Managing Complex Objects

A thesis submitted by

GERARD ROBERT ELLIS,
B.App.Sc.

in the

Department of Computer Science
University of Queensland

for the degree of

DOCTOR OF PHILOSOPHY

February, 1995
Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original work, except as otherwise acknowledged in the text and that the material has not been submitted either in whole or in part for a degree at this or any other university.

Gerard Ellis
Melbourne

Portions of the material contained herein have appeared in [Ell89b, Ell89a, Ell90a, Ell90b, Ell91, Ell92, LE92, Ell93a, ELR94, EL94a, Ell95].
Contents

1 Introduction ................................................. 21
   1.1 Object classification ..................................... 21
   1.2 Application Domains ..................................... 26
   1.3 The structure of this thesis ............................. 31

2 Indexing Methods for Complex Objects ................. 35
   2.1 Complex graphical objects: a sample domain ............ 38
   2.2 Some Definitions about Partial Orders ................ 43
   2.3 Brute Force Search for Parents and Children .......... 46
   2.4 Two-Level Methods ..................................... 49
   2.5 Hierarchical Methods .................................... 50
   2.6 Finding the Parents of an Object ...................... 51
      2.6.1 Depth-first Search for Parents .................... 53
      2.6.2 Topological Search for Ancestors ................ 55
   2.7 Finding the Children of an Object .................... 59
      2.7.1 Depth-First Search for Descendants ............... 60
      2.7.2 Topological Search for Descendants ............... 60
      2.7.3 Implementing Descendant Search .................. 66
   2.8 Chain Inserting Method ................................ 68
   2.9 Comparison of Techniques .............................. 71
      2.9.1 Two-Level versus Hierarchical Methods ........... 71
      2.9.2 Hierarchical versus Chain Methods ............... 72


<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9.3</td>
<td>Depth-first versus Topological Methods</td>
<td>80</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Complexity of Hierarchies of Objects</td>
<td>82</td>
</tr>
<tr>
<td>2.9.5</td>
<td>Experiments</td>
<td>83</td>
</tr>
<tr>
<td>2.10</td>
<td>Summary</td>
<td>86</td>
</tr>
<tr>
<td>3</td>
<td>Partially Ordered Set Abstract Data Type</td>
<td>87</td>
</tr>
<tr>
<td>3.1</td>
<td>Member: A Hierarchy Primitive</td>
<td>88</td>
</tr>
<tr>
<td>3.2</td>
<td>Insert: A Primitive for Constructing Hierarchies</td>
<td>89</td>
</tr>
<tr>
<td>3.3</td>
<td>Delete: Retracting Objects from Hierarchies</td>
<td>90</td>
</tr>
<tr>
<td>3.4</td>
<td>Subset: Hierarchy Match</td>
<td>92</td>
</tr>
<tr>
<td>3.5</td>
<td>Union: Hierarchy Merge</td>
<td>94</td>
</tr>
<tr>
<td>3.6</td>
<td>Intersection: Hierarchy Unification</td>
<td>97</td>
</tr>
<tr>
<td>3.7</td>
<td>Summary</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>Analysis of Hierarchies for Efficient Search</td>
<td>101</td>
</tr>
<tr>
<td>4.1</td>
<td>Ancestor Search using Intersection of Descendants</td>
<td>102</td>
</tr>
<tr>
<td>4.2</td>
<td>Descendant Search using Unification</td>
<td>113</td>
</tr>
<tr>
<td>4.3</td>
<td>Factored Search</td>
<td>114</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Experiments</td>
<td>122</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary</td>
<td>124</td>
</tr>
<tr>
<td>5</td>
<td>Implementing Order and Lattice Operations</td>
<td>125</td>
</tr>
<tr>
<td>5.1</td>
<td>Applications of Lattice Encoding Techniques</td>
<td>125</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Using Codes for Indexing Very Complex Objects</td>
<td>127</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Using Codes to Implement Type Operations</td>
<td>127</td>
</tr>
<tr>
<td>5.2</td>
<td>Previous Encoding Methods</td>
<td>129</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Leibniz’s Universal Characteristic</td>
<td>135</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Transitive Closure Boolean Matrix</td>
<td>138</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Discussion of Some Poset Characteristics</td>
<td>139</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Compact Encoding</td>
<td>140</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Modulated Encoding</td>
<td>143</td>
</tr>
</tbody>
</table>
5.2.6 More Poset Characteristics ........................................ 146
5.2.7 Compact Hierarchical Encoding ................................. 148
5.2.8 Tree Encoding ...................................................... 151
5.2.9 Vector Time Encoding Causality Spaces ....................... 152
5.3 Supporting Lattice Operations with Subsumption Encodings .... 156
5.4 More Poset Definitions .............................................. 159
5.5 Term Encoding a Lattice ............................................ 163
5.5.1 Implementing Term Code Operations ......................... 167
5.6 Analysis of Term Encoding Efficiency ............................ 168
5.6.1 Results on Real Object Databases ............................. 172
5.7 Summary ............................................................. 176

6 Compiling and Compressing Conceptual Graphs 177

6.1 Introduction ......................................................... 177
6.2 Compressing objects in hierarchies ............................... 179
6.3 A modeling language for complex objects ........................ 182
6.3.1 Relevance of conceptual graphs to broader research ........ 184
6.4 What are Conceptual Graphs? ....................................... 189
6.4.1 Canonical Graphs ................................................. 191
6.4.2 The Relationship between Formation Rules and Subsumption 194
6.5 Storing and Retrieving Conceptual Graphs? ......................... 196
6.5.1 The Generalisation Hierarchy as a Contents Addressable Mem-
ory ................................................................. 197
6.5.2 Searching in the Generalisation Hierarchy ..................... 198
6.5.3 Inserting in the Generalisation Hierarchy ..................... 199
6.6 Compilation in the generalisation hierarchy ...................... 200
6.7 Instructions .......................................................... 204
6.8 Experiments .......................................................... 206
6.9 Summary ............................................................. 209
7 Typing Complex Objects

7.1 Introduction ................................................. 211

7.2 Typing a Graph with its Skeleton Type ......................... 213
  7.2.1 Indexing into the Poset of Skeletons ................. 215
  7.2.2 Coding the Poset of Skeletons .......................... 218
  7.2.3 Complexity of Codes .................................. 219

7.3 Factorisation of Graphs ...................................... 221

7.4 Summary .................................................. 225

8 Conclusions and Future Work .................................. 227

8.1 Conclusions ................................................ 227

8.2 Recommendations ......................................... 229

8.3 Future Work .............................................. 229
List of Figures

1.1 An aerial view of a city .............................................. 22
1.2 Organic Compound Classes ........................................... 27
1.3 Complete hierarchy of fractal images (These images were developed by Culik II and Dube [ID93]) .................................. 28
1.4 Hierarchy of probabilistic finite generators representing images in Figure 1.3 ...................................................... 28
1.5 A hierarchy of conceptual graph program specifications ............ 30
1.6 Example query on a chess database ................................... 31

2.1 A chemical database .................................................... 38
2.2 Chemicals represented as node and edge tuples ......................... 39
2.3 Chemicals represented as data structures ................................ 40
2.4 The chemical database in Figure 2.5 using hashing techniques ...... 41
2.5 A hierarchical view of the chemical database in Figure 2.5 ........... 41
2.6 Combining subcompounds as approximations to chemical compounds 42
2.7 The search space for the object u in a hierarchy (Lehmann) .......... 43
2.8 Partitioning of a hierarchy of graphs for the graph of u ............... 45
2.9 A brute force algorithm for finding the parents of an object .......... 47
2.10 A brute-force algorithm for finding ancestors of an object .......... 47
2.11 A brute force algorithm for finding the children of an object ......... 47
2.12 A brute-force algorithm for finding descendants of an object ........ 48
2.13 An unordered database of chemical compounds ....................... 48
2.14 Two level ordering of chemical compounds .......................... 49
2.15 A two-level algorithm for finding ancestors of an object ........ 50
2.16 A two-level algorithm for finding the descendants of an object .... 50
2.17 A hierarchy representing subcompound relationships in the database
    in Figure 2.13 .......................................................... 51
2.18 Breadth-first versus depth-first versus topological search through a
    hierarchy for subcompounds ................................. 52
2.19 An algorithm for computing the parents of an object in a hierarchy . 53
2.20 An algorithm for finding the ancestors of an object using depth first
    search ................................................................. 54
2.21 An algorithm for finding ancestors using topological search ........ 56
2.22 Topological search through a hierarchy for subcompounds .......... 57
2.23 An algorithm for finding the descendants of an object given the parents 61
2.24 Computing descendants as a subset of intersection of descendants of
    parents .............................................................. 62
2.25 An algorithm for finding descendants using topological search ........ 62
2.26 Topological with intersection search through a hierarchy for super-
    compounds ......................................................... 63
2.27 Computing the intersection of descendants of parents with dynamic
    codes ................................................................. 67
2.28 An algorithm for finding ancestors of an object using binary search
    of chain coverings ............................................. 69
2.29 An algorithm for finding descendants of an object using binary search
    of chain coverings ............................................. 70
2.30 Multiple views of of term lattices ................................ 73
2.31 Comparison of depth-first vs topological on a term lattice $TL_{3,3}$ .... 80
2.32 Next comparison of depth-first vs topological on a term lattice $TL_{3,4}$ 81
2.33 An example of a worst case number of edges in a hierarchy and a
    solution to the problem ......................................... 82
2.34 A conceptual graph of a chess pattern ............................. 83
2.35 Schemas from the Morph chess seed database ....................... 84
LIST OF FIGURES

3.1 An algorithm for testing for object membership in a hierarchy . . . . . 88
3.2 An algorithm for inserting an object in a hierarchy . . . . . . . . . . 89
3.3 An algorithm for inserting a known object in a hierarchy . . . . . . 89
3.4 An example of inserting a chemical compound into a chemical com-
    pound database (hierarchy) . . . . . . . . . . . . . . . . . . . . . . . 90
3.5 An algorithm for deleting an object from a hierarchy . . . . . . . . 90
3.6 An algorithm for deleting a known object from a hierarchy . . . . . . 91
3.7 An algorithm for checking whether there is a path between two objects
    in a hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
3.8 An algorithm for testing whether one hierarchy is a subhierarchy of
    another . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
3.9 Checking for object membership given some known ancestors . . . . 93
3.10 Example hierarchy A . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
3.11 Example hierarchy B . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
3.12 An algorithm for merging two hierarchies . . . . . . . . . . . . . . 96
3.13 Inserting an element given some known ancestors . . . . . . . . . . 96
3.14 Result of merging hierarchies in Figures 3.11 and 3.10 . . . . . . . . 97
3.15 An algorithm for intersecting two hierarchies . . . . . . . . . . . . 98
3.16 Intersection of the hierarchies in Figures 3.11 and 3.10 . . . . . . . . 98

4.1 An encoded aerial view of a city . . . . . . . . . . . . . . . . . . . . . 102
4.2 An encoded aerial view of a city with no descendants other than ⊥
    in common . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
4.3 An encoded aerial view of a city . . . . . . . . . . . . . . . . . . . . . 105
4.4 Searching for ancestors with codes . . . . . . . . . . . . . . . . . . . 107
4.5 An algorithm for finding ancestors using topological search and encoding108
4.6 An algorithm for finding ancestors in a layer . . . . . . . . . . . . . 108
4.7 Topological with intersection search through a hierarchy for subcom-
    pounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
4.8 An encoded aerial view of a city reorganised by partitioning . . . . . 110
4.9 A topological algorithm for finding descendants using unification operations ........................................ 114
4.10 Comparison of path and modulated search for an node in a term lattice $TL_{3,3}$ ......................................................... 115
4.11 An depth-first algorithm for checking an node is a member of a poset 115
4.12 A module can be treated as a single node in a hierarchy .................. 118
4.13 Modulation and factorisation of a hierarchy .......................... 118
4.14 Hierarchy of factors .......................................................... 118
4.15 Cover tree of a modulated term lattice $TL_{3,3}$ ........................ 118
4.16 An algorithm for searching for an node using factored path search 120

5.1 Implementing Type Hierarchies ............................................. 128
5.2 Computation of GLB using an encoding [Mel90] .......................... 130
5.3 Some alternative encodings of a Poset ........................................ 132
5.4 Early Mechanical Implementation of Object Encoding .................. 136
5.5 A compact encoding of the decision tree $DT_{2,3}$ ......................... 142
5.6 A modulated encoding of the decision tree $DT_{2,3}$ ....................... 143
5.7 A compact hierarchical encoding of the decision tree $DT_{2,3}$ ............. 148
5.8 Compact hierarchical encoding of a hierarchy ......................... 148
5.9 Compact hierarchical encoding of the boxes hierarchy in Figure 4.3 . 149
5.10 Modulation of the boxes hierarchy in Figure 4.3 ....................... 150
5.11 Tree encoding of a hierarchy .............................................. 151
5.12 Vector Time and possible inheritance encoding .......................... 153
5.13 Vector time encoding of the hierarchy in Figure 5.8 .................... 155
5.14 A decision graph implementation of detection of bottom ................ 156
5.15 Encoding a clashing and unifying hierarchy ............................ 160
5.16 The set of CoAtomIdeals of a poset ....................................... 163
5.17 Term encoding general problem ............................................. 163
5.18 Coding a term graph .......................................................... 164
5.19 A recursive view of the term encoding in Figure 5.18 .................. 164
LIST OF FIGURES

5.20 Term encoding of the decision tree $DT_{2,3}$ .................................... 166
5.21 Term encoding versus bit-vector encoding a tree ............................. 166
5.22 Translating a term encoding into a Bit-vector encoding .................... 167
5.23 Comparison of term encoding with compact hierarchical encoding on
        a $TL_{28}$ ......................................................................................... 169

6.1 A query on a set of spatial images .................................................. 179
6.2 Hierarchy of a set of spatial images ................................................ 179
6.3 Storing images as the difference between adjacent subimages .......... 180
6.4 Storing images as the difference between adjacent subimages .......... 181
6.5 Hierarchy of compressed probabilistic finite generators in Figure 1.4 . 182
6.6 An axiomatisation of Schubert's steamroller ................................. 187
6.7 Two canonical graphs ................................................................. 192
6.8 Restriction of two graphs in Figure 6.7 ........................................... 192
6.9 Join of the two graphs in Figure 6.8 .............................................. 192
6.10 Simplification of $g$ in Figure 6.9 .................................................. 193
6.11 A sample canon ............................................................................ 193
6.12 A query graph $u$ ........................................................................... 196
6.13 The generalisation hierarchy of the graphs in Figs 6.7, 6.8, 6.9 and
        6.10 .............................................................................................. 197
6.14 A generalisation hierarchy ............................................................ 199
6.15 The generalisation hierarchy in Figure 6.14 after inserting $u$ .......... 200
6.16 Representing graphs as differences from a generalisation [GT87] . . 201
6.17 Encoding conceptual graphs in a generalisation hierarchy with canon-
        ical formation instructions .......................................................... 203
6.18 Some linear derivations of graphs in the hierarchy in Figure 6.13 . . 206
6.19 The generalisation hierarchy Figure6.13 compressed using differences
        from Largest Parent ........................................................................ 207
6.20 The generalisation hierarchy Figure6.13 compressed using differences
        from All Parents ............................................................................ 207
7.1 The names of the graphs in the poset of unlabeled connected graphs with 5 nodes or less, §5 [Leh93] .................. 213
7.2 An encoding of the poset of unlabeled connected graphs § ............... 216
7.3 Example of querying using graph skeleton codes ....................... 217
7.4 Zickwolf's [Zic91] "concept lattice" of graph classes. Each graph represents the class of graphs sharing the attributes appearing above it in the lattice. ................................. 218
7.5 Embedding a poset of connected graphs in a poset of general graphs . 221
7.6 The skeleton product ▵ of a triangle graph .............................. 222
7.7 A Fret ▵ product ........................................ 223

8.1 Complete hierarchy of fractal images Figure 1.3 ......................... 230
8.2 A relationship participation lattice ................................. 232
8.3 A hierarchy of quantified schemas of three variables ........................ 232
List of Tables

2.1 Some example hierarchies and their chains ...................... 72
2.2 Comparison of depth-first, topological, and chain search through chains, decision trees, term lattices, and boolean lattices .................... 73
2.3 Statistics about topology of two databases and their extensions with basis graphs ..................................................... 84
2.4 Statistics of some database constructions ......................... 85
4.1 Collection of box tuples .............................................. 102
4.2 Statistics of classifying same database twice (Parent search only, second time) ...................................................... 123
5.1 The first $n$ primes (up to 11) and the bits need to encode them . . . 138
5.2 Calculating a Clash Partition ........................................ 165
5.3 The characteristics of some example spaces ...................... 169
5.4 Comparison of encoding methods on various spaces in Table 5.3 . . 170
5.5 Topology statistics of some databases ............................. 172
5.6 Further topology statistics of some databases and indications of code sizes in bits ....................................................... 173
6.1 Results of compressing chess files .................................. 208
7.1 Characteristics of some sets of connected graphs including the number of connected graphs up to 10 nodes [Slo73], sequence 649, p.75 . . 219
7.2 Representing a graph by projecting into a complete graph ....... 220
7.3 Lehmann's Skeleton $\mathcal{F}$ multiplication table [Leh93]. Graphs are at the left and type posets are along the top. The entries are posets of type-labeled graphs. The large circles occur in nested posets; these are direct products of the inner and outer posets. (The nested notation has been extended to subdirect representations [Wil89].) ........... 224

8.1 Logic in Hierarchies .................................................. 231


**Acknowledgements**

I thank my supervisor Peter Robinson and John Staples who believed in me enough to support me in my application for the PhD programme. Peter Robinson who had to go through the weekly meetings which helped me develop my ideas from rantings from work on the weekend, to longer rantings and then through the long editing process to get this thesis. I thank my co-supervisor Robert Levinson who had worked on similar ideas for his own thesis and encouraged me to develop those ideas. Bob has the great ability of getting excited by his student’s work which gives the student the confidence essential in the early creative part of a PhD. I thank Bob for the opportunity to study in the second year of my PhD enrollment at The University of California, Santa Cruz. I thank my co-supervisor Peter “Where’s the meat?” Van Roy who gave me an appreciation of detail, as well as a finer appreciation of compiling technologies for logic programming languages.

I thank Hassan Aît-Kaci for giving me the opportunity to work on the LIFE compiler in the Paradise project at Digital’s Paris Research Laboratory, and the confidence that conceptual graphs is on the right track. I thank Peter Creasy for listening to me when I could not find Peter Robinson. I thank Fritz Lehmann for encouraging me, especially about codes. The ideas in chapter 7 were prompted by the suggestions of Fritz Lehmann who is searching for the code to replace all graph comparisons. I thank Andrew Lister for his support and for the wonderful environment he has overseen as head of the Computer Science Department at the University of Queensland. I thank my current employer, RMIT, for their support in the final year of production of this thesis.

