1985.

[110] R. Muetzelfeldt, D. Robertson, A. Bundy, and M. Uschold. The use of Prolog for improving the rigour and accessibility of ecological modelling. *Ecological Modelling*, 46:9–34, 1989.

[111] M. Mugnier and M. Chein. Polynomial algorithms for projections and matching. In H. Pfeiffer and T. Nagle, editors, *Conceptual Structures: Theory and Implementation. Proceedings of Seventh Annual Workshop*, Las Cruces, New Mexico, 1992. Springer-Verlag.

[112] J. Muller and J. Spinrad. Incremental modular decomposition. *Journal of the ACM*, 19:257–356, 1989.

[113] B. Nebel and H. Burckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. In *Twelfth National Conference on Artifical Intelligence*, Seattle, Washington, 1994.

[114] L. Nourine. *Quelques Propriétés Algorithmiques des Treillis.* PhD thesis, Académie de Montpellier, Université de Montpellier, 1993.

[115] R. V. O'Neill, D. L. DeAngelis, J. B. Waide, and T. F. H. Allen. *A Hierarchical Concept of Ecosystems.* Princeton University Press, Princeton, New Jersey, 1986.

[116] A. P. Pentland. Fractal-based description of natural scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):661–674, 1984.

[117] L. Polidori, J. Chorowicz, and R. Guillande. Description of terrain as a fractal surface, and application to digital elevation model quality assessment. *Photogrammetric Engineering and Remote Sensing*, 57:1329–32, 1991.

[118] C. Pollard and I. Sag. *Information-Based Syntax and Semantics.* CSLI Lecture Notes No. 13. Center for the Study of Language and Information, Stanford University, Stanford, CA, 1987.

[119] F. Popowich and C. Vogel. A logic based implementation of head-driven phrase structure grammar. In *Natural Language Understanding and Logic Programming III*, pages 227–245. Elsevier Science Publishers, Netherlands, 1991.

[120] A. Porto. A framework for deducing useful answers to queries. Technical Report DI/UNL-16/88, Universidade Nova de Lisboa, Lisbon, Portugal, 1988.

[121] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence 5.* Edinburgh University Press, Edinburgh, UK, 1970.

[122] K. H. Riitters, R. V. O'Neill, C. T. Hunsaker, J. D. Wickham, D. H. Yankee, S. P. Timmins, K. B. Jones, and B. L. Jackson. A factor analysis of landscape pattern and structure metrics. *Landscape Ecology*, 10(1):23–39, 1995.

[123] L. Roberts, R. Levinson, and R. Hughey. Issues in parallel hardware for graph retrieval. In *First International Conference on Conceptual Structures, Theory and Applications*, Quebec, Canada, 1993.

[124] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M. Uschold. *Eco-logic: Logic-based Approaches to Ecological Modelling.* MIT Press, Cambridge, Massachusetts, 1991.

[125] J. A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, March 1992.

[126] C. Rogers. Indices of landscape structure, School of Resource and Environmental Management 699 project, Simon Fraser University, 1993.

[127] H. C. Romesburg. *Cluster Analysis for Researchers.* Krieger Publishing, Malabar, Florida, 1984.

[128] B. Russell. Mathematical logic as based on the theory of types. In *Logic and Knowledge.* George Allen and Unwin Ltd., London, 1956.

[129] E. Rykiel. Artificial intelligence and expert systems in ecology and natural resource management. *Ecological Modelling*, 46:3–8, 1989.

[130] H. Saarenmaa, N. D. Stone, L. J. Folse, J. M. Packard, W. E. Grant, M. E. Makela, and R. N. Coulson. An artificial intelligence modelling approach to simulating animal/habitat interactions. *Ecological Modelling*, 44:125–141, 1988.

[131] S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, Stanford University, Stanford, CA, 1986.

[132] S. M. Shieber. *Constraint-Based Grammar Formalisms: Parsing and Type Inference for Natural and Computer Languages.* MIT Press, Cambridge, Mass., 1992.

[133] C. Sidner. Focussing for interpretation of pronouns. *American Journal for Computational Linguistics*, 7(4):217–231, 1981.

[134] N. K. Simpkins and P. Hancox. Chart parsing in Prolog. *New Generation Computing*, 8(2):113–138, 1990.

[135] F. H. Sklar and R. Costanza. The development of dynamic spatial models for landscape ecology: A review and prognosis. In M. G. Turner and R. H. Gardner, editors, *Quantitative Methods in Landscape Ecology*, Ecological Studies 82, pages 239–288, New York, 1991. Springer-Verlag.