Finally, I thank my family especially my wife, Fiona Cameron, for her patience and support through this long process.
Abstract

Todays computers are good at finding things in memory by name or address, but they are relatively poor at finding things by description or structure. Indexing on names and values within computers is well understood, however research into indexing on description, structure, and meaning is in a relatively early stage. This thesis addresses the problem of domain-independent methods for organising and retrieving arbitrarily complex information. Domains of complex objects which can be managed by the methods described in this thesis include chemicals, CAD designs, images, program specifications, conceptual schema, bibliographies, natural language texts, and logical formulas.

Indexing methods such as decision trees, decision graphs, and discrimination networks are designed for terms consisting of sets of attribute-value pairs with some simple (functional) structural relationships between pairs. They are not useful for capturing more complex structural relationships. Our methods are domain and representation independent, instead indexing on some partial order defined over the object domain.

As the basis of our approach we have implemented a partially ordered set (or hierarchy) abstract data type. A database of objects are arranged in a hierarchy based on some partial ordering over the objects. A fundamental object hierarchy operation is classifying an object into the hierarchy. That is, find the closest objects which subsume the object or are subsumed by the object. This is the address of the object in the object hierarchy. An object hierarchy is a contents addressable memory where the objects are the addresses. The number of object comparisons needed to classify an object into the hierarchy is in some sense independent of the object domain. The number of comparisons is affected by the shape of the hierarchy, that is, the ordering over the database of objects.

We give new methods for several orthogonal aspects of storage and retrieval in object hierarchies and improve on the implementation of existing classification techniques. Methods are given for managing object hierarchies such as inserting objects, testing for object membership; and operations on pairs of hierarchies such as matching, merging and generalising. New algorithms are given which prune object
comparisons in some domains. New efficient methods are described for implementing type lattice operations. A method of typing graphic structure is given which can be used as a filter on object comparison. New methods are given for compressing and compiling objects in hierarchies by replacing objects with formation rules applied to subobjects. For each of these methods we illustrate their efficiency on four benchmark orders: chains, decision trees, term lattices and boolean lattices; and on hierarchies of chess graphs in our conceptual graph database implementation Peirce.

We illustrate these methods by using an order-sorted first-order logic: conceptual graphs, but they equally apply to other knowledge representations. The methods described here comprise a comprehensive approach to domain independent complex object storage and retrieval.
The Peirce project: a conceptual graphs workbench

The Peirce [EL92a] project is an open, non-proprietary, international, collaborative effort to build a state-of-the-art, industrial strength, portable, freely available conceptual graph workbench. Peirce is integrating and standardizing the conceptual graph development efforts that are taking place around the world. Peirce will speed introduction of new techniques into the community; facilitate comparison of competing techniques; help researchers cooperate in development; and speed application development. There are already 80 researchers from 12 countries involved in the Peirce project.

Peirce will also be used for knowledge-based systems, graphical programming, computer assisted theorem proving, specification and verification. Peirce will allow developers to model large conceptual graph dictionaries. The Peirce project is divided up into modules including programming standards; database storage and retrieval; linear notation input and output; massively parallel hardware; graph editing and display; conceptual catalogs; conceptual graph programming languages; inference/theorem-proving mechanisms; learning mechanisms; natural language parsers and generators; information systems engineering; and a vision system.

There have been two international workshops on Peirce [EL92b, LE93], and a third is scheduled to be held at the 1994 International Conference on Conceptual Structures at University of Maryland. The basic core database module consists of 22000 lines of C++ code with basic methods for reading, writing, comparing, and storing graphs. On top of this, a graphical user interface (GUI) has been developed [NLE93, ELN94], and a general machine learning module is being developed. An adaptive, pattern-oriented chess playing system [LS91], Morph, will be used to verify and show the high-level support for learning mechanisms and databases of CGs. Much of the technology in Morph is built on top of a generic retrieval, learning and search model known as APS [GL92]. There is a mailing list which discusses the Peirce project. Subscriptions go to peirce-project-request@cs.uq.oz.au, and submissions go to peirce-project@cs.uq.oz.au.
Chapter 1

Introduction

Current database systems lack the representational and computational power to manage complex objects such as chemical formulas, images, program specifications, natural language semantics, problem states, and conceptual schemas. These objects cannot be retrieved by use of keys, but need to be indexed on complex structural or semantical relationships. Sophisticated indexing methods already exist for some structures including single attribute-values, lists of attribute-value pairs, and directed acyclic graphs of attribute-values. These indexing methods are representation or domain dependent and do not handle features such as rotation and symmetry of objects. Our methods are domain and representation independent and are based on an abstract data type for partially ordered sets. For a given domain of objects, a partial order over the objects serves as an index for the domain. We also develop methods for compilation and compression of the conceptual graph knowledge representation, but we expect that the methods will generalise to many other knowledge representations.

1.1 Object classification

Relational databases have been very successful in managing large volumes of simple data. They are mathematically sound and have a conceptually simple basis. However, can they be used to manage large volumes of complex objects with recursive structure or many interrelationships between subobjects? For example, relational
databases generally do not store or exploit shared structure between tuples in the same relation table or across relation tables. This structure itself may be the basis for retrieval in a complex object database.

Stonebraker [Sto89] gave the following example as an application that is badly served by the relational model. The user wants to store the layout of a city. The city is broken down into suburbs, suburbs into blocks, and blocks into buildings. Each of these areas is represented as a box, as in Figure 1.1(a). Each box can be represented by the co-ordinates of the lower left corner \( (x_1, y_1) \) and the upper right corner \( (x_2, y_2) \). Each box is stored in a tuple of the form

\[
\text{box}(id, x_1, y_1, x_2, y_2)
\]

where \( id \) is the box identifier. Following is the box relation for the aerial view of the city in Figure 1.1.

\[
\begin{align*}
\text{box}(a, 0.0, 0.0, 8.5, 9.2) & \quad \text{box}(g, 5.4, 4.2, 6.5, 6.0) & \quad \text{box}(m, 3.8, 1.1, 2.9, 2.5) \\
\text{box}(b, 1.5, 4.2, 3.5, 7.0) & \quad \text{box}(h, 0.3, 0.8, 5.1, 3.2) & \quad \text{box}(n, 2.9, 1.1, 3.7, 2.6) \\
\text{box}(c, 1.8, 5.2, 2.1, 5.9) & \quad \text{box}(i, 0.7, 1.1, 2.4, 2.8) & \quad \text{box}(o, 3.1, 1.2, 3.6, 2.5) \\
\text{box}(d, 2.5, 4.4, 3.2, 6.5) & \quad \text{box}(j, 0.9, 2.3, 1.2, 2.6) & \quad \text{box}(p, 6.0, 0.5, 2.9, 9.0) \\
\text{box}(e, 5.2, 4.0, 7.5, 8.0) & \quad \text{box}(k, 3.4, 2.0, 4.5, 2.8) & \quad \text{box}(q, 6.7, 1.9, 7.4, 2.6) \\
\text{box}(f, 6.5, 6.2, 7.2, 7.3) & \quad \text{box}(l, 3.7, 2.3, 4.0, 2.7) & \quad \text{box}(r, 6.3, 1.0, 8.7, 1.5)
\end{align*}
\]

A typical query may be to find all boxes in a given region, for example, the region defined by the corners \((3, 1)\) and \((5, 3)\). This could be formulated in the SQL relational query language [Cod70] as
select * from box where \( x_1 \geq 3 \) and \( y_1 \geq 1 \) and \( x_2 \leq 5 \) and \( y_2 \leq 3 \)

The common indexing techniques used in relational databases such as B-trees and hash indexes do not provide support for such a query. If there are \( N \) boxes, then \( N \) boxes will be compared using a relational database. This is unreasonably slow for large \( N \). It would be possible to index on the first \( x \) or \( y \) co-ordinate to reduce the number of boxes compared. This would still be slow for large \( N \).

An efficient intuitive solution to the problem exists. Consider the simple aerial view of a city in Figure 1.1. The boxes can be arranged into a hierarchy induced by the inclusion relationship over boxes. Box \( v \) is contained in box \( u \) if

\[
x_{1v} \geq x_{1u} \land y_{1v} \geq y_{1u} \land x_{2v} \leq x_{2u} \land y_{2v} \leq y_{2u}
\]

is true. Box \( u \) contains box \( v \) if and only if there is a path from node \( u \) to node \( v \). In the example, the query is the thick outlined box \( u \). The problem is to find boxes contained in box \( u \).

A two phase search [Lip82, Lev85, Mac88, Lev89, Ell92, Lev92, BHN+92] can be used to find boxes that are contained within box \( u \). The first phase searches from the outer boxes for boxes inside it which contain box \( u \), denoted ancestors of \( u \). The first phase finds the smallest of these boxes which are called the parents of \( u \). The second phase searches within the parent boxes which contain \( u \) for boxes which are contained in \( u \), denoted descendants of \( u \). The second phase returns the largest of these boxes which are called the children of \( u \). In Figure 1.1, the boxes compared in the ancestor search are \( a, b, e, h, p, i, k, m \) and \( n \). The only parent of \( u \) in Figure 1.1 is \( h \). In the descendant search, boxes \( i, j, k, m, n \), and \( o \) are compared. The children of \( u \) are \( k \) (which contains \( l \)), \( m \) and \( o \).

Notice that the hierarchies are only as deep as the nesting of the regions: suburb, block, building. A city could be partitioned into quarters, quarters into subquarters, subquarters into subsubquarters, etc, until each leaf contains exactly one building or smaller object of interest. Such a hierarchy would resemble the balanced trees of traditional indexing methods. A search within such an arrangement of boxes would compare a small number of boxes.

The efficiency of this solution to the boxes problem can be attributed to treating
boxes as whole objects, rather than computing with individual attributes \(x_1, y_1, x_2,\) and \(y_2,\) and then combining through joins of the attributes. A hierarchy captures ordering information over objects, simple or complex.

The box example is an example of the information retrieval problem. The \textit{domain of discourse DoD} is represented by the outer box. A query box \(u\) represents a subset of the DoD. Boxes within \(u\) represent solution subdomains relevant to the query. Many domains exhibit such hierarchical properties. Domains may also have many different hierarchies defined over them.

Efficient retrieval methods exist for simple box objects such as K-D Trees [Omo87], decision trees [Col90] or decision graphs [KS90]. However, the methods described here handle general objects such as graphs or arbitrary logical formulas. Further, compilation and compression techniques as described in chapter 6 produce results comparable with the more specialised methods.

Object-oriented data base systems [EN89] have their roots in the SIMULA programming language developed in the 1960s. In SIMULA, the concept of a \textit{class} groups together the internal data structure of an object in a class declaration. The SMALLTALK language [GRS] was one of the first languages to explicitly incorporate additional OO concepts such as inheritance. Objects consist of attribute value pairs and methods. Objects have a structure similar to the feature structures used in logic programming [AKP93, Car92]. In object-oriented databases and programming languages, a \textit{complex object} is a new composite object defined from previously defined objects in a nested or hierarchical fashion. The indexing methods for object-oriented databases such as H-trees [LOL92] and class-hierarchy indexes [KKD89] are variants of B+ trees [Com79] where the class-hierarchy inheritance structure is used as the control structure over B-trees on attributes in each class.

Methods based on feature terms (nested attribute-value pair sets) can exploit dependencies between subfeature terms. In decision graphs over Prolog terms or feature terms, attribute-value pairs in nested structures are not considered until root attributes (of the nest) match.

In case based reasoning [Kol93], a central operation is case classification which involves finding the "address" of a case in a collection of cases. A case consists of
1.1. *OBJECT CLASSIFICATION*

a set of attribute-value pairs. The indexing methods used in case-based reasoning such as shared-feature networks, discrimination networks [Fei63] and redundant discrimination networks are similar to those used in logic programming such as decision graphs [KS90]. A discrimination network index is based on partitioning cases using some feature. Internal nodes in a discrimination network correspond to some question about the values of an attribute, and children refer to subhierarchies of cases which share a particular value of that attribute. Kołodner’s *CYRUS* [Kol84] was one of the first implementations of Schank’s *Dynamic Memory*. In dynamic memory the organisational structures are called MOPs (Memory Organisation Packets). Each MOP has two functions. It holds general knowledge about the shared features of cases it organises, and it holds an organisation structure that indexes those cases.

Production systems use algorithms based on or similar to the RETE algorithm developed by Forgy [For82]. These methods can handle complex sets of constraints, but are not readily extended to handle rotation or symmetry. RETE implements a many to many match, where the query (working memory) is incrementally refined and incrementally matched against production patterns. RETE allows concurrent queries for supporting production systems. In this thesis we concentrate on the simpler problem of matching an object against many database objects and do not address concurrency. Levinson [Lev94] has developed a Universal Data Structure which generalises RETE.

A major advantage of our approach is that it is not dependent on any relevance factors which are used on matching cases. In essence all features, simple or structural, of a “case” in our system are potentially relevant and may be the basis of retrieval. In indexing techniques such as discrimination networks relevant features to a query may not be compared, since they may be further down the hierarchy from non-matching irrelevant features which would block searching in that subhierarchy. This is because independent features are made dependent on each other in the hierarchy search. Redundant discrimination networks were introduced to overcome this problem, by having multiple hierarchies with different orderings of attributes, so that if relevant attributes which would be blocked by irrelevant attributes would not be so in another hierarchy. The problem with this approach is that there may
be many different possible orderings, hence many different hierarchies to store. Our method does not order incomparable features. Features which are incomparable or independent can be compared in parallel.

Our methods generalise the indexing methods used in object-oriented databases, case based reasoning systems, and Prolog, since we do not restrict our objects to sets of attribute-value pairs, rather we make no assumptions about our complex objects. Our complex objects can contain arbitrary interrelationships between subobjects. The only constraint we place on a collection of objects is that there is some partial ordering over them. Our indexes construct and maintain an index based on such partial orders.

1.2 Application Domains

Here we illustrate some domains where our object classification techniques can be used for retrieval:

**Chemicals:** Figure 1.2 shows a set of chemical compounds which are ordered by the subcompound relationship (subgraph) in a hierarchy. The hierarchy has a number of levels: atoms (Carbon, Hydrogen, etc.); functional groups (small fragments of compounds identifying a group of compounds with similar behaviour such as Hydrocarbon, Amino, and Hydroxy); and compound classes (such as Formaldehyde and Acetone). The hierarchy could be used for answering queries such as finding all the subcompounds and supercompounds of a chemical. Alternatively, close matching compounds which share many of the subcompounds or structural features of a compound could also be found by classification.

**Images:** Image databases are an interesting example of information management. Operations on images such as union, intersection, transposition, inverse, and subset all have counterparts in information management. Figure 1.3 shows a hierarchy of fractal images ordered by subimage. These images are from a system developed by Culik II and Dube [ID93]. The hierarchy contains a Blank image, a Sierpinski Triangle, a Diamond, a Sparkling Diamond, and A
Figure 1.2: Organic Compound Classes

Sierpinski Triangle and Sparkling Diamond. A Diamond is a subimage of a Sparkling Diamond (it occurs recursively in the image). The Sparkling Diamond image is a subimage of the Sierpinski Triangle and Sparkling Diamond image.

Culik II and Dube [ID93] showed how complex images including those with fractal geometries can be defined as regular languages which can be encoded in Probabilistic Finite Generators (PFGs). PFGs are a kind of finite state machine. In this way, PFGs reduce the complexity of the stored objects and the associated matching operations. PFGs can encode infinite resolution images. In Figure 1.4, we show the corresponding hierarchy of PFGs for the hierarchy of images in Figure 1.3. The PFGs are ordered by subgraph modulo probabilities. For example, the PFG for Diamond is a subgraph of the PFG for Sparkling Diamond.

Using these hierarchies it is possible to support queries for finding all the subimages, superimages, and close matching images in a database of images.

**Program Specifications:** The program specifications in Figure 1.5 are ordered into a hierarchy using a generalisation of behaviour relationship. The program specifications are written in the conceptual graph knowledge repre-
Figure 1.3: Complete hierarchy of fractal images (These images were developed by Culik II and Dube [ID93])

Figure 1.4: Hierarchy of probabilistic finite generators representing images in Figure 1.3
s\textsuperscript{entation} [Sow84]. The hierarchy splits the set of programs into programs which relate sequences to sequences (RelationBetweenSequences) and programs which relate sequences and an object to a sequence (RelationBetweenSequencesAndElement). The Insert program prepends an element to a sequence to get a new sequence. Insert’s behaviour is a specialisation of taking a sequence and an element and returning a sequence. Hence Insert $\subseteq$ RelationBetweenSequencesAndElement. Similarly, Append appends an object to the end of a sequence of objects and returns the sequence of objects, and is a specialisation of RelationBetweenSequencesAndElement. A Subsequence program relates a sequence to subsequence of it, hence Subsequence is a RelationBetweenSequences. Permutation relates sequences to permutations of the sequence and is a RelationBetweenSequences. Similarly, Propersubsequence is a kind of Subsequence. Identity is a kind of Subsequence where both the subsequence is exactly the same length as the original. Identity is also a kind of Permutation returning the original. Posort is a kind of permutation. Reverse is a kind of permutation. Sort is a kind of Posort, hence is a kind of Permutation and RelationBetweenSequences.

A hierarchy of program specifications supports finding all the generalisations; specialisations (implementations); and close match program/specification of a program/specification.

\textbf{State in State-Based Search Problems:} Figure 1.6 shows a subhierarchy from the Morph adaptive chess playing system [LS91]. The patterns refer to possible states in a game of chess. The states are ordered by substate (subgraph). Each stored state pattern has an associated weight which is used in machine learning algorithms for selecting moves in a chess game. The current chess state is classified into the hierarchy to find similar states which can be used to select the next move.
Figure 1.5: A hierarchy of conceptual graph program specifications
1.3 The structure of this thesis

The rest of this thesis describes in detail methods which form a comprehensive approach to domain independent complex object storage and retrieval. We also develop methods for compilation and compression of the conceptual graph knowledge representation, and expect that the methods will generalise to many other knowledge representations. The remaining chapters of this thesis are structured as follows.

Chapter 2: We survey methods of classification of complex objects and give some improvements to algorithms. We analyse the complexity of the methods in some benchmark hierarchies and then give the result of some experiments on conceptual graph databases.

Chapter 3: An abstract data type is given for a partially ordered set abstract data type including methods for testing for membership, inserting objects, deleting objects, testing whether one hierarchy subsumes another and merging two hierarchies.
Chapter 4: Descriptions of a new algorithm for finding descendants of a query which checks whether an object is known to intersect with the query. An algorithm is also given which makes use of unification information in descendant search. We detail experiments that show that the intersection method improves on previous methods for certain operations. A member algorithm is given for hierarchies called term lattices which exploits the modular structure of the hierarchy. We prove the modular member algorithm is more efficient than previous methods.

Chapter 5: This chapter examines the problem of encoding hierarchies for supporting an algorithm discussed in chapter 4. Encoding methods are also used for implementing operations on types (fundamental operations in conceptual graphs) and in encoding time in distributed systems. We survey previous methods and discuss new methods which improve on encoding methods. We also give methods for supporting previous methods. We analyse the results of the encoding methods on some benchmark hierarchies and then look at their performance on hierarchies in the Peirce database.

Chapter 6: Conceptual graphs can be represented as a sequence of canonical formation rules. We examine replacing conceptual graphs in hierarchies with their formation rules applied to ancestors. We detail how these formation rules can be considered as instructions for an abstract machine resulting in compilation of conceptual graph databases. In experiments we show a new method of representing conceptual graphs in hierarchies that results in improved compression of conceptual graph databases.

Chapter 7: In the last main chapter we examine representing graphs as types, by encoding them in hierarchies. This is inspired by Lehmann's goal [Leh93] and earlier Leibniz's [(1603)] of doing much of reasoning with simple mechanical operations. We look at representing graph structure for small graphs (size 10 nodes or less) with a single machine word and testing for graph subsumption with a single bit-wise operation.
Chapter 8: In the final chapter we discuss the results in this thesis and discuss future work including the motivating application of this work in a conceptual graph specification and programming language.
Chapter 2

Indexing Methods for Complex Objects

*Use as much of the ordering as possible in indexing a set of objects.*

In total orders, such as strings ordered lexicographically, balanced decision trees are usually used for indexing. In a total order, there is an ordering relationship between every pair of objects. The root of the tree is a *pivot element* which is the middle element of the set. When a string $u$ is inserted into the tree, $u$ is compared to the pivot string. If the string $u$ occurs before the pivot, the left branch of the tree is searched. If $u$ is the pivot, then the pivot is returned. If $u$ occurs after the pivot, the right branch is searched. In the search for $u$ in a balanced tree, a logarithmic number of string comparisons are made in the worst case. The string $u$ is inserted between the largest string less than $u$ and the smallest string greater than $u$.

In a partial order, it is not necessary that there exists an ordering relationship between pairs of objects. Partially ordered sets range from totally ordered sets to completely unordered sets. An example of a partially ordered set is a set of strings ordered by substring. An object $u$ is *classified* by finding its largest subobjects and smallest superobjects in the set and hence determining the relationship between $u$ and all other objects. In practice, the number of comparisons used to classify an object tends to be logarithmic in the number of objects in the database. An object $u$ is inserted between its largest subobjects and smallest superobjects, that is, *classified*. Classification should not be confused with the algorithm of Knuth
[Knu68] which topologically walks a hierarchy to give a linear extension or ordering of the nodes during the walk.

The central operation of a terminological classification system is classification of a term definition into a hierarchy. KL-ONE [BS85] is an early terminological reasoning system based on the design in [Bra77]. Many knowledge representation systems have been built based on the ideas in KL-ONE: KANDOR [PS84], KL-TWO [Vil85], MESON [EO86], YAK [CF90], BACK [Pel91], CLASSIC [PSMB+91], K-Rep [MDW91], KRYPTON [BGL85], KRIS [BH91], LOOM [Mac91], NIKL [SM91], and SB-ONE [Kob91]. There are also a number of conceptual graph implementations: semantic interpreter [SW86], KALIPSOS [FLDC86, Far92], CONGRES [RF87], CGP and SOCD [GT86, GT87, GT88, Tsu88, GTL+92], machine learning algorithms[MGG90, Min90], a Smalltalk tool [Sku88], CONSTRUCT [ENYG89], CGPro [AR89], CP [PH91], CGMA [YCO92], CGP [HK93], and Peirce [EL92a]. SOCD uses a partial implementation of the algorithm developed in [Lev84], and CP uses a depth-first classification algorithm which is shown to be inferior by Levinson [Lev84]. In a terminological reasoning system, a concept hierarchy corresponds to a class hierarchy and a relation hierarchy corresponds to a method hierarchy. In conceptual graphs, encapsulation of methods in a class can be done by constructing class concepts [Sow93] which store method definitions in a nested context.

Woods [Woo91] said of classification systems “More sophisticated algorithms can and should be developed.”. In this chapter, we survey methods for classification and describe primitive operations for search in an abstract data type 1 for a partially ordered set of objects.

An area of research yet to be explored is the issue of organising general quantified statements on the basis of generality [Woo91]. The complexity and decidability of general term subsumption in terminological logics has been studied in detail [LB87, Neb88, Neb90, SS89, PS89, SSS91, DLNN91a, DLNN91b]. Most of the smaller languages experimented with were NP-Complete or undecidable. Most interesting languages with a high degree of expressiveness are semi-decidable or

---

1 Though we use conceptual graphs as the knowledge representation in our Peirce implementation, the algorithms described in this chapter do not rely on conceptual graphs being used.
undecidable. The complexity of terminological knowledge representations and sub-
sumption may seem to be a drawback. However, the efficient management of large complex domains by terminological classification systems is possible. There are two aspects of subsumption which can be independently tackled: term subsumption, and classification search. The first is dependent on the term language or domain, and the second is the hierarchical classification process which uses term subsumption.

Woods [Woo91] criticises previous tractability research in the KL-ONE tradition for focusing on worst-case behaviour, addressing an unrealistic goal of completeness, neglecting the issue of knowledge base size, and focusing on subsumption rather than classification. Woods argues that the primary concern is not the cost of subsumption, but the cost of classification into a large taxonomy. Hence, a current major goal of terminological system developers is to minimise the number of subsumption operations done when classifying an object. Development emphasis is on hierarchical indexing methods which make no assumptions about the structure of the objects in the domain, but only use the subsumption order information in the domain. This means that their classification methods scale with the complexity of the objects being indexed, since their hierarchical indexing methods are independent of object description complexity. Because of this, terminological systems have the potential to significantly outperform systems based on structure matching when used in domains with highly complex structure or no structure.

The Peirce system [EL92a] 2 is a kind of terminological classification system based on conceptual graphs. The Peirce system contains the advanced methods developed by [Lev84, Lev92] which were further refined in Peirce [Ell92]. These classification techniques have been shown to be better than any techniques used in the above terminological systems [BHN+92, BFH+94]. The techniques have recently been adopted by the terminological community, specifically in the KRIS system [BFH+94]. Peirce 3 contains more advanced classification techniques (see [Ell93a]

---

2The Peirce database, developed by Gerard Ellis, is an implementation of a subset of conceptual graphs available by anonymous ftp from ftp.cs.uq.oz.au/pub/peirce/peirce0.1.tar.Z.

3Peirce has additional advantages over other conceptual graph systems: the Peirce software is public domain; Peirce is integrating research and tool development within the conceptual graph community through supporting subgroups within the Peirce project; development of an object-
and chapter 4) which have not yet been adopted.

In Peirce, we are developing techniques based on a partially ordered set abstract data type for classifying first-order formulas. Currently, Peirce can only store and retrieve general conceptual graphs (or nested conceptual graphs) and cannot handle arbitrary formulas. Classification of a formula \( f \) involves finding all formulas in a database which subsume (generalise) \( f \) and all formulas which are subsumed by (specialise) \( f \). The hierarchical method is a sublinear indexing method for first-order formulas. These methods are independent of knowledge representation or application domain. To organise a set of objects, it is only necessary to provide an order oracle such as a subgraph-morphism algorithm for graphs, or a proof procedure for generalisation between two first-order formulas.

These techniques can be used to develop program specification libraries, scientific databases, computer-aided design databases, image libraries, chemical libraries, and ontologies. These kinds of databases have in common large collections of components which are formally defined in first-order logic.

### 2.1 Complex graphical objects: a sample domain

In this chapter, we will use the chemical compound database in Figure 2.1. Here we demonstrate that current technologies such as relational databases and logic programming languages such as Prolog do not support efficient retrieval in such chemicals databases. If these chemical compounds are to be represented in relational databases or Prolog, each chemical compound (graph) might be represented as a oriented graphic logic programming language is under way; and a general machine learning environment is also under development.
2.1. **COMPLEX GRAPHICAL OBJECTS: A SAMPLE DOMAIN**

![Graphical objects](image)

Figure 2.2: Chemicals represented as node and edge tuples

A node tuple contains a (chemical) graph identifier, a label (for example “c” for carbon, “h” for hydrogen, and “o” for oxygen), and node identifier. The edge tuple has as arguments: a graph identifier, a source node identifier, and a sink node identifier. The first column is a cycle of 7 carbon atoms. The second column is a cycle of 6 carbon and one hydrogen atom. The third column represents the same compound as the second column, rotated two atoms clockwise. The fourth column is a string of three carbon atoms.

The advantage of using graphical objects in a computer is metaphorically similar to the way it may be processed by a human. Using a visual form of a chemical it is possible to see the object in a parallel way, each atom being recognised at the same time. But for a human to recognise the compound in tuple form it is necessary for the person to reconstruct the compound by interpreting the tuples.

The values of the identifiers have no informational content other than the topological linking of the graph. Identifiers are like variables, in the sense that any identifier will match any other identifier given the adjacency constraints on each identifier are satisfied. Chemical 1 does not match any of the other three chemicals. Chemical 2 is the same as chemical 3, and does not match any others. Chemical 4 is a subcompound of chemicals 1, 2 and 3. In essence what is required is to add and exploit hierarchical relationships in a relational table. This information could be recorded in subgraph tuples as
Prolog or relational databases do not record the higher order relationships between complex (relational) objects. These sorts of higher order relationships cannot be inferred from such sets of terms in general. Programs must be written for each domain to determine these higher order relationships.

A similar argument holds for data structures. The above chemical compounds can be represented as in Figure 2.3. A possible graph data structure could have nodes and edges stored in arrays. Each edge is represented as a pair of node array indices. Prolog matching would determine that all the chemical compounds in Figure 2.3 are unrelated. This is because languages such as Prolog do not provide primitive support for complex graphical objects that match under rotation and reflection. Prolog should have general graphs as first class objects. Graphs must be part of theory in the form of subsumption and hence matching and indexing operations must be developed.

A meta-program could be written to determine the relationship between the chemicals, and a general meta-program could be written that uses these relationships to sort the complex objects into a hierarchy. However, new meta-comparison functions and new data structures have to be implemented for each new object domain. It would be better to use one general knowledge representation language that could model all the object domains that are encountered and build matching, unification, indexing and compilation techniques around that knowledge representation language. A knowledge representation language should be broken up into sublanguages where specific tools can be built which exploit the properties of the
sublanguage. The sublanguages can then be integrated through a general knowledge representation where every instance string can be classified into one of the sublanguages and operations performed at that level.

A simple indexing method for chemical compounds is to hash on the atom types, and intersect the sets under each atom type. However, this does not help in the organic chemistry domain, where most compounds contain carbon, hydrogen, and oxygen. Figure 2.4 shows the small chemical database in Figure 2.5 under this approach. This hashing approach tends to be as bad as linear sequential search over databases of closely related objects. When dealing with complex objects such as chemical compounds, it is critical to avoid as many comparisons as possible. This is especially important the more complex objects become and as the database grows.

Figure 2.5 shows a hierarchical view of the chemical database example. A path between two chemicals $a$ and $b$ indicates $a$ is a subcompound (subgraph) of $b$.

A typical task is to find all compounds that contain compound $u$. Ancestor search in a hierarchy of chemicals will find all the subcompounds of $u$ in the database. De-
Figure 2.6: Combining subcompounds as approximations to chemical compounds

scendant search will find all of the chemicals which contain \( u \). The virtual address of an object \( u \) is adjacent to the closest subpatterns and the closest superpatterns in the database. Thus even if the object is not in the database, the virtual address can be computed and the object can be inserted at that address if necessary. Close matching chemicals which have subcompounds in common but are not directly related to \( u \) are a short distance away from the virtual address of \( u \) in the hierarchy.

Figure 2.6 shows that close generalisations of objects are better approximations than very general generalisations of objects. The two compounds on the left only lack the information on how to join the two compounds to form the compound in the middle. The two atoms on the right are a poor approximation to the compound in the middle. They don’t describe how many of the atoms are involved or the relationship between the atoms. The set of supercompounds of the compounds on the left would be a proper subset of the set of supercompounds of the atoms on the right, and the intersection of the set of supercompounds for the left pair would be a proper subset of the intersection of the set of supercompounds of the right pair.

Checking to see whether one chemical is a subcompound of another is equivalent to the subgraph isomorphism problem which is NP-complete [GJ79]. It does not matter what language is used to represent the compounds, computing a match is still expensive. We illustrated above that the choice of language does affect the efficiency of indexing and search mechanisms within large collections of complex objects such as chemicals. From the user’s point of view the ease of programming or representation in a language affects the choice. By choosing a language that has graphs as primitives it is easier to represent complex objects such as chemicals.

In the following sections we survey methods of retrieving and classifying objects in hierarchies. We now define some terms used in the discussion of hierarchical classification algorithms.
2.2 Some Definitions about Partial Orders

A partially ordered set or poset $\mathcal{P} = (P, \sqsubseteq)$ is a set of objects $P$ which is partially ordered with respect to a partial order $\sqsubseteq$. An element $u$ subsumes $v$ if $u \sqsubseteq v$, conversely $u$ is subsumed by $v$ if $u \sqsubseteq v$. An element $u$ properly subsumes $v$ (written as $u \preceq v$) if $u \sqsubseteq v$ and $u \neq v$ ($u \not\sqsubseteq v$), conversely $u$ is properly subsumed by $v$ is written as $u \sqsubseteq v$. An element $u$ is said to be comparable to element $v$ if and only if either $u$ or $v$ subsumes the other, $u \sqsubseteq v \lor u \sqsubseteq v$, and are said to be incomparable if neither $u$ nor $v$ subsumes the other, $u \not\sqsubseteq v \land u \not\sqsubseteq v$.

A partial order can be viewed as a directed acyclic graph induced by the partial order, referred to as a Hasse diagram or hierarchy. Each hierarchy contains no transitive links nor loops. If there is a path from node $u$ to node $v$, then $u$ is said to be an ancestor of $v$, conversely $v$ is said to be a descendant of $u$. An element $u$ of $P$ partitions $P - \{u\}$ into three sets: ancestors, descendants, and incomparables; as seen in Figure 2.7. The element top $\top$ subsumes every element in $P$, and every element subsumes the element bottom $\bot$. The nodes in this diagram represent objects (graphs) and should not be confused with atoms from the chemical-graph examples earlier. The Power Set of set $A$ is the set of all subsets of $A$, written $\mathcal{P}A$. The type declaration $S : \mathcal{PO}$ indicates that $S$ is some set of objects of type $O$. The following terms are used in algorithms and proofs that follow. The name
of an algorithm which does element comparisons is distinguished from the name of the set they compute by a leading lower case character. The cardinality or size of set $A$ is denoted by $\#A$.

**Definition 2.2.1** The following terms describe sets in a poset $P = (P, \sqsubseteq)$. We assume a subposet $\overline{S} = (S, \sqsubseteq)$, such that $S \subseteq P$.

- **Ancestors$(u, S)$** = \{ $v : S \mid v \sqsubseteq u$ \}
- **Descendants$(u, S)$** = \{ $v : S \mid u \sqsupset v$ \}
- **Comparables$(u, S)$** = \{ $v : S \mid u \sqsupset v \lor v \sqsupset u$ \}
- **Incomparables$(u, S)$** = \{ $v : S \mid u \nsubseteq v \land v \nsubseteq u$ \}
- **Ancestors$\neq(u, S)$** = \{ $v : S \mid v \sqsupset u$ \}
- **Descendants$\neq(u, S)$** = \{ $v : S \mid u \sqsupset v$ \}
- **Base$(S)$** = \{ $u : S \mid \nexists v : S \bullet u \sqsupset v$ \}
- **Crown$(S)$** = \{ $u : S \mid \nexists v : S \bullet v \sqsupset u$ \}
- **Parents$(u, S)$** = **Base**($\textit{Ancestors}(u, S)$)
- **Children$(u, S)$** = **Crown**($\textit{Descendants}(u, S)$)
- **ParentsFilter**$(L, K, S)$ = \{ $v : L \mid \text{Parents}(v, S) \subseteq K$ \}
- If $F$ is a subset of $S$ where $\perp \in F$ and $\perp \in S$
  
  DescendantsMeet$(L, F, S)$ = \{ $v : L \mid \{ \perp \} \subseteq (\textit{Descendants} \neq(v, S) \cap F)$ \}
- **Chains**$(S)$ = \{ $T : \mathcal{P} S \mid \forall u, v : T \bullet u \sqsupset v \lor v \sqsupset u$ \}.
- **CoChains**$(S)$ = \{ $T : \mathcal{P} S \mid \forall u, v : T \bullet u \nsubseteq v \land v \nsubseteq u$ \}.
- **width**$(S)$ = $\max(\{ \#C \mid C : \text{CoChains}(S) \})$
- **depth**$(S)$ = $\max(\{ \#C - 1 \mid C : \text{Chains}(S) \})$
- **MeetIrreducibles**$(S)$ = \{ $u : S \mid \#\text{Parents}(u, S) = 1$ \}
2.2. SOME DEFINITIONS ABOUT PARTIAL ORDERS

![Diagram of a hierarchy of graphs]

Figure 2.8: Partitioning of a hierarchy of graphs for the graph of $u$

- $\text{JoinIrreducibles}(S) = \{ u : S \mid \#\text{Children}(u, S) = 1 \}$
- $\text{Fringe}(L, S) = \text{Children}(L, S) - L$
- $\text{Twigs}(L, S) = \{ v : \text{Fringe}(L, S) \mid \text{Parents}(v, S) \subseteq L \}$
- $\text{Shoots}(L, S) = \text{MeetIrreducibles}(\text{Twigs}(L, S))$
- $\text{Atoms}(S) = \text{Children}(\top, S)$
- $\text{CoAtoms}(S) = \text{Parents}(\bot, S)$

For example, consider the hierarchy of chemical objects in Figure 2.8. Here the chemicals are ordered in a hierarchy by considering the subcompound relationship. By considering the object $u$ this database is partitioned into ancestors, descendants and incomparables of $u$. In this database $u$ has ancestors $\{ \top, c, f \}$ and descendants $\{ h, i, j, \bot \}$ shown. The only parent of $u$ is $f$, and the children are $\{ h, i, j \}$. The incomparables of $u$ are $\{ a, b, d, e, g \}$. The base of $\{ \top, a, b, c, d, e, f \}$ is $\{ d, e, f \}$, and $\text{Crown}(\{ u, g, h, i, j, \bot \}) = \{ u, g \}$.