[136] J. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, 1984.

[137] D. Stauffer. *An Introduction to Percolation Theory.* Taylor and Francis, London, 1985.

[138] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass., 1994.

[139] G. Stumme. Knowledge acquisition by distributive concept exploration. In *Third International Conference on Conceptual Structures*, pages 98–111, Santa Cruz, CA, 1995.

[140] P. Tarau. BinProlog 3.30 User Guide. Technical Report 95-1, Département d'Informatique, Université de Moncton, February 1995. Available by ftp from *clement.info.umoncton.ca.*

[141] P. Tarau, V. Dahl, and A. Fall. Backtrackable state with linear assumptions, continuations and hidden accumulator grammars. In *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS'95)*, Portland, Oregon, 1995.

[142] P. Tarau, V. Dahl, and A. Fall. Assumption grammars. In *Submitted to International Symposium, on Programming Language Implementation and Logic Programming (PLILP'96)*, 1996.

[143] D. S. Touretzky. *The Mathematics of Inheritance Systems.* Pitman/Morgan Kaufmann, London, 1986.

[144] W. Trotter. *Combinatorics and Partially Ordered Sets.* The Johns Hopkins University Press, Baltimore, 1992.

[145] M. G. Turner. Landscape ecology: the effect of pattern on process. *Annual Review of Ecological Systems*, 20:171–197, 1989.

[146] M. G. Turner, R. Costanza, and F. H. Sklar. Methods to evaluate the performance of spatial simulation models. *Ecological Modeling*, 48:1–18, 1989.

[147] M. G. Turner and V. H. Dale. Modeling landscape disturbance. In M. G. Turner and R. H. Gardner, editors, *Quantitative Methods in Landscape Ecology*, Ecological Studies 82, pages 323–351, New York, 1991. Springer-Verlag.

[148] M. G. Turner, R. H. Gardner, V. H. Dale, and R. V. O'Neill. Predicting the spread of disturbance across heterogeneous landscapes. *OIKOS*, 55:121–129, 1989.

[149] M. G. Turner, W. H. Romme, and R. H. Gardner. Landscape disturbance models and the long-term dynamics of natural-areas. *Natural Areas Journal*, 14(1):3–11, 1994.

[150] M. G. Turner, W. H. Romme, R. H. Gardner, R. V. O'Neill, and T. K. Kratz. A revised concept of landscape equilibrium: Disturbance and stability on scaled landscapes. *Landscape Ecology*, 8(3):213–227, 1993.

[151] C. Vogel, F. Popowich, and N. Cercone. Logic based inheritance reasoning. In *Prospects for Artificial Intelligence.* IOS Press, Burke, VA, 1993.

[152] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, March 1992.

[153] R. Wille. Restructuring lattice theory. In *Ordered Sets.* NATO ASI Series C83, Reidel, Dordecht, Holland, 1982.

[154] R. Wille. Lattices in data analysis: How to draw them with a computer. In *Algorithms and Order*. Reidel, Boston, 1989.

[155] R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23(2-9):493–515, 1992. Reprinted in Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergamon Press, Oxford, 1992.

[156] P. H. Winston. Learning structural descriptions from examples. In *The Psychology of Computer Vision*. McGraw-Hill, New York, NY, 1975.

[157] K. With and A. W. King. Toward the development of a generalized, spatially explicit theory of species' responses to landscape structure. In *Proc. International Association of Landscape Ecology Symposium*, Galveston, Texas, 1996.

[158] W. A. Woods. What's in a link: Foundations for semantic networks. In *Representation and Understanding*. Academic Press, Orlando, Florida, 1975. Reprinted in Readings in Knowledge Representation, R. J. Brachman and H. J. Levesque (Eds.), Morgan Kaufmann, Los Altos, CA, 1985.

[159] W. A. Woods and J. G. Schmolze. The KL-ONE family. *Computers and Mathematics with Applications*, 23(2-5):133–177, 1992. Reprinted in Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergamon Press, Oxford, 1992.

[160] G. Yang, Y. Choi, and J. Oh. CGMA: A novel conceptual graph matching algorithm. In H. Pfeiffer and T. Nagle, editors, *Conceptual Structures: Theory and Implementation. Proceedings of Seventh Annual Workshop*, Las Cruces, New Mexico, 1992. Springer-Verlag.

[161] R. Young, G. Plotkin, and R. Linz. Analysis of an extended concept-learning task. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Cambridge, MA, 1977.