A chain is a subset of $S$ in which every two elements are comparable. A chain, $\{ u_1, \ldots, u_n \}$, is totally ordered, $u_1 \sqsubseteq u_2 \sqsubseteq \ldots \sqsubseteq u_n$. Some chains are $\{ \top, a, d, h, \bot \}$, $\{ \top, b, e, j, \bot \}$, and $\{ c, f \}$. A cochain or an anti-chain is a subset of $S$ in which no two elements are comparable. Some cochains are $\{ a, b, c \}$, $\{ d, e, f \}$, $\{ d,$
e, u, g} and \{h, i, j, g\}. The width of a poset \(S\) is the maximum size of cochains in \(S\). For example in Figure 2.8 the width is 4, since there is at least one cochain of width 4 and no larger cochains. The depth of hierarchy \(S\) is measured by the maximum size of chains (minus 1) in \(S\). For example in Figure 2.8 the depth is 5.

The meet-irreducibles are the objects with only one parent object. In Figure 2.8, the meet-irreducibles are \\{a, b, c, d, e, f, u\}. The join-irreducibles are the objects with only one child. In the example, the join-irreducibles are \{c, d, e, g, h, i, j\}. For inheritance hierarchies there are usually many more join-irreducibles compared to the number of meet-irreducibles. Join-irreducibles usually correspond to instances or instance classes. Meet-irreducibles usually correspond to new attributes or methods.

The fringe of the \(\text{ancestors}_\equiv\) of \(u\) (\{T, c, f, u\}) in Figure 2.8 is \{a, b, g, h, i, j\}. The twigs of the \(\text{ancestors}_\equiv\) of \(u\) are \{a, b\}. The shoots of the \(\text{ancestors}_\equiv\) of \(u\) are \{a, b\}. The fringe of the \(\text{ancestors}_\equiv\) of \(j\) (\{T, b, c, e, f, u, j\}) are \{a, g, h, i, \perp\}. The twigs of the \(\text{ancestors}_\equiv\) of \(j\) are \{a, g\}. The shoots of the \(\text{ancestors}_\equiv\) of \(j\) are \{a\}. In Figure 2.8 the atoms are \{a, b, c\}. The coatoms are the leaves of the hierarchy which are parents of \(\perp\). For example, the coatoms are \{h, i, j, g\}.

For many of the sets defined above we overload the functions to take a set as the first argument. For example,

\[
\text{Ancestors}(L, P) = \bigcup_{v \in L} \text{Ancestors}(v, P)
\]

We also take, for example, \(\text{Ancestors}(\emptyset, P) = \emptyset\).

The Peirce system manages any set of complex objects which have a subsumption method defined over them. To classify an object \(u\) into a Peirce hierarchy \((S, \sqsubseteq)\), it is necessary to compute the neighbourhood of \(u\) in \(S\): \(\text{Parents}(u, S)\) and \(\text{Children}(u, S)\). The object \(u\) can then be connected to each parent and child of \(u\) in the hierarchy.

### 2.3 Brute Force Search for \textit{Parents} and \textit{Children}

The \textit{brute force} approach of finding the relationship between a query object \(u\) and the database \(P\) is to compare each object \(v\) of \(P\) to see if a relationship exists with \(u\). In many cases the closest related objects are also required.
2.3. BRUTE FORCE SEARCH FOR PARENTS AND CHILDREN

Figures 2.9 and 2.11 give algorithms to compute the parents and children of an object or the singleton set \{u\} if \(u \in P\). These are the algorithms used in the Crystallographic database [Ae79]. Each query is processed in batch, speed was not a prime concern in their design. Experimental conceptual graphs systems such as CONGRESS [RF87] also use this sequential search technique. These methods are referred to as Design Method I in Levinson's survey of classification methods [Lev92].

```
function parents\_brute\(u : O, P : PO\) returns PO;
begin
    return Base(ancestors\_brute\(u, P\));
end
```

Figure 2.9: A brute force algorithm for finding the parents of an object

```
function ancestors\_brute\(u : O, P : PO\) returns PO;
begin
    K ← ∅;
    for each \(v \in P\) do
        if \(v \sqsupseteq u\) then
            K ← K ∪ \{v\};
    return K;
end
```

Figure 2.10: A brute-force algorithm for finding ancestors of an object

```
function children\_brute\(u : O, P : PO\) returns PO;
begin
    return Crown(descendants\_brute\(u, P\));
end
```

Figure 2.11: A brute force algorithm for finding the children of an object

The ancestors\_brute algorithm sequentially searches the set \(P\) checking each element \(v\) is an ancestor of \(u\). The variable \(K\) accumulates the ancestors of \(u\). The descendants\_brute algorithm similarly sequentially searches \(P\) for descendants of \(u\). Each of these algorithms does exactly |\(P\)| object comparisons. These algorithms are undoubtedly correct, but they are not efficient (unless all the objects are pairwise
function *descendants_brute*(\(u : O, P : PO\)) returns \(PO\);
begin
    \(K \leftarrow \emptyset\);
    for each \(v \in P\) do
        if \(u \sqsupseteq v\) then
            \(K \leftarrow K \cup \{v\}\);
    return \(K\);
end

Figure 2.12: A brute-force algorithm for finding descendants of an object

Figure 2.13: An unordered database of chemical compounds

incomparable). The operation \(\sqsupseteq\) is primitive, since \(\sqsupseteq\) can be determined by \(\sqsupseteq \land \not\sqsubseteq\). If we denote a call to the oracle \(\sqsupseteq\) as a comparison, then to find out the relationship between all of the objects will take \(|P|(|P| - 1)\) comparisons. In complex domains these comparisons are expensive and the goal of a classification method should be to avoid as many comparisons as possible.

Consider the small chemical database in Figure 2.13. The query \(u\) would be compared to each of the chemicals \(a, b, c, d, e, f, g, h\). There is obviously more structure in this example to exploit in the parent and children search. Relationships between known objects in \(P\) should be used when searching for relationships between an unknown object \(u\) and \(P\). That is, the common substructures of the objects should be used to associate them, and computation on these common substructures should be shared.
2.4 Two-Level Methods

As their name suggests these methods have two levels: a set of screens or index objects; and a set of data objects. Screens are commonly occurring substructures in the domain. Figure 2.14 shows the database in Figure 2.13 given chemicals $a$ and $b$ are chosen as screen objects.

Two-level methods have been commonly used in chemical retrieval systems [ACL+73, Baw83, Wil80], and in systems such as CAS, a world-wide repository for chemical descriptions [Sch88, Kas90, ACW+85]. Sowa [Sow84] recommended a two level method for processing conceptual graphs. The screening process involves using relevance factors and preference scores for objects before trying subsumption testing. Rau [Rau88] has implemented such a two-level method of retrieval for conceptual graphs. Figure 2.15 shows such an algorithm for computing the ancestors of an object using a set of screens.

In a two-level method the query object is compared to all of the screens, then the database objects which have all their screens matched are collected as candidates for comparison. All of the candidates are checked to see if they are contained in $u$. In Figure 2.14, the query $u$ is compared to the screens $a$ and $b$, then $c$, $d$, $e$, $f$, $g$ and $h$ are compared since their screens objects are substructures of $u$.

Similarly, Figure 2.16 is an algorithm for computing descendants of an object using screens. First the query object is compared to all of the screens, then the
function ancestors_screen\(u : O, \text{Screens} : \text{PO}, P : \text{PO}\) returns \(\text{PO}\);
begin
AncestorScreens = ancestors_brute\(u, \text{Screens}\);
Candidates ← \(\{v : P \mid \text{Ancestors}(v, \text{Screens}) \subseteq \text{AncestorScreens}\}\);
return ancestors_brute\(u, \text{Candidates}\);
end

Figure 2.15: A two-level algorithm for finding ancestors of an object

database objects whose screens include those matched are collected as candidates for comparison. All of the candidates are then compared to see if they contain \(u\). In Figure 2.14, the query \(u\) is compared to the screens \(a\) and \(b\), then to database objects \(d\), \(g\) and \(h\). The screens and ancestors computed in an ancestor search could be used in this search, if both ancestors and descendants are requested.

function descendants_screen\(u : O, \text{Screens} : \text{PO}, P : \text{PO}\) returns \(\text{PO}\);
begin
AncestorScreens = ancestors_brute\(u, \text{Screens}\);
Candidates ← \(\{v : P \mid \text{Ancestors}(v, \text{Screens}) \supseteq \text{AncestorScreens}\}\);
return descendants_brute\(u, \text{Candidates}\);
end

Figure 2.16: A two-level algorithm for finding the descendants of an object

There is still more information that is not being used. For example, in the ancestor search \(f\) is compared even though the structure \(c\) is contained in \(f\) and \(c\) is not in \(u\). Information gained incrementally in the search is not being communicated to the rest of the search.

2.5 Hierarchical Methods

In the tradition of Levinson [Lev92] our philosophy is to exploit available associativity and structure in the domain. Hierarchical methods are a step in this direction. A hierarchy records all of the non-transitive subsumption relationships between the objects. A path from object \(u\) to \(v\) represents \(u \sqsupset v\). Figure 2.17 shows the hierarchy over the objects in Figure 2.13.
Figure 2.17: A hierarchy representing subcompound relationships in the database in Figure 2.13

Traditionally [Lip82, Lev85, Mac88, Lev89, Ell92, Lev92, BHN+92, BFH+94] the hierarchy search problem has been broken up into two subproblems: given a query, \( u \), find the parents of \( u \), then using the parents find the children of \( u \). If there is an object in the hierarchy equivalent to \( u \), then that object rather than the parents of \( u \) is found in the first phase. Thus the first phase is really \( Parents_u(u, P) \) search rather than parents search.

### 2.6 Finding the Parents of an Object

The parents of \( u \) can be found by searching the generalisation or ancestor space of \( u \). This is usually done by a breadth-first, depth-first or topological (level) search. Levinson [Lev85] empirically demonstrated that topological search compares less objects than depth-first and breadth-first search. The reason for this can be seen in Figure 2.18. Each search can be described in terms of information known about the parents of an object being examined (denoted as \( v \)) as follows.

**Breadth-first:** At least one parent of \( v \) has been compared to \( u \) and is an ancestor of \( u \).
Figure 2.18: Breadth-first versus depth-first versus topological search through a hierarchy for subcompounds

**Depth-first:** Exactly one parent of \( v \) has been compared to \( u \), and is an ancestor of \( u \).

**Topological:** All parents of \( v \) which are ancestors of \( u \) have been compared to \( u \).

In Figure 2.18, the chemical-graphs \( a \) and \( b \) are subgraphs of the query chemical-graph \( u \). The query \( u \) is a subgraph of the \( a \) (\( a \) is a child of \( u \)), but is unrelated to \( c \). In the parent search, both breadth-first and depth-first methods check \( a \) to see whether it is a subgraph of \( u \) even though not all of its parents are subgraphs. The advantage of depth-first search is that if an object \( v \) is found to be an ancestor of the query \( u \), then all of \( v \)'s ancestors can be marked as being ancestors of \( u \) without comparisons. Whereas in a topological search, an object is only compared to \( u \) if all its parents subsume the query \( u \). The chemical graph \( a \) is not compared since \( c \) does not subsume the query \( u \).

Topological search compares exactly all of the ancestors of an object plus the twigs of the ancestor space. In Figure 2.18, the ancestors of \( u \) are \( \{ T, a, b \} \), the fringe of the ancestors of \( u \) are \( \{ c, d \} \) and the twigs of the ancestors of \( u \) are \( \{ c \} \). Depth-first and breadth-first search compare all of the ancestors and fringe objects in the worst case. The fringe space is the failure space of depth-first and topological
2.6. Finding the Parents of an Object

The twig space is the failure space of topological search. Breadth-first search seems to offer no obvious benefits compared to topological search.

Levinson [Lev85] found that the comparisons saved in depth-first search by propagating known ancestor information and hence avoiding some comparisons in the ancestor space did not make up for the extra comparisons avoided by the topological search in the fringe space. The ancestor space is usually a fraction (determined by the branching factor) of the ancestor fringe space. The twigs of the fringe space are usually a small part of the fringe.

2.6.1 Depth-first Search for Parents

Depth-first search is usually done from the top element or from some primitive elements in the database. If a database object \( v \) is an ancestor of \( u \), then the children of \( v \) in the database are searched for closer ancestors, then siblings of \( v \) are compared. Figure 2.19 gives an algorithm for computing parents which calls the depth-first search ancestors algorithm in Figure 2.20. In the \( \text{ancestors}_\text{depth}_\text{first}_\text{=} \) algorithm, the variable \( K \) represents the known elements (computed so far) of \( \text{Ancestors}_\text{=} (u, P) \). This algorithm is similar to that used by Lipkis [Lip82] in a KL-ONE classifier and includes the enhanced communication of search information [Lev85, Mac88]. The variable \( I \) is the set of objects known not to belong to \( \text{Ancestors}_\text{=} (u, P) \), that is, specialisations and incomparables of \( u \).

```
function parents_\text{depth}_\text{first}_\text{=} (u : O, P : \text{PO}) returns boolean;
begin
  return Base (ancestors_\text{depth}_\text{first}_\text{=} (u, P));
end
```

Figure 2.19: An algorithm for computing the parents of an object in a hierarchy

The computation of parents has been separated from the computation of ancestors, to simplify the program a little and to point out that the parents cannot be known without knowing the ancestors of an object. The parents are computed as the base of the ancestors. Traditionally [Ell92, Lev92, BHN+92, BFH+94] deciding which ancestors are the parents is normally done in the ancestor search to avoid
function ancestors\_depth\_first\_\_aux(u : O, v : O, P : PO) returns PO;
begin
  Seen ← ∅;
  K ← ∅;
  I ← ∅;
  ancestors\_depth\_first\_aux(u, v, P);
  return K;
end

procedure ancestors\_depth\_first\_aux(u : O, v : O, P : PO)
begin
  Seen ← Seen ∪ {v};
  if u ∉ I ∧ (v ∈ K ∨ v ⊆ u) then
    begin
      K ← K ∪ {v} ∪ Ancestors(v, P);
      for each w ∈ children(v, P) do
        if w ∉ Seen then
          ancestors\_depth\_first\_aux(u, w, P);
        end
      else
        I ← I ∪ Descendants\_\_aux(v, P);
      end
  end

Figure 2.20: An algorithm for finding the ancestors of an object using depth first search.
traversing the hierarchy a second time, but the costs are similar to a second traversal. Parents are ancestors which have no other ancestors below them. This is tested in depth-first search when no ancestors are produced by searching the children. In topological search the parents are found by each ancestor of \( u \) removing its parents from the base of the ancestors computed so far.

The \( \text{ancestors}_{\text{depth-first}} \) algorithm is similar to that used by Lipkis in the first implementation of classification of concepts in KL-ONE [Lip82], except the assignment to known ancestors was \( K \leftarrow K \cup \{v\} \). MacGregor [Mac88] recognised that if \( v \supseteq u \) then \( \text{Ancestors}(v, P) \) are also ancestors of \( u \). MacGregor also recognised that if \( v \supset u \), then \( \text{Descendants}(v, P) \) are also not ancestors of \( u \) and this information is propagated in the search. Before an object \( v \) is compared to \( u \), it should be checked that the relationship is not already known. That is, before comparing \( v \), \( v \) is checked for being a known non-ancestor (in \( I \)) or a known ancestor (in \( K \)), where \( I \) and \( K \) are global to \( \text{ancestors}_{\text{depth-first}} \text{aux} \). The accumulator \( \text{Seen} \) is also global. The power of hierarchical methods derives from the ability to propagate information through the database search.

### 2.6.2 Topological Search for Ancestors

Levinson [Lev84] pointed out that only if every parent of \( v \) is an ancestor of \( u \), can \( v \) be an \( \text{ancestors}_{\text{eq}} \) of \( u \). Levinson made sure that the parents were compared before the children by doing a “breadth-first” search. In fact what is required is a topological search as pointed out in [Ell92]. Levinson [Lev89, Lev92] later sorted his objects by size before traversal. This ensured the topological properties of his graphs. However, for many domains there may not be a suitable size attribute for the objects. For example, conceptual graphs and similar terminological logics which have labels that can expand and contract, are not necessarily ordered by size. Consider graphs with subtyping on labels. These cannot be sorted by size if they are all the same size graph. There could be millions of graphs having the same graph structure but with different combinations of labels. In [Ell92], the topological order of search is maintained purely based on the ordering of the objects.

An algorithm for finding ancestors using topological search is given in Fig-
function ancestors_topological(u : O, P : PO) returns PO;
begin
    K ← ∅;
    L ← {⊤};
    while L ≠ ∅ do
        begin
            Ancestors ← ancestors_brute env(u, L);
            K ← K ∪ Ancestors;
            L ← Crown(Children(Ancestors, P));
            L ← ParentsFilter(L, K, P);
        end
    return K;
end

Figure 2.21: An algorithm for finding ancestors using topological search

ure 2.21. Children has been overloaded to apply to sets of objects rather than a single object, that is, \( \text{Children}(S, P) = \bigcup_{v \in S} \text{Children}(v, P) \) where \( S \subseteq P \) and is usually a cochain. At each iteration a cochain or layer \( L \) of objects which have all their ancestors matched are compared in the ancestors_brute env call. Known ancestors are accumulated in \( K \). The next layer is the crown of the children of ancestors of \( u \) in the current layer. This new layer is further restricted by only considering objects where all their parents are ancestors of \( u \) which is computed in \( L ← ParentsFilter(L, K, P) \). This trims the layer to ancestors of \( u \) and twigs of the known ancestors of \( u, K \).

In [Ell92] we suggest the use of a priority queue to implement the topological walk rather than the operation \( L ← \text{Crown}(\text{Children}(L, P)) \). We use a priority queue in the Peirce implementation [Ell93b]. Each object in the hierarchy is assigned a number which identifies the level the object is on. The priority queue is implemented as an array of lists, where the level number is the index into the array where the \( i \)th list is a subset of the objects on the \( i \)th level. To insert an object \( v \) on the \( i \)th level in the queue, \( v \) is inserted at the front of the \( i \)th list. To delete the next element in the topological search, the next element from the current layer \( i \) is deleted. If the current layer \( i \) is empty, then the first element of the layer \( i + 1 \) is deleted. In this way elements can be queued and dequeued in constant time.
Consider the search for ancestors (subcompounds) in Figure 2.22. Initially, $L = \{ \top \}$. Since $\top$ is an ancestor of any object, $\top$ is added to $K$. The next layer is $\text{Crown}(\text{Children}(\{ \top \}, P)) = \{ a, b \}$. The parents of $a$ and $b$ ($\{ \top \}$) are known ancestors of $u$ ($\top \in K$), so $L = \{ a, b \}$. In the next iteration, $K = \{ \top, a, b \}$, $L = \text{Crown}(\text{Children}(\{a, b\})) = \text{Crown}(\{c, d, e\}) = \{ c, d, e \}$. The parents of each of $c$, $d$ and $e$ ($\{a\}$, $\{a, b\}$, $\{a, b\}$, respectively) are all known ancestors, thus $L = \{ c, d, e \}$. Since no element of the current layer $L$ is an ancestor of $u$, $\text{Ancestors}_=(u, \{c, d, e\}) = \emptyset$, then the next layer is empty. The program terminates and returns the set of known ancestors $K = \{ \top, a, b \}$.

**Proposition 2.6.1** If $\top, \bot \in P$ and $u \neq \bot$, then the ancestors topological algorithm computes $\text{Ancestors}_=(u, P)$.

**Proof:** We first prove

$$K = \text{Ancestors}_=(u, P) - \text{Descendants}_=(\text{Ancestors}_=(u, L), P)$$

is an invariant, and at termination $K = \text{Ancestors}_=(u, P)$ and $K$ is output.

Finally, we show that ancestors topological terminates.

Initially,

$$K = \emptyset$$
\[ L = \{ \top \} \]

\(\text{Ancestors}_= (u, \{ \top \}) = \{ \top \},\) and \(\text{Descendants}_= (\{ \top \}, P) = P.\) Hence
\(\text{Ancestors}_= (u, P) - P = \emptyset = K\) and the invariant holds initially.

After each iteration
\[ K' = K \cup \text{Ancestors}_= (u, L), \]
\[ L' = \text{Crown}(\text{Children}(\text{Ancestors}_= (u, L), P)),\] and
\[ L'' = \text{Parents}\text{Filter}(L', K', P).\]

We use a standard notation for verification where a program variable is primed to indicate its value after a state update. For example, the assignment \(X \leftarrow X + 1\) is transformed to \(X' = X + 1.\)

\(K' = K \cup \text{Ancestors}_= (u, L)\)
\[= (\text{Ancestors}_= (u, P) - \text{Descendants}_= (\text{Ancestors}_= (u, L), P)) \cup \text{Ancestors}_= (u, L)\]
(from invariant)
\[= \text{Ancestors}_= (u, P) - (\text{Descendants}_= (\text{Ancestors}_= (u, L), P) - \text{Ancestors}_= (u, L))\]
(since \(\text{Ancestors}_= (u, L) \subseteq \text{Ancestors}_= (u, P)\) given \(L \subseteq P\))
\[= \text{Ancestors}_= (u, P) - \text{Descendants}_= (\text{Ancestors}_= (u, L), P)\]
(since \(\text{Descendants}_= (S, P) = \text{Descendants}_= (S, P) - S)\)
\[= \text{Ancestors}_= (u, P) - \text{Descendants}_= (\text{Crown}(\text{Children}(\text{Ancestors}_= (u, L), P)), P)\]
(since \(\text{Descendants}_= (S, P) = \text{Descendants}_= (\text{Crown}(\text{Children}(S, P)), P))\)
\[= \text{Ancestors}_= (u, P) - \text{Descendants}_= (L', P)\]
(contract \(L'\))

We now show
\(\text{Ancestors}_= (u, P) - \text{Descendants}_= (L', P) = \text{Ancestors}_= (u, P) - \text{Descendants}_= (L'', P).\)

Since \(L'' \subseteq L'\) need only show for any \(y \in L'\) if \(y \sqsupseteq u\) then \(y \in L''\). We assume \(y \in L', y \sqsupseteq u,\) and \(y \notin L''\) and obtain a contradiction. Under these assumptions there is some \(x \in \text{Parents}(y, P)\) such that \(x \notin K'.\) But \(x \in \text{Ancestors}_= (u, P),\)
therefore \( x \in \text{Descendants}_\approx(L', P) \), since
\[
K' = \text{Ancestors}_\approx(u, P) - \text{Descendants}_\approx(\text{Ancestors}_\approx(u, L'), P).
\]
Hence there is some \( z \in L' \) such that \( z \sqsupseteq x \). Therefore \( z \sqsupseteq y \) and both \( z \) and \( y \) are in \( L' \) contradicting that \( L' \) is a crown.

Hence \( K' = \text{Ancestors}_\approx(u, P) - \text{Descendants}_\approx(L'', P) \).

Now show \( \text{Ancestors}_\approx(u, P) - \text{Descendants}_\approx(L'', P) = \text{Ancestors}_\approx(u, P) - \text{Descendants}_\approx(\text{Ancestors}_\approx(u, L''), P) \).

Since \( \text{Ancestors}_\approx(u, L'') \subseteq L'' \), then we need only show for any \( y \in L'' \) if \( y \sqsupseteq u \) then \( y \in \text{Ancestors}_\approx(u, L'') \) which is clear.

Hence \( K' = \text{Ancestors}_\approx(u, P) - \text{Descendants}_\approx(\text{Ancestors}_\approx(u, L''), P) \), and so the invariant holds after each iteration.

At termination \( L = \emptyset \). \( \text{Ancestors}_\approx(u, \emptyset) = \emptyset \) and \( \text{Descendants}_\approx(\emptyset, P) = \emptyset \).

Therefore \( K = \text{Ancestors}_\approx(u, P) \), and \( K \) is output.

The algorithm terminates, since after each iteration one more new layer of elements of \( P \) is seen (some of the children of the last layer), the partial order is a directed acyclic graph, so the algorithm must terminate when or before all layers of \( P \) are seen. \( \square \)

### 2.7 Finding the Children of an Object

To insert an object in the proper place in the hierarchy, the objects children must also be determined. The children of an object \( u \) are traditionally found [Lip82, Lev84, Lev92, Ell92] by searching from the parents of \( u \). An alternative would be to apply a search similar to the one described in section 2.6 for parents, but going in the opposite direction starting at the bottom \( \bot \). The search would be for closer descendants (smaller supergraphs), with the query matching database objects, rather than database objects matching the query. However, the databases these methods are designed for are wide shallow hierarchies which fan out from the
top, with most of the objects at the coatoms (attached to $\bot$) ranging from tree-like to term lattice-like hierarchies as in Table 2.2. In hierarchies of these shapes, it is better to continue the search for the children of $u$ from the parents of $u$, using the information already gathered during the parent search.

2.7.1 Depth-First Search for Descendants

Lipkis [Lip82, SL83] gave a depth-first algorithm similar to that of Figure 2.23 which showed that the descendants of $u$ must be under the descendants of the parents of $u$. The search begins from the children of the parents of $u$. If an object $v$ is a descendant of $u$, then all of its descendants are known to be descendants of $u$, otherwise the search continues from the children of $v$ which have not been seen and are not known to be descendants of $u$. The set $K$ is the set of known descendants of $u$, and $\text{Seen}$ is the set of elements of $P$ seen in the search. Objects seen before or known to be descendants are not compared. The $\text{descendants\_depth\_first}$ algorithm can be modified to compute the children, $C$, of $u$. If $v$ has no child $w \in C$, then $v$ is added to $C$ and the parents of $v$ in $C$ are removed from $C$.

2.7.2 Topological Search for Descendants

Levinson [Lev84, Lev92] noted the children of $u$ must be in the intersection of the descendants of the parents of $u$ (see Figure 2.24). We assume that $u$ is not in the poset. If $u$ is in the poset, then finding the descendants is not interesting since they are already known. This property is a generalisation of the screen requirement in the two-level search where children were in the screens of $u$. The children are the first objects encountered in the topological search of the intersection which are matched by the object $u$. Objects outside the intersection are ignored. This is important for databases of complex objects, where the cost of object comparisons is significant. Previous methods [Lip82, SL83] did not use this information, and did comparisons outside the intersection. In section 2.7.3 (also see [Ell92]), ways of computing this intersection efficiently are discussed. The intersection of the descendants of the parents then becomes the focus of the search. An object is only compared when it is in the focus. The depth-first algorithm in Figure 2.23 could be improved by
function descendants(Parents(u, P) : PO, u : O, P : PO) returns PO;
begin
    Seen ← ∅;
    K ← ∅;
    for each v ∈ Parents(u, P) do
        for each w ∈ Children(v, P) do
            if w /∈ Seen ∧ w /∈ K then
                descendants_depth_first(w, u, P);
        end
    end
    return K;
end

procedure descendants_depth_first(w : O, u : O, P : PO)
begin
    Seen ← Seen ∪ {w};
    if u ⊇ w then
        K ← K ∪ {w} ∪ Descendants(w, P);
    else
        for each v ∈ Children(w, P) do
            if v /∈ Seen ∧ v /∈ K then
                descendants_depth_first(v, u, P);
        end
    end
end

Figure 2.23: An algorithm for finding the descendants of an object given the parents
Figure 2.24: Computing descendants as a subset of intersection of descendants of parents

similarly computing the Focus and adding the guard \( v \in \text{Focus} \) before the test \( v \sqsupseteq u \).

---

\[
\begin{align*}
\text{function } & \text{descendants\_topological}(\text{Parents}(u, P) : \mathbb{PO}, u : O, P : \mathbb{PO}) \\
& \text{returns } \mathbb{PO}; \\
\text{begin} \\
& \text{Focus } \leftarrow \bigcap_{v: \text{Parents}(u, P)} \text{Descendants}(v, P); \\
& K \leftarrow \{ \bot \}; \\
& L \leftarrow \text{Crown}(\text{Children}(\text{Parents}(u, P), P)); \\
& \text{while } L \neq \{ \bot \} \text{ do} \\
& \text{begin} \\
& & K \leftarrow K \cup \text{Descendants\_brute}(u, L \cap \text{Focus}, P); \\
& & L \leftarrow \text{Crown}(\text{Children}(L, P)); \\
& & L \leftarrow L - K; \\
& \text{end} \\
& \text{return } K; \\
\text{end}
\end{align*}
\]

Figure 2.25: An algorithm for finding descendants using topological search

The topological search in Figure 2.25 starts from the crown of the children of the parents of \( u \) as the first layer \( L \). This layer is intersected with the focus, then any elements in this intersection are checked to see if they are descendants of \( u \), if so, then all there descendants are added to the known descendants \( K \). The next layer of
Figure 2.26: Topological with intersection search through a hierarchy for supercompounds

objects is the crown of the children of the previous layer minus known descendants. This differs from how Levinson [Lev85, Lev92] searched the hierarchy, but does exactly the same number of comparisons of objects. The difference is discussed in more detail in section 2.7.3.

Consider a search for the descendants of object u in Figure 2.26. Without the intersection focus, the depth-first algorithm given in Figure 2.23 would compare all of the descendants of the parents a and b, that is, objects c, d, e, f, g and h. Using the intersection, d, g and h are compared.

Consider the topological algorithm evaluating the query u in the database in Figure 2.25. The parents of u are {a, b}. Initially, Focus = {d, g, h}, Children({a, b}, P) = {c, d, e, h}, so L = {c, d, e} since e ⊆ h. In the first iteration, the elements of {c, d, e} ∩ {d, g, h} = {d} are compared with u in the descendants brute call. The call descendant_brute(u, S) algorithm computes descendants_{=} (u, S). The element d and its descendants are added to the known descendants of u, K = {d}, since u ⊇ d. The new layer is calculated, that is, L = {f, g, h}. In the next iteration the elements of {f, g, h} ∩ {d, g, h} = {g, h} are compared to u, and since u ⊇ g, g is added to K = {d, g}. The next layer is L = ∅, since there are no children of f, g nor h, and so K remains unchanged. In the next iteration L = ∅, the algorithm
terminates and returns the known descendants \( \{d, g\} \).

**Proposition 2.7.1** If \( \top, \bot \in P \) and \( u \neq \top \) and given \( \text{Parents}(u, P) \), then the descendants topological algorithm computes \( \text{Descendants}_=(u, P) \).

**Proof:** We first show that

\[
K \subseteq \text{Descendants}_=(u, P)
\]

\[
\land K \cup \text{Descendants}_=(L, P) \supseteq \text{Descendants}_=(u, P)
\]

\[
\land \bot \in K
\]

\[
\land \exists S \bullet (S \subseteq P \land K = \text{Descendants}_=(S, P))
\]

is invariant, then show at termination \( K = \text{Descendants}_=(u, P) \) and the algorithm terminates.

Initially,

\[
\text{Focus} = \bigcap_{v: \text{Parents}(u, P)} \text{Descendants}(v, P)
\]

\[
K = \{\bot\}
\]

\[
L = \text{Crown}(\text{Children}(\text{Parents}(u, P), P))
\]

Clearly \( K \subseteq \text{Descendants}_=(u, P) \). Also

\( \text{Descendants}_=(\text{Crown}(\text{Children}(\text{Parents}(u, P), P)), P) \supseteq \text{Descendants}_=(u, P) \),

thus \( K \cup \text{Descendants}_=(L, P) \supseteq \text{Descendants}_=(u, P) \). Clearly, \( \bot \in K \). Since \( \bot \in P \), there is an \( S \), namely \( \{\bot\} \) such that \( K = \text{Descendants}_=(S, P) = \{\bot\} \).

After each iteration

\[
K' = K \cup \text{Descendants}_=(\text{Descendants}_=(u, L \cap \text{Focus}), P)
\]

\[
L' = \text{Crown}(\text{Children}(L, P))
\]

\[
L'' = L' - K'.
\]

We first show \( K' \subseteq \text{Descendants}_=(u, P) \). Since

\( \text{Descendants}_=(u, L \cap \text{Focus}) \subseteq \text{Descendants}_=(u, P) \), then

\( \text{Descendants}_=(\text{Descendants}_=(u, L \cap \text{Focus}), P) \subseteq \)
Descendants\subseteq(\text{Descendants}\subseteq(u, P), P) = \text{Descendants}\subseteq(u, P). \text{ Also, by the invariants, } K \subseteq \text{Descendants}\subseteq(u, P), \text{ and so } K' \subseteq \text{Descendants}\subseteq(u, P) \text{ as required.}

We next show \perp \in K' \text{ which is clear since } \perp \in K \text{ and } K' \subseteq K.

We next show \exists S' \bullet (S' \subseteq P \land K' = \text{Descendants}\subseteq(S', P)). \text{ We assume } \exists S \bullet (S \subseteq P \land K = \text{Descendants}\subseteq(S, P)), \text{ then}

\[
K' = K \cup \text{Descendants}\subseteq(\text{Descendants}\subseteq(u, L \cap \text{Focus}), P) \\
= \text{Descendants}\subseteq(S, P) \cup \text{Descendants}\subseteq(\text{Descendants}\subseteq(u, L \cap \text{Focus}), P) \\
\quad \text{(expand } K) \\
= \text{Descendants}\subseteq(S \cup \text{Descendants}\subseteq(u, L \cap \text{Focus}), P) \\
\quad \text{(since } \text{Descendants}\subseteq(A, P) \cup \text{Descendants}\subseteq(B, P) = \text{Descendants}\subseteq(A \cup B, P)).
\]

The set $S' = S \cup \text{Descendants}\subseteq(u, L \cap \text{Focus})$ has the required properties for the invariant.

We use the following equations in the rest of the proof

\[
\text{Descendants}\subseteq(\text{Crown}(\text{Children}(L, P)), P) = \text{Descendants}(L, P) \quad (2.1)
\]

\[
K' = \text{Descendants}\subseteq(K', P) \quad (2.2)
\]

The second equation follows because $K' = \text{Descendants}\subseteq(S', P)$ and for any set $A$, $\text{Descendants}\subseteq(A, P) = \text{Descendants}\subseteq(\text{Descendants}\subseteq(A, P), P)$

We now show that $K' \cup \text{Descendants}\subseteq(L'', P) \supseteq \text{Descendants}\subseteq(u, P)$.

\[
K' \cup \text{Descendants}\subseteq(L'', P) \\
= K' \cup \text{Descendants}\subseteq(L' - K', P) \quad \text{(expand } L'') \\
= \text{Descendants}\subseteq(K', P) \cup \text{Descendants}\subseteq(L' - K', P) \quad \text{(from eqn 2.2)} \\
= \text{Descendants}\subseteq(K' \cup (L' - K'), P) \quad \text{(distribute } \text{Descendants}\subseteq) \\
= \text{Descendants}\subseteq(K' \cup L', P) \quad (A \cup (B - A) = A \cup B)
\]
\[ \text{Descendants}_= (K', P) \cup \text{Descendants}_= (L', P) \quad \text{(distribute Descendants)} \]
\[ = K' \cup \text{Descendants}_= (L', P) \quad \text{(from eqn 2.2)} \]
\[ = K' \cup \text{Descendants}_= (\text{Crown} (\text{Children}(L, P)), P) \quad \text{(expand L')} \]
\[ = K \cup \text{Descendants}_= (\text{Descendants}_= (u, L \cap \text{Focus}), P) \cup \text{Descendants}_= (L, P) \quad \text{(expand K')} \]

From invariant \( K \cup \text{Descendants}_= (L, P) \supseteq \text{Descendants}_= (u, P) \), and so
\[ K \cup \text{Descendants}_= (L, P) \cup \text{Descendants}_= (\text{Descendants}_= (u, L \cap \text{Focus}), P) \supseteq \text{Descendants}_= (u, P). \]

At termination \( L = \{ \bot \} \). \( \text{Descendants}_= (\{ \bot \}, P) = \{ \bot \} \), thus
\[ K \cup \{ \bot \} \supseteq \text{Descendants}_= (u, P). \] However, \( K \subseteq \text{Descendants}_= (u, P) \) and \( \bot \in K \) from invariant, therefore \( K = \text{Descendants}_= (u, P) \).

The algorithm starts at the parents of \( u \) and on each iteration examines the children of the current layer in \( P \). Since the partial order is a finite directed acyclic graph, the algorithm must terminate. \( \square \)

### 2.7.3 Implementing Descendant Search

Levinson [Lev84, Lev85, Lev92] noted that the intersection of the descendants of the ancestors of \( u \) must contain the descendants of \( u \), and that object comparison outside this intersection is unnecessary. Levinson [Lev92] computed the intersection by walking the descendants of each parent of \( u \), incrementing a count on each object met. On the walk of the descendants of the last parent, each object’s count was tested to see if it was equal to the number of parents, if so the object was in the intersection. Levinson [Lev92] found that though this intersection significantly reduced the number of objects compared, it was still a significant cost itself and was worthy of further refinement.

The following *bit-vector intersection method*, first reported in [Ell92], noted that
for an object \( v \) to be in the intersection of the descendants of the parents of \( u \), \( v \) must have a path from each of the parents of \( u \). A boolean code is assigned to each of the parents of \( u \). The boolean code of the \( i \)th parent consists of all zeroes except the \( i \)th bit is set to 1. The descendants of the parents of \( u \) are walked topologically. When visiting each node \( v \), \( v \)'s code is computed by ORing the codes of \( v \)'s parents. A 1 in the \( i \)th bit in \( v \)'s code represents that there is a path from the \( i \)th parent of \( u \) to \( v \). If every bit in \( v \)'s code is set to 1, then \( v \) is in the intersection of the descendants of the parents of \( u \).

The bit code length is equal to the size of the parent set. For many domains a single word is sufficient to hold the code, since the number of parents of an object is usually small or constant relative to the size of the database. For example, in the chess databases we use in experimentation there are approximately 3 parents on average. Figure 2.27 illustrates how these codes can be computed. Given the parents of \( u \) are \( \{a, b, c\} \) and the descendants of the parents are \( \{d, e, f, i, j, k, l, m\} \), \( \{f, l, m\} \), and \( \{e, g, h, k, l, m, n, o\} \) respectively, then the intersection of the descendants is \( \{l, m\} \). In Figure 2.27 only \( l \) and \( m \) have the code 111 indicating they are in the intersection of the descendants of \( a, b, \) and \( c \).

These codes are similar to the codes assigned to objects in hierarchical encoding methods discussed in chapter 5. The bit-vector intersection method is a significant improvement on the intersection method in [Lev92] and is currently used in the Morph system [LS91] and in the Peirce system [EL92b].
2.8 Chain Inserting Method

Total orders (chains) can be searched in $O(\log n)$ time using traditional binary search techniques [Knu68]. How can these techniques be used to search partial orders? Baader et al. [BHN+92, BFH+94] describe a method which partitions a hierarchy into a set of chains. A chain covering of a poset $P$ is a partition of $P$ into chains, that is, totally ordered subsets. For a given $C_j$ of the covering $P = C_1 \cup \ldots \cup C_m$ binary search is used to find the parent($u, C_j$) and the child($u, C_j$). A possible chain covering of Figure 2.26 is $C_1 \cup C_2 \cup C_3 \cup C_4$, where $C_1 = \{a,c,f\}$, $C_2 = \{b,e,g\}$, and $C_3 = \{d\}$, and $C_4 = \{h\}$. The parents of $u$ in $P$ are a subset of the parents collected from all the chains in the covering. Faigle and Turán [FT88] describe a similar algorithm, but use a test procedure that returns the relationship between two objects: $\sqsubseteq$, $\sqsubset$, or $\not\sqsubseteq \land \not\sqsubset$. This requires two calls to the primitive operation $\sqsubseteq$. Baader et al. [BHN+92, BFH+94] propagate positive and negative information similar to the algorithms in Figures 2.28 and 2.29.

The algorithm in Figure 2.28 finds the ancestors rather than the parents, and communicates the known ancestors between searches using the global variable $K$. Similarly, the known non-ancestors are communicated through $I$. Before an object $v$ is compared to $u$, a check is made that $v$ is not already known to be an ancestor or a non-ancestor. If $u_i$ is an ancestor of $u$, then all of the ancestors of $u_i$ in $P$, which includes those ancestors above $u_i$ in the chain $C$, are ancestors of $u$. The bottom half of $C$ is then searched for closer ancestors. If $u_i$ is not an ancestor of $u$, then all of its descendants in $P$ (not just $C_j$) are also not ancestors of $u$, including the descendants of $u_i$ in $C$. In this case the top half of $C$ is searched for ancestors of $u$.

Similarly, the algorithm in Figure 2.29 searches each chain for descendants of $u$, communicating known descendants $K$ and known non-descendants $I$ to the next chain search. The focus of the intersection of the descendants of the parents of the element $u$ is used to further avoid comparisons with $u$.

Computing a minimal chain covering is non-trivial, and takes more than quadratic time [Jun90]. Baader et al. [BHN+92, BFH+94] give simple heuristics for incremental construction of chain coverings which are near optimal. Given that Parents($u, P$) and Children($u, P$) have been computed, $u$ is inserted in the longest chain satisfying
function ancestors\_chain(u : O, P : \textbf{PO}) \returns \textbf{PO};
begin
\hspace{1em}K \gets \emptyset;
\hspace{1em}I \gets \emptyset;
\hspace{1em}\textbf{for each} \ C_j \in \text{ChainCovering}(P) \ \textbf{do}
\hspace{1.5em}ancestors\_binary\_search(u, C_j, P);
\hspace{1em}return \ K;
\end
end

procedure ancestors\_binary\_search(u : O, C : Chain(O), P : \textbf{PO})
begin
\hspace{1em}high \gets \#C;
\hspace{1em}low \gets 1;
\hspace{1em}i \gets n/2;
\hspace{1em}\textbf{while} \ low < high \ \textbf{do}
\hspace{1.5em}begin
\hspace{2em}i \gets (low + high) \text{DIV} \ 2;
\hspace{2em}\textbf{if} \ u_i \not\in I \ \land \ (u_i \in K \ \lor \ u_i \sqsupseteq u) \ \textbf{then}
\hspace{2.5em}K \gets K \cup \text{Ancestors}_=(u_i, P);
\hspace{2.5em}low \gets i + 1;
\hspace{1.5em}\textbf{else}
\hspace{2em}I \gets I \cup \text{Descendants}_=(u_i, P);
\hspace{2em}high \gets i - 1;
\hspace{1em}\end
end
\end

Figure 2.28: An algorithm for finding ancestors of an object using binary search of chain coverings
function descendants_chain(u : O, Parents(u, P) : PO, P : PO) returns PO;
begin
  Focus ← ∩
           {[v : Parents(u, P)} Descendants(v, P);
  K ← ∅;
  I ← ∅;
  for each C_j ∈ ChainCovering(P) do
    descendants_binary_search(u, C_j, P);
  return K;
end

procedure descendants_binary_search(u : O, C : Chain(O), P : PO)
begin
  high ← #C;
  low ← 1;
  i ← n/2;
  while low < high do
    begin
      i ← (low + high) DIV 2;
      if u_i ∉ I ∧ (u_i ∈ K ∨ (u_i ∈ Focus ∧ u ⊇ u_i)) then
        K ← K ∪ Descendants_(u_i, P);
        high ← i - 1;
      else
        I ← I ∪ Ancestors_(u_i, P);
        low ← i + 1;
      end
  end
end

Figure 2.29: An algorithm for finding descendants of an object using binary search of chain coverings
one of the following conditions:

- Binary search has yielded both a parent and a child of \( u \) in the chain, and they are successive elements of the chain. In this case, insert \( u \) between the parent and child in the chain. This means \( u \) is related to elements of the chain.

- Binary search has yielded a parent (or child) of \( u \) in the chain, and it is the least (resp. greatest) element of the chain. In this case, \( u \) is inserted below (resp. above) this element in the chain. This means \( u \) is related to elements of the chain.

If there is no chain satisfying one of these conditions, a new chain consisting of only \( u \) is created. In experiments [BHN+92, BFH+94], the chain coverings obtained using this method were better than 10% suboptimal.

Consider inserting \( u \) into \( P \) in Figure 2.22. Assume the chains of the hierarchy before inserting \( u \) are \( C_1 \cup C_2 \cup C_3 \cup C_4 \), where \( C_1 = \{a, c, f\} \), \( C_2 = \{b, e, g\} \), and \( C_3 = \{d\} \), and \( C_4 = \{h\} \). Object \( u \) cannot be inserted into \( C_1 \), because it is not comparable with every element. The only chain in which all elements are comparable to \( u \) is \( C_3 \). Thus \( u \) can be inserted into \( C_3 \) to get \( \{u, d\} \).

### 2.9 Comparison of Techniques

In this section we discuss in further detail the various advantages and disadvantages of the methods mentioned earlier: two-level, hierarchical, depth-first, topological and chain methods.

#### 2.9.1 Two-Level versus Hierarchical Methods

Levinson [Lev92] proved that topological hierarchical methods compare less objects than two-level methods by showing that the filter provided by \( \text{parents}_\text{filter}(L, K, P) \) was stronger than the requirement in a two-level search that all the screens that are contained in an object match the query. That is, \( \text{Ancestors}(v, \text{Ancestors}(u, \text{Screens})) \subseteq \text{Ancestors}(v, \text{Ancestors}(u, P)) \) given \( \text{Screens} \subseteq P \). As an example, contrast the
Table 2.1: Some example hierarchies and their chains

<table>
<thead>
<tr>
<th>Example</th>
<th>Chain</th>
<th>Decision Tree</th>
<th>Term Lattice</th>
<th>Boolean Lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Chain 1" /></td>
<td><img src="image2" alt="Decision Tree 1" /></td>
<td><img src="image3" alt="Term Lattice 1" /></td>
<td><img src="image4" alt="Boolean Lattice 1" /></td>
</tr>
<tr>
<td></td>
<td><img src="image5" alt="Chain 2" /></td>
<td><img src="image6" alt="Decision Tree 2" /></td>
<td><img src="image7" alt="Term Lattice 2" /></td>
<td><img src="image8" alt="Boolean Lattice 2" /></td>
</tr>
<tr>
<td></td>
<td><img src="image9" alt="Chain 3" /></td>
<td><img src="image10" alt="Decision Tree 3" /></td>
<td><img src="image11" alt="Term Lattice 3" /></td>
<td><img src="image12" alt="Boolean Lattice 3" /></td>
</tr>
<tr>
<td></td>
<td><img src="image13" alt="Chain 4" /></td>
<td><img src="image14" alt="Decision Tree 4" /></td>
<td><img src="image15" alt="Term Lattice 4" /></td>
<td><img src="image16" alt="Boolean Lattice 4" /></td>
</tr>
</tbody>
</table>

number of objects compared in the search for ancestors in section 2.4 and those compared in topological search for ancestors in the same hierarchy in section 2.6.2.

In topological descendant search an element is only compared to if it is in
\[
\bigcap_{v: \text{Ancestor}(u, P)} \text{Descendants}(v, P),
\]
whereas in a two level search an element is compared if it is in
\[
\bigcap_{v: \text{Ancestor}(u, \text{Screen})} \text{Descendants}(v, P).\]
As an example, contrast the number of objects compared in the search for descendants in section 2.4 and those compared in topological search for descendants in the same hierarchy in section 2.7.2.

### 2.9.2 Hierarchical versus Chain Methods

We expect that, in practice, inheritance hierarchies will be very wide and shallow. Usually the depth will be logarithmic in the width. Since it does not seem possible to analyse general partial orders because of their variability, we analyse some interesting cases in the hierarchy of partially ordered sets. Table 2.1 gives an example of four different benchmarks for inheritance hierarchies: chains, decision trees, term lattices and boolean lattices. We emphasise it is the shape of the hierarchies and their
Figure 2.30: Multiple views of of term lattices

Table 2.2: Comparison of depth-first, topological, and chain search through chains, decision trees, term lattices, and boolean lattices
effect on algorithms we are examining and not specific instances of these orders. In particular we do not consider the complexity of the objects stored in the hierarchies. We use the following properties to discuss hierarchy search algorithms on each of the benchmark kinds of hierarchies.

Nodes: The nodes in a hierarchy correspond to objects. The number of nodes is used as a base reference for formulating other properties as seen in Table 2.2.

Edges: The number of edges in a hierarchy indicate the amount of multiple inheritance in a hierarchy. If the number of edges is close to the number of nodes, then there is little or no multiple inheritance.

Coatoms: The number of coatoms in a hierarchy gives a lower bound on the width of a hierarchy. The width of a hierarchy is the size of a maximal cochain in the hierarchy. The set of coatoms form a cochain, so the width has to be at least as big as the size of the coatom set. The coatoms are all join-irreducibles, since their only child is ⊥.

Ancestors of a coatom: The coatoms occur at the bottom of the hierarchy and hence usually have the most ancestors. We use this attribute as a measure of the objects compared in topological ancestor search.

Fringe of a coatom’s ancestors: The fringe of a coatom’s ancestors is the set of nodes one edge away from the ancestor space of a coatom and is the failure space in hierarchical search methods for a coatom.

Twigs of a coatom’s ancestors: The twigs of a coatom’s ancestors are the elements of the fringe of a coatom’s ancestors which have all parents as ancestors of the coatom.

Shoots of a coatom’s ancestors: The shoots of a coatom’s ancestors are the meet-irreducibles in the fringe space of a coatom’s ancestors.

Meet-irreducibles: The meet-irreducibles (see definition 2.2.1) in a hierarchy correspond to the shoots of all objects in the hierarchy. They are the largest possible failure space in topological ancestor search. The meet-irreducibles include the atoms.
Join-irreducibles: The join-irreducibles (see definition 2.2.1) include the coatoms of the hierarchy and are a measure of the instances or instance classes in the hierarchy.

We now define each of the kinds of hierarchies and prove some properties of each. We assume that the set \( P \) is of size \( n \), and the base of any logarithmic function is 2 unless otherwise specified. We assume for the discussion the hierarchies have uniform branching and depth.

Chain: Chains are defined in definition 2.2.1 and corresponds to one line of inheritance. For every \( u \) and \( v \) in chain \( C \), \( u \) is comparable to \( v \). That is, a chain is a totally ordered set. Chains are the ordered sets operated on by traditional binary sorting algorithms. A chain is a hierarchy of depth \( n \) with branching factor 1. For example in Table 2.1 the chain is of size \( n = 5 \) and hence has depth 5.

Decision Tree: A decision tree is a tree, where each multi-way branch is a decision. A decision consists of an assignment of some value from a set of mutually exclusive values to some variable. A (complete) decision tree has depth \( d \) where every internal node has out-degree \( k \).

Definition 2.9.1 A decision tree \( DT_{k,d} = (Nodes(k,d), Edges(k,d)) \) is recursively defined as follows:

1. \( DT_{k,0} \) is a tree with a single node and no edges; and
2. \( DT_{k,d} \) is constructed by connecting \( k \) nodes to each coatom (leaf node) in \( DT_{k,d-1} \).

Proposition 2.9.2 The decision tree \( DT_{k,d} \) has \( k^d \) coatoms, \( \frac{k^{d+1}-1}{k-1} \) nodes, \( \frac{k^{d+1}-1}{k-1} - 1 \) edges, \( d \) ancestors of each coatom, \( (k - 1)d \) fringe elements of coatom ancestors and \( (k - 1)d \) coatom ancestor shoots.

Proof: From the recursive definition of \( DT_{k,d} \) the recurrence relation for the number of coatoms is \( \#CoAtoms(k,d) = k \cdot \#CoAtoms(k,d - 1) \) and so \( \#CoAtoms(k,d) = k^d \). Further, the recurrence relation for the number
of nodes is \( \#\text{Nodes}(k, d) = \#\text{Nodes}(k, d - 1) + \#\text{CoAtoms}(k, d) \) and so
\( \text{Nodes}(k, d) = \frac{k^{d+1}-1}{k-1} \). Except for the top node of \( DT_{k,d} \) there is an incoming edge for every node, hence \( \#\text{Edges}(k, d) = \frac{k^{d+1}-1}{k-1} - 1 \). Clearly each coatom
has \( d \) ancestors. Each coatom ancestor has \( k \) children including an ancestor of the coatom, hence a coatom has \( (k-1)^d \) fringe elements of its ancestors. Since all are meet-irreducibles there are also \( (k-1)^d \) coatom ancestor twigs and shoots. \( \square \)

For example in Table 2.1 the decision tree \( DT_{2,2} \) of size \( n = 7 \) has depth 3 and
4 chains in the covering with average size 1.75.

**Term Lattice:** A term lattice shape, illustrated in Table 2.1, is the kind found in
Prolog’s Herbrand terms. Unlike hierarchies shaped like decision trees, term
lattice shaped hierarchies allow multiple inheritance.

**Definition 2.9.3** The term lattice \( TL_{k,d} = (\text{Nodes}(k, d), \text{Edges}(k, d)) \) is
recursively defined as follows:

1. \( TL_{k,0} \) is the graph with a single node and no edges; and
2. \( TL_{k,d} \) is constructed by taking \( k \) copies, \( G_0, \ldots, G_{k-1} \), of \( TL_{k,d-1} \) and
   connecting each node of \( G_0 \) to the corresponding node of each of \( G_1, \ldots, G_{k-1} \).

**Proposition 2.9.4** The term lattice \( TL_{k,d} \) has \( (k-1)^d \) coatoms, \( k^d \) nodes,
\( \frac{d(k-1)}{k} k^d \) edges, \( 2^d \) ancestors for each coatom, \( (k-1)^d \) meet-irreducibles, \( (k-1)^d \) join-irreducibles, \( (k-1)^d \) twigs of ancestors of each coatom, and \( (k-1)^d \) shoots of ancestors of each coatom.

**Proof:** From the recursive definition of \( TL_{k,d} \) the recurrence relation for the
number of nodes is \( \#\text{Nodes}(k, d) = k \cdot \#\text{Nodes}(k, d - 1) \) and so
\( \#\text{Nodes}(k, d) = k^d \). Further, the recurrence relation for the number of edges
is \( \#\text{Edges}(k, d) = k \cdot \#\text{Edges}(k, d - 1) + (k-1) \cdot \#\text{Nodes}(k, d - 1) \) and so
\( \#\text{Edges}(k, d) = \frac{d(k-1)}{k} k^d \). Coatoms have no descendants in \( TL_{k,d} \), so the
coatoms of \( TK_{k,d} \) are the coatoms of \( G_1, \ldots, G_{k-1} \), and hence the recurrence
relation for the number of coatoms is
\[ \#CoAtoms(k, d) = (k - 1) \cdot \#CoAtoms(k, d - 1) \]
and so
\[ \#CoAtoms(k, d) = (k - 1)^d. \]

For every ancestor of a coatom in $G_i$ where $i = 1, \ldots, k - 1$, there is an ancestor in $G_0$, hence the recurrence relation for the number of ancestors of a coatom is
\[ \#Ancestors(k, d) = 2 \cdot \#Ancestors(k, d - 1) \]
and so
\[ \#Ancestors(k, d) = 2^d. \]

The top nodes of $G_1, \ldots, G_{k-1}$ have one parent (are meet-irreducibles), the top node of $G_0$, and the rest of the meet-irreducibles are in $G_0$. Hence the recurrence relation for the number of meet-irreducibles is
\[ \#MeetIrreducibles(k, d) = (k - 1) + \#MeetIrreducibles(k, d - 1) \]
and so
\[ \#MeetIrreducibles(k, d) = (k - 1)d. \]

In $TL_{k,d}$ the twigs (and shoots) of ancestors of each coatom are all of the meet-irreducibles in the poset and hence there are $(k - 2)d$ of them. In $TL_{k,d}$ the only join-irreducibles are the coatoms, all the other elements all have at least $k - 1$ children. Hence there are $(k - 1)^d$ join-irreducibles. \( \square \)

The number of coatoms in a term lattice $TL_{k,d}$ is $(k - 1)^d$ and is therefore a lower bound on the width of a hierarchy.

For example in Table 2.1 the term lattice $TL_{3,2}$ which has $n = 9$ nodes, depth 3 and 4 chains in the covering with average size 2.25. Figure 2.30 gives some interpretations of term lattices. Figure 2.30(a) shows that level $i$ in the recursive view of the hierarchy represents possible instantiations of variable $X_i$ with some atom in $1 \ldots k - 1$ with the example $TL_{3,2}$. Figure 2.30(c) shows the branching arrangement for $TL_{k,1}$. Figure 2.30(b) shows the actual terms induced by the variable instantiations in Figure 2.30(a). A reason for drawing term lattices in this recursive fashion is the number of edges is reduced. These diagrams are unambiguous, but leave out most multiple inheritance edges. In a term lattice $TL_{k,d}$ there are $\frac{d(k-1)}{k}k^d$ edges, but in the recursive view there are $k^d - 1$ edges, that is edges are linear in the number of nodes $k^d$.

The recurrence relation for the number of edges in the recursive view of a
term lattice $RTL_{k,d}$ is $\#Edges(k,d) = k.\#Edges(k,d-1) + k - 1$, hence $\#Edges(k,d) = k^d - 1$. There is the added cost of the module container ellipses. The recurrence relation for containers in a term lattice $RTL_{k,d}$ is $\#Containers(k,d) = k.\#Containers(k,d-1)+1$ and $\#Containers(k,1) = 1$, hence $\#Containers(k,d) = k^d$ which is linear in the number of nodes. This method of drawing term lattices gives a diagram with edges and modules linear in the number of nodes. These savings in display of term lattices point to a way of storing term lattices which is linear in the nodes.

**Boolean Lattice:** A boolean lattice is in a one to one relationship with the power set $\mathbb{P}S$ of size $2^{|S|}$. A boolean lattice corresponds to all possible generalisations of one particular concept. We do not believe that these spaces are particularly interesting for real world knowledge bases, rather we believe knowledge bases range in shape from decision trees to term lattices.

**Definition 2.9.5** The boolean lattice $B_d = (\text{Nodes}(d), \text{Edges}(d))$ is recursively defined as follows:

1. $B_0$ is the graph with a single node and no edges; and

2. $B_d$ is constructed by taking 2 copies, $G_0$ and $G_1$, of $B_{d-1}$ and connecting each node of $G_0$ to the corresponding node of $G_1$.

**Proposition 2.9.6** The boolean space $B_d$ has $2^d$ nodes and $\frac{d}{2}.2^d$ edges.

**Proof:** From the recursive definition of $B_d$ the recurrence relation for the number of nodes is $\#\text{Nodes}(d) = 2.\#\text{Nodes}(d-1)$ and so $\#\text{Nodes}(d) = 2^d$. Further, the recurrence relation for the number of edges is $\#\text{Edges}(d) = 2.\#\text{Edges}(d-1) + \#\text{Nodes}(d-1)$ and so $\#\text{Edges}(d) = \frac{d}{2}.2^d$. □

The depth of a boolean lattice is $\log n$ and the average branching is $0.5\log n$. The width of a boolean lattice is $\left(\frac{\log n}{\frac{d^2}{d+1}}\right)$, that is a cochain that runs across the middle of the power set. For example in Table 2.1 the boolean lattice of size $n = 8$, the depth is 3 ($2^3 = 8$) and 3 chains in the covering with average size 2.67.
2.9. COMPARISON OF TECHNIQUES

Table 2.2 summarises the results of the above propositions and adds some further information. We use the notation \( \Omega(f(n)) \) to say the function \( f(n) \) is a lower asymptotic bound on complexity. \( O(g(n)) \) means the function \( g(n) \) is an upper bound and \( \Theta(h(n)) \) means \( h(n) \) has the same asymptotic complexity. The rows ‘Topological Ancestor Search’ and ‘Chain Ancestor Search’ in Table 2.2 give the worst case number of object comparisons for two operations: finding the ancestors of \( u \) by using topological search; and finding the ancestors of an object using chain search. Each operation is described in detail below:

**Topological Ancestor Search:** This search corresponds to that in an *ancestor topological* algorithm. In this search exactly the ancestors and twigs are compared. The worst case for hierarchical methods on boolean lattices is searching for some coatom, since coatoms have the most number of ancestors. In decision trees the ancestors correspond to a path. In term lattices ancestors of coatoms grow slower than the width of the hierarchy. Ancestor search is inefficient on traditional orders like chains, but is optimal for decision trees. Also ancestor search is more efficient than chain search on term lattices, but tends to perform as badly or worse than chain search on boolean lattices.

**Chain Ancestor Search:** This search corresponds to the *ancestors chain* algorithm. The dominant factor for chain search becomes the large number of chains in real databases. The number of chains tend to grow nearly as fast as the size of the database. The chain search is optimal for a single chains, but is inefficient on inheritance structures such as decision trees, term lattices and boolean lattices which all have width exponential in their depth. The number of ancestors of objects in a database tends to grow slower than the number of chains.

In decision trees, topological ancestor search does \( O((k - 1) \log_k n) \) comparisons (twigs), while chain ancestor search does \( O((\log k) \frac{n+1}{k}) \) comparisons. Chain search does exponentially (in the depth) more comparisons than ancestor search on decision trees. In term lattices, topological ancestor search does \( O(2^{\log_k n}) \) comparisons while the dominant term in chain ancestor search comparisons is \((k - 1)(\log_k n)^{-1}\) the difference being significant for \( k > 3 \).
Figure 2.31: Comparison of depth-first vs topological on a term lattice $TL_{3,3}$

This analysis explains why Baader et al. [BHN+92, BFH+94] found that the chain search method was less efficient than topological hierarchical search on real databases. Chain search is good for thin, deep hierarchies, while hierarchical methods are better on shallow, wide hierarchies. The ratio of average number of ancestors and the width of the database is the key factor in deciding on the method to be used.

The final row in Table 2.2 gives the values for a member operation. A member operation checks whether $u$ is an element of $P$. Member requires finding a path to the object, searching from $T$ and selecting a child of $T$ that is closer to $u$, continuing this process until $u$ is found or until no closer ancestor of $u$ can be found. The number of comparisons is equal to average depth $\times$ average branching. Path search is very inefficient in chains, $O(n)$, corresponding to sequential search. Path search is optimal in decision trees, $O(k \log_k n)$, and $O(\log^2 n)$ on boolean lattices. Chapter 4 describes an algorithm that achieves $O(k \log_k n)$ for term lattices.

2.9.3 Depth-first versus Topological Methods

Topological ancestor search compares all of the ancestors of $u$ to find $u$, as well as twigs of the ancestor space. Depth-first search may have to compare all of the ancestors of $u$ as well as all fringe objects in the worst case. In many cases the size of $\text{Fringe}(\text{Ancestors}_e(u, P), P)$ is larger than the size of $\text{Ancestors}_e(u, P)$ due to branching.

To see the growth of ancestors and branching factors we consider two term
2.9. COMPARISON OF TECHNIQUES

Figure 2.32: Next comparison of depth-first vs topological on a term lattice $TL_{3,4}$

lattices. Proposition 2.9.4 states that term lattices $TL_{k,d}$ have $2^d$ nodes in a coatom’s ancestor space and on average $\frac{d(k-1)}{k}$ edges per node. The fringe space of a coatom’s ancestors in a term lattices $TL_{k,d}$ are clearly larger than the ancestor space $2^d$, but less than the branching factor $\frac{(k-1)d}{k}$ times the ancestor space. The twig space of a coatom’s ancestors is $(k-2)d$.

The ancestor space of the coatom $u$ in the term lattice $TL_{3,3}$ in Figure 2.31 is marked by dark-filled circles. Elements of $Fringe(Ancestors_u(u, P), P)$ are marked with dark-filled squares (twigs, all twigs are shoots in a term lattice) and non-filled squares. The hierarchy has depth 3, and $u$ has 8 ancestors, 12 fringe elements including 3 twigs, and a maximal cochain of 12 out of a total space of 27 objects. In a topological ancestor search the dark-filled squares (twigs) and circles (ancestors) are compared. In depth-first ancestor search these may also be compared as well as the non-filled squares (rest of fringe).

Some of the edges in the term lattice $TL_{3,4}$ in Figure 2.32 are omitted for readability. The hierarchy has depth 4, the coatom $u$ has 16 ancestors, 29 fringe elements including 4 twigs (and 4 shoots), and a maximal cochain of 32 out of the 81 objects. This illustrates the growth of the maximal cochain (and hence the chain covering) with respect to the ancestor space, which indicates why ancestor search is more efficient than binary chain search. The fringe of a coatom’s ancestor space is growing rapidly with respect to the coatom’s ancestor space. In these spaces topological ancestor search would outperform depth-first search.
Figure 2.33: An example of a worst case number of edges in a hierarchy and a solution to the problem

### 2.9.4 Complexity of Hierarchies of Objects

In the worst case it may take $O(n^2)$ object comparisons to determine the relationship between all objects in $P$, where $n = |P|$. It takes $n(n-1)$ comparisons to verify that a set of $n$ incomparable objects is a flat hierarchy. The space used by a hierarchy structure could be dominated by the number of edges in the hierarchy, that is, the cumulative size of the parents sets. The parent set sizes in total must be the same as the total size of all children sets. In boolean lattices $B_d$ of $n$ nodes there are $\frac{n \log n}{2}$ edges. In chains of $n$ nodes there are $n - 1$ edges. In decision trees $DT_{k,d}$ of $n$ nodes, there are $n - 1$ edges. In term lattices $TL_{k,d}$ of $n$ nodes, there are $\frac{k-1}{k} n \log_k n$ edges. In the worst case there can be $O(n^2)$ edges. For example the left hierarchy in Figure 2.33 illustrates a worst case example. Usually, in inheritance systems the number of parents of an object is related to the size of the objects themselves (see vector-time spaces in chapter 5 for a counterexample). The hierarchy on the right illustrates how objects can be inserted in the database to decrease the storage requirements.

In practice the number of edges in the databases we have had experience with [LS91] tend to use a fraction of the space of the objects. Of the databases considered in section 2.9.5 the edges in the database “B with basis” used the largest percentage of space relative to the objects themselves with 55 percent, where the database graphs were small. The average number of parents was largest for “A with basis” where each object has an average of 4 parents. In boolean spaces for example there are $\frac{n \log n}{2}$ edges for $n$ nodes, but the nodes are bit-vectors of length $\log n$, hence the edges are linear in the size of the space taken up by the objects ($n$ bit-vectors.
2.9. COMPARISON OF TECHNIQUES

Figure 2.34: A conceptual graph of a chess pattern

take \( n \log n \) space). In practice, we expect many hierarchies have a shape between a decision tree and a term lattice, hence a reasonable estimate of the edges in such hierarchies is \( O(n \log n) \).

2.9.5 Experiments

The following experiments were done on the Peirce database developed by the author [EL92a, Ell93c, Ell93b], and consisting of 26,000 lines of C++ code. The Peirce system is a basic implementation of a conceptual graph [Sow84] database which forms the basis of a freely available, portable \(^4\) conceptual graphs workbench being developed in the Peirce project\(^5\): an international conceptual graphs community collaborative project [EL92a, EL92b, LE93, EL94b].

Surveys of classification methods used in terminological systems [BHN+92, BFH+94] found that the methods developed by Levinson [Lev84, Lev92] and refined in Peirce [Ell92, Ell93a] were the most efficient methods. Woods [Woo91] presented a classification algorithm similar to the ones described earlier and showed that typical-case complexity of the algorithm is logarithmic in the size of the knowledge base. Wood’s algorithm is similar to earlier classification algorithms of Schmolze and Lipkis [SL83] that have since been improved by [Lev84, Lev92, Ell92, Ell93a].

The files that were tested in Tables 2.3 and 2.4 were from the Morph adaptive chess playing system [LS91] which were translated into conceptual graphs by the author. Table 2.3 describes the four databases of chess graphs used. The file “A

\(^4\) The Peirce source code is available by anonymous ftp from ftp.cs.uq.oz.au, /pub/peirce/peirce0.1.tar.Z

\(^5\) The Peirce project was founded by Gerard Ellis and Robert Levinson (University of California at Santa Cruz)
Figure 2.35: Schemas from the Morph chess seed database

Table 2.3: Statistics about topology of two databases and their extensions with basis graphs

<table>
<thead>
<tr>
<th>Database</th>
<th>Nodes</th>
<th>Arcs</th>
<th>Parents</th>
<th>Children</th>
<th>Ancestors</th>
<th>Descendants</th>
<th>Depth</th>
<th>Coatoms</th>
<th>Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A without</td>
<td>1778</td>
<td>13069</td>
<td>3.2</td>
<td>3.7</td>
<td>9.6</td>
<td>9.6</td>
<td>4.0</td>
<td>912</td>
<td>51</td>
</tr>
<tr>
<td>A with basis</td>
<td>1815</td>
<td>16455</td>
<td>4.0</td>
<td>4.5</td>
<td>20.2</td>
<td>20.2</td>
<td>5.0</td>
<td>912</td>
<td>10</td>
</tr>
<tr>
<td>B without</td>
<td>3104</td>
<td>17412</td>
<td>1.85</td>
<td>2.76</td>
<td>2.8</td>
<td>2.8</td>
<td>2.0</td>
<td>2958</td>
<td>144</td>
</tr>
<tr>
<td>B with basis</td>
<td>3146</td>
<td>17824</td>
<td>1.88</td>
<td>2.82</td>
<td>9.5</td>
<td>9.5</td>
<td>3.9</td>
<td>2976</td>
<td>24</td>
</tr>
</tbody>
</table>

without” contains 1778 chess patterns learned by Morph similar to Figure 2.34. Here a solid arc between concepts represents support or attack depending on colour of the piece and a dotted line represents a transitive support or attack operation. The file “B without” is a seed database of 3104 simple patterns of the form shown in the schemas in Figure 2.35, where a Piece could be one of the types WhitePawn, BlackPawn, ..., WhiteQueen, BlackQueen. The files “A with basis” and “B with basis” include 42 basis graphs in addition to the graphs in files “A without” and “B without”, respectively. These basis graphs are the top layers seen in Figure 1.6. These basis graphs are the graphs from which all other graphs are derived. In the chess databases they are single concepts (e.g. [WhiteBishop], the white bishop) and some feature skeletons (e.g. [Piece(Rank=>T)], a piece on some rank (row)). Adding the basis graphs increases the edges (edges between an object and its parents) by 25.9 % in the case of database A and 2.4 % for database B.

Figure 1.6 shows a query on database “A with basis” that is similar to that used in construction of a hierarchy, except in this case the query object is already present. The chess graphs shown are the graphs compared to the query (circled) during topological search. Notice that the search quickly converges on the query
Table 2.4: Statistics of some database constructions

<table>
<thead>
<tr>
<th>Database</th>
<th>Visited</th>
<th></th>
<th></th>
<th>Compared</th>
<th></th>
<th>% of Brute Force</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parents</td>
<td>Children</td>
<td></td>
<td>Parents</td>
<td>Children</td>
<td>Total</td>
<td>Parents</td>
</tr>
<tr>
<td>A without</td>
<td>191.8</td>
<td>441.1</td>
<td>135.0</td>
<td>623.7</td>
<td>104.9</td>
<td>31.2</td>
<td>243906</td>
</tr>
<tr>
<td>A with basis</td>
<td>332.5</td>
<td>900.2</td>
<td>267.9</td>
<td>1753.4</td>
<td>44.3</td>
<td>3.1</td>
<td>86990</td>
</tr>
<tr>
<td>B without</td>
<td>174.6</td>
<td>367.4</td>
<td>61.9</td>
<td>154.1</td>
<td>141.8</td>
<td>7.3</td>
<td>462596</td>
</tr>
<tr>
<td>B with basis</td>
<td>127.1</td>
<td>318.7</td>
<td>57.7</td>
<td>143.1</td>
<td>41.2</td>
<td>2.5</td>
<td>137329</td>
</tr>
</tbody>
</table>

graph. At the moment no hashing of types and relation signatures is done. Hashing would reduce the comparisons by a further 12 graphs in this query. The graphs that were compared are relatively simple compared to the graphs in the coatoms of the hierarchy. Only one of the 913 more complex coatom objects was compared and only 51 objects were compared out of a possible 1815. This database is typical of the hierarchies that the hierarchical classification methods were designed for: shallow and extremely wide. Typically the depth of such a hierarchy is logarithmic with respect to the width.

Table 2.4 details the computation in constructing the hierarchies for the databases. Constructing a hierarchy consists of a series of object insertion operations. An insertion operation, which is described in in chapter 3, involves classifying the object to be inserted: finding the parents and the children of the object. The execution times were recorded on a SparcStation II. In the topological parent search, the “all parents must match before comparing” rule prunes many of the objects that were visited (ancestor and fringe nodes). In “A with basis”, an average of 44.3 objects were compared (ancestor and twig nodes) of the average of 332.5 objects visited (a reduction of 86.7 percent). In children search, the “must be in the intersection of the Descendants of the Parents before comparing” rule prunes nearly all of the objects visited in A with basis, an average of 267.9 reduced to 3.1 (98.8 percent).

Table 2.4 indicates that the major cost when classifying an object is in the parent search. The children search is relatively inexpensive in terms of objects compared, especially in databases that contain basis graphs. However, the cost of visiting objects while still not comparing them is significant. The edges visited is always at least double the number of objects visited, since this includes looking back at the
parents. The other edges account for objects which are incomparable parents (to u) of objects (which are two edges away from the ancestor space of u). As objects and their matching become more complex, the number of objects compared dominates the overall construction time.

The topological method did 7.6 percent of the computation that brute force \( n(n - 1) \) would do for database A, and 4.8 percent for B. Adding the basis graphs reduced these comparisons by 64 percent and 70 percent respectively. This can be attributed to the strengthening of the “all parents must be matched before comparing” rule, since there are more parents to test. The time taken to construct “A with basis” increased slightly (3 percent) compared to A, because more objects and edges were visited, though less objects were compared. The time taken for construction of “B with basis” compared to B was reduced by 55 percent.

2.10 Summary

Topological hierarchical classification methods are more efficient than alternative methods for wide, shallow, multiple inheritance hierarchies. Topological search is more efficient than depth-first search because information about relationships between database objects and the query are used most effectively to constrain which objects are compared. The ancestor space grows slower than the fringe of objects compared in depth-first search. Hierarchical search is more efficient than binary chain search methods because the number of chains in a covering of a hierarchy tends to grow much faster than the number of ancestors of the query.
Chapter 3

Partially Ordered Set Abstract Data Type

Modern computers need to be told the address in memory of an object for the object to be retrieved.

Imagine a fruit, it is round, has a red skin, a white crisp flesh, is juicy, sweet and tangy, ...

A modern computer would need to be told to retrieve an apple by name, a computer with an associative memory would have classified the description as that of (something like) an apple or at least very close. This allows partial descriptions or partial addressing. An associative memory would be pointing in the vicinity of round, red-skinned, ...fruit.

This chapter describes an abstract data type (ADT) for partially ordered sets of objects. This abstract data type is particularly useful for managing larges sets of complex objects. The operations that are provided in the poset abstract data type include

- \textit{member}(u, P) - true iff \( u \in P \).

- \textit{insert}(u, P) - \( P \leftarrow P \cup \{u\} \).
• \(\text{delete}(u,P) - P \leftarrow P - \{u\}\).

• \(\text{subset}(P,Q)\) - true iff \(P \subseteq Q\).

• \(\text{union}(P,Q) - P \leftarrow P \cup Q\).

• \(\text{intersection}(P,Q) - P \leftarrow P \cap Q\).

This ADT has been implemented in C++ in the Peirce system [Ell93c, Ell93b]. These routines are useful for implementing complex object databases. They can be used to support querying, asserting objects, retracting objects, matching databases against one another, merging databases, and generalising databases. We detail algorithms and examples for each method.

### 3.1 Member: A Hierarchy Primitive

```
function member(u : O, P : PO, output Parents_ = : PO)
returns boolean;
begin
    Parents_ = = parents_topological(u, P);
    return Parents_ = = \{ u \};
end
```

Figure 3.1: An algorithm for testing for object membership in a hierarchy

Testing whether an object is a member of the database is a primitive operation used by many of the other methods in the ADT. We implement member by searching for the parents of an object using the \(\text{ancestors\_topological}\) algorithm. The \(\text{ancestors\_encoded}\) algorithm in Figure 4.5 in chapter 4 could also be used. Figure 3.1 gives the basic algorithm for \(\text{member}\). The \(\text{parents\_topological}\) algorithm in Figure 2.9 in chapter 2 returns \(\text{Parents}_\text{=} (u,P)\). If \(u \in P\), then \(\text{Parents}_\text{=} (u,P) = \{ u \}\). If the object \(u\) is not a member of \(P\), we may wish to insert it. In this case, the parents are returned in the argument \(\text{Parents}_\text{=}\) for use in subsequent computations.
3.2 Insert: A Primitive for Constructing Hierarchies

Hierarchy construction can be viewed as a sequence of \textit{insert} or \textit{classification} operations. To insert an object \(u\) into a hierarchy, all relationships between \(u\) and every object in the database must be found. More specifically, the parents and the children must be found. The \textit{insert} algorithm is similar to that used in [Lev85].

\begin{verbatim}
procedure insert(u : O, P : PO);
begin
  if ~member(u, P, Parents) then
    insert_aux(u, Parents, children_topological(Parents, u, P), P);
end
\end{verbatim}

Figure 3.2: An algorithm for inserting an object in a hierarchy

\begin{verbatim}
procedure insert_aux(u : O, Parents : PO, Children : PO, P : PO);
begin
  P ← P ∪ \{u\};
  Parents(u, P) ← Parents;
  Children(u, P) ← Children;
  for each v ∈ Parents(u, P) do
    Children(v, P) ← (Children(v, P) − Children(u, P)) ∪ \{u\};
  for each v ∈ Children(u, P) do
    Parents(v, P) ← (Parents(v, P) − Parents(u, P)) ∪ \{u\};
end
\end{verbatim}

Figure 3.3: An algorithm for inserting a known object in a hierarchy

The \textit{insert} algorithm in Figure 3.2 first checks to see if \(u\) is a member of \(P\). If \(u\) is not a member of \(P\), then \(u\) is inserted. The algorithm \textit{member}(u, P, Parents) in Figure 3.1 computes \(Parents = Parents(u, P)\). If \textit{member} returns false, then \(Children(u, P)\) is computed, and \(u\) is inserted between the parents and children of \(u\) using the \textit{insert_aux} algorithm. As well as adding \(u\) to \(P\) and storing the parents and children of \(u\), \(u\) is added to the children set of \(u\)'s parents, and to the parent set of \(u\)'s children. Links between the children and parents of \(u\) are removed to avoid transitive links.
Figure 3.4: An example of inserting a chemical compound into a chemical compound database (hierarchy)

Figure 3.4 illustrates the insertion of the chemical graph $u$ into the database on the left. Object $u$ is inserted under its parents, $a$ and $b$, and above its children $d$ and $g$. Links between $a$ and $d$ and $b$ and $d$ are removed.

### 3.3 Delete: Retracting Objects from Hierarchies

```plaintext
procedure delete(u : O, P : PO);
begin
  if member(u, P) then
    delete_aux(u, P);
end
```

Figure 3.5: An algorithm for deleting an object from a hierarchy

The delete method is used for retracting objects from object databases. The `delete` method in Figure 3.5 first checks to see if $u$ is a member of $P$. If $u$ is in $P$ then the actual deletion is done by the `delete_aux` algorithm in Figure 3.6. The `delete_aux` algorithm removes links between $u$ and its parents and children. The removal of $u$ and its links from the hierarchy may cause parents and children of $u$ to no longer be related in the hierarchy data structure, in which case direct links between them must be added. The algorithm for checking whether there is a path from $v$ to $u$ in poset $P$ is given in Figure 3.7. The algorithm is a depth-first search restricted to searching down to the same level as $u$ for a path. The test $u = v$ is a simple test for
3.3. DELETE: RETRACTING OBJECTS FROM HIERARCHIES

---

```plaintext
procedure delete\_aux(u : O, P : PO);
begin
  for each v ∈ Children(u, P) do
    Parents(v, P) ← Parents(v, P) − {u};
  end
  for each v ∈ Parents(u, P) do
    Children(v, P) ← Children(v, P) − {u};
  end
  for each w ∈ Parents(u, P) do
    for each v ∈ Children(u, P) do
      if ¬path(w, v, P) then
        Children(w, P) ← Children(w, P) ∪ {v};
        Parents(v, P) ← Parents(v, P) ∪ {w};
    end
  end
  P ← P − {u};
end
```

Figure 3.6: An algorithm for deleting a known object from a hierarchy

---

```plaintext
procedure path(v : O, u : O, P : PO) return boolean;
begin
  if u = v then return TRUE;
  else for each w ∈ Children(v) do
    if level(w) ≤ level(u) ∧ path(w, u, P) return TRUE;
  end
  return FALSE;
end
```

Figure 3.7: An algorithm for checking whether there is a path between two objects in a hierarchy
equality (of pointers) rather than a subsumption test.

For example, consider the deletion of u from the hierarchy on the right in Figure 3.4. The links attaching parents \{a, b\} and children \{d, g\} and u are removed. Then links are attached between a and d, and b and d, since there is no other path to d from a or b.

### 3.4 Subset: Hierarchy Match

```
function subset(P : PO, Q : PO) returns boolean;
begin
  L ← \{ ⊤_P\};
  π ← \{ (⊤_P, ⊤_Q)\};
  while L ≠ \{ ⊥_P\} do
    begin
      L ← Crown(Children(L, P));
      for each u ∈ L do
        if ¬member_initial(u, Ancestors_(π(Parents(u, P)), Q), X, π, Q);
          return FALSE;
        end
      end
    return TRUE;
  end
```

Figure 3.8: An algorithm for testing whether one hierarchy is a subhierarchy of another

The subset method tests whether the poset P is a subset of the poset Q, \( P \subseteq Q \). In the following methods we assume the knowledge bases P and Q are partially ordered by the same order \( \sqsubseteq \). A naive subset algorithm would iterate through elements \( u \) of \( P \) checking \( u \) is a member of \( Q \) using ancestors_topological. However, this will redo work done when searching for \( v \in Ancestors(u, P) \) in \( Q \), since if \( v \sqsubseteq u \) then \( Ancestors(v, Q) \subseteq Ancestors(u, Q) \). The mapping \( π : P \rightarrow Q \) is a partial mapping of elements of \( P \) into elements of \( Q \), which is computed during the database matching operation. If all ancestors of \( u \) have been mapped to elements in \( Q \) before \( u \) is checked, then the \( Ancestors_(π(Parents(u, P)), Q) \) are known to be ancestors of \( u \) in \( Q \). The notation \( π(S) \) represents the set \( \{ v : range(π) \mid ∃ w ∈ S \land π(w) = v \} \).
The *subset* algorithm in Figure 3.8 ensures this property by topologically walking \( P \) mapping elements \( u \) of \( P \) into \( Q \) with the *member* \(_\text{initial}\) algorithm. The argument \( X \) in the call to *member* \(_\text{initial}\) is the returned parents of \( u \) which are ignored.

\[
\text{function } \text{member}_\text{initial}(u : O, K : \text{PO}, i/o \pi : \text{PO} \times \text{PO}, \text{output Parents}_\text{=} : \text{PO}, Q : \text{PO}) \text{ returns boolean;}
\]

\[
\text{begin}
\]

\[
\text{Parents}_\text{=} \leftarrow \text{parents}_\text{topological}_\text{initial}(u, K, \pi, Q);
\]

\[
\text{if } \text{Parents}_\text{=} = \{v\} \land u \sqsupseteq v \text{ then } \pi \leftarrow \pi \cup \{u \mapsto v\}; \text{ return TRUE;}
\]

\[
\text{else return FALSE;}
\]

\[
\text{end}
\]

Figure 3.9: Checking for object membership given some known ancestors

The *member*_\(_\text{initial}\) function is used to check whether a member of \( P \), \( u \), is a member of some poset \( Q \). The second argument of *member*_\(_\text{initial}\), \( K \), is a subset of elements of \( \text{Ancestors}_\text{=} (u, Q) \). The third argument is a partial mapping \( \pi : P \mapsto Q \) from elements of \( P \) to elements of \( Q \). The fourth argument \( \text{Parents}_\text{=} \) is the parents of \( u \) in \( Q \), if \( u \) is not a member of \( Q \). If \( v \) is an occurrence of \( u \) in \( Q \), then \( u \) is mapped to \( v \) in \( \pi \).

The *ancestors*_\(_\text{topological}_\text{initial}\) algorithm which is called by *parents*_\(_\text{topological}_\text{initial}\) is a modified version of *ancestors*_\(_\text{topological}\) algorithm where the lines

\[
\text{Ancestors} \leftarrow \text{ancestors}_\text{brute}(u, L);
\]

\[
L \leftarrow \text{Crown} (\text{Children} (\text{Ancestors}, P));
\]

which finds all of the ancestors of \( u \) in the current layer \( L \) of the topological walk and compute the next layer are replaced with

\[
\text{tmp} \leftarrow L \cap \text{range}(\pi);
\]

\[
\text{Ancestors} \leftarrow \text{ancestors}_\text{brute}(u, L - \text{tmp});
\]

\[
L \leftarrow \text{Crown} (\text{Children} (\text{Ancestors} \cup \text{tmp}, P));
\]

which records in \( \text{tmp} \) the elements of the current layer \( L \) known to be ancestors of \( u \). The elements of the \( L - \text{tmp} \) are compared to see if they are ancestors of \( u \). The next layer \( L \) may include children of elements in \( \text{tmp} \) as well as \( \text{Ancestors} \) of \( u \) in the current layer, since there may be more specialised ancestors of \( u \) under known ancestors of \( u \) in \( \text{range}(\pi) \). This method avoids comparing known ancestors of \( u \) in
$L$, that is elements of $L \cap \text{range}(\pi)$.

As an example consider the comparison of the hierarchy $A$ in Figure 3.10 and the hierarchy $C$ in Figure 3.14. In the following examples we assume that the databases share the same symbols, primitives and basis graphs. Initially the basis graphs of $A$ are mapped to the basis graphs of $C$, that is initially $\pi = \{(p, qa), (q, qb), (r, qc), (o, qe), (i, qf), (n, qg), (j, qh), (k, qi), (m, qj)\}$. The search in subset starts with $L = \text{Base}(\text{Basis}(A)) = \{o, i, n, j, k, m\}$ rather than $\top$. On the first iteration the next layer under the basis is searched ($L = \{a, b\}$). The following table shows the objects which were compared when checking each element of $A$ for membership of $C$.

<table>
<thead>
<tr>
<th>$v$</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>qk, c</td>
<td>b</td>
<td>c, f</td>
<td>ql, d</td>
<td>e</td>
<td>f</td>
<td>ql, g</td>
<td>h</td>
</tr>
</tbody>
</table>

The ancestors encoded algorithm described in chapter 4 and in [Ell93a] would further improve this by removing $ql$ from the comparisons for $d$. This is because $ql$ has no descendants in common with the descendants of all of the ancestors of $d$. Similarly $ql$ would not need to be compared with $g$. The object $c$ would not need to be compared if size was used as a precondition for testing, since $c$ is larger than $a$ and hence could not be a subcompound. The problem with $c$ and $f$ is that they have only one parent and hence have less constraints preventing comparisons.

### 3.5 Union: Hierarchy Merge

The union method can be used to merge databases. The result of merging posets $P$ and $Q$ is the hierarchy over $P \cup Q$. The merge operation can be seen as a database unification operation. The union algorithm in Figure 3.12 could be used as the basis for the harder problem of knowledge base integration, and as the merge function in a divide and conquer merge sort algorithm. Merging of two totally ordered sets can be computed in time linear in the size of the sets. Merging partially ordered sets is not necessarily linear, but as this algorithm illustrates the order information in the two hierarchies being merged can be exploited to reduce the number of comparisons.

A naive algorithm for merging two hierarchies $P$ and $Q$ would be to walk the
Figure 3.10: Example hierarchy A

Figure 3.11: Example hierarchy B
**Procedure union**\( (P : \mathbb{PO}, Q : \mathbb{PO}) \);

begin
\begin{align*}
L &\leftarrow \{ \top_P \}; \\
\pi &\leftarrow \{ (\top_P, \top_Q) \}; \\
\text{while } L &\neq \{ \bot_P \} \text{ do} \\
&\begin{align*}
&\text{for each } u \in L \text{ do} \\
&\quad \text{insert } \text{ initial}\( u, \text{Ancestors}_\pi(\text{Parents}(u, P), Q), \pi, Q \); \\
&L \leftarrow \text{Crown}(\text{Children}(L, P)) ;
&\end{align*}
&\end{align*}
\end{align*}

end

Figure 3.12: An algorithm for merging two hierarchies

Hierarchies of \( P \) and insert each element \( u \) of \( P \) in the hierarchy of \( Q \) using the *insert* in Figure 3.2. However, it is useful to use a similar algorithm to the *subset* algorithm which used the known ancestors of \( u \) in \( Q \), that is \( \text{Ancestors}_\pi(\text{Parents}(u, P), Q) \). This property is ensured again by a topological walk of \( P \).

**Procedure insert initial**\( (u : O, i/o K : \mathbb{PO}, \text{output } \pi : \mathbb{PO} \times \mathbb{PO}, Q : \mathbb{PO}) \);

begin
\begin{align*}
&\text{if } \neg \text{member initial}\( u, K, \pi, \text{Parents}, Q \) \text{ then} \\
&\quad \text{insert aux}\( u, \text{Parents}, \text{children topological}(u, \text{Parents}, Q), Q \); \\
&\quad \pi \leftarrow \pi \cup \{ u \mapsto u_Q \}; \\
&\end{align*}

end

Figure 3.13: Inserting an element given some known ancestors

The algorithm *insert initial* is used for inserting elements of poset \( P \) into the poset \( Q \). The mapping \( \pi : P \mapsto Q \) is used to record where \( u \) is inserted or already occurs. Known ancestors \( K \) are given as a starting point for the search for \( u \) in \( Q \).

Consider the operation \( \text{union}(A, B) = C \) where \( A, B \) and \( C \) are in Figures 3.10, 3.11 and 3.14, respectively. Since an element \( u \) of \( P \) is inserted in \( Q \), \( u \) is classified into \( Q \), thus there are two columns in the following table for ancestor and descendant search.

<table>
<thead>
<tr>
<th>( v )</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparisons (Ancestor Search)</td>
<td>qk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparisons (Descendant Search)</td>
<td>qm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Since $A$ and $B$ only have the object $a = qk$ in common other than the basis objects, the merge function does very few comparisons in this example. This example illustrates that hierarchies are very useful records of order information that can be exploited in operations on hierarchies.

### 3.6 Intersection: Hierarchy Unification

The *intersection* method can be used for computing the common generalisation of two databases. The *intersection* algorithm in Figure 3.15 topological walks the hierarchy $P$, and uses the *member initial* to search for elements of $P$ in $Q$. The *member initial* algorithm takes advantage of the known ancestors $\text{Ancestors}_{\leq}(\pi(\text{Parents}(u, P)), Q)$. Any element in $P$ which is not a member of $Q$ is deleted from $P$.

Figure 3.16 shows the result of intersecting the hierarchies $B$ (Figure 3.11) and $A$ (Figure 3.10). $D$ is the common subhierarchy.
procedure intersection(P : PO, Q : PO);
begin
    L ← \{\top_P\};
    \(\pi \leftarrow \{(\top_P, \top_Q)\}\);
    while \(L \neq \{\bot_P\}\) do
        begin
            for each \(u \in L\) do
                if \(\neg \text{member\_initial}(u, \text{Ancestors}_=(\pi(\text{Parents}(u, P)), Q), \pi, Q)\) then
                    delete\_aux(u, P);
                end
            \(L \leftarrow \text{Crown}(\text{Children}(L, P))\);
        end
end

Figure 3.15: An algorithm for intersecting two hierarchies

\[
\begin{align*}
D & \quad T \\
\end{align*}
\]

Figure 3.16: Intersection of the hierarchies in Figures 3.11 and 3.10
3.7 Summary

In this chapter we have described a partially ordered set abstract data type. The methods supported in this ADT include: insertion of objects, creation of hierarchies, matching databases against each other, unifying two databases, and finding the common generalisation of two databases. These operations can be used when matching, unifying, and generalising nested graphs which may have graph databases as labels on the nodes. The methods illustrate the ability of hierarchies to store useful ordering information that can be exploited in hierarchical operations. Methods such as union may be useful in divide and conquer approaches to operations such as classifying partially ordered sets.
Chapter 4

Analysis of Hierarchies for Efficient Search

Hierarchies can be recognised as kinds of orders. A hierarchy can be decomposed into structure hierarchies and label hierarchies, which can further be factorised for search.

It has been shown in chapter 2 that careful use of the ordering information in a database of objects improves retrieval of information. By recording the parent and child information, and hence ancestor and descendant information, it was shown how significant numbers of object comparisons can be pruned during classification of an object. The knowledge that \( v \sqsupseteq u \) only if \( \forall w : \text{Ancestors}(v, P) \bullet w \sqsubseteq u \) can be used as a precondition for comparing the database object \( v \) to the query object \( u \). The object \( v \) can only match the query \( u \) if all ancestors of \( v \) match \( u \).

In this chapter we show how further analysis of the ordering in a object database can significantly reduce object comparisons in search. Intersection of descendants of objects can be used to improve ancestor search when searching for descendants of the query. An algorithm which takes advantage of unification operations is given for descendant search. The algorithm uses the fact that if an object \( v \) does not unify with the query \( u \), then \( v \) cannot contain \( u \). An algorithm is given which efficiently tests for membership of recursively structured hierarchies.
4.1 Ancestor Search using Intersection of Descendants

In chapter 1, we illustrated a hierarchical solution to the boxes problem which Stonebraker [Sto89] gave as an application that is badly served by the relational database model. The boxes represent the layout of a city. The city is broken down into suburbs, suburbs into blocks, and blocks into buildings. Each of these areas is represented as a box. Figure 4.1(a) gives a sample data set where the boxes overlap. Each box can be represented by the co-ordinates of the lower left corner \((x_1, y_1)\) and the upper right corner \((x_2, y_2)\). In a relational database, each box could be stored in a tuple of the form \(\text{box}(id, x_1, y_1, x_2, y_2)\) where \(id\) is the box identifier. Table 4.1 gives the box relation for the aerial view of the city in Figure 4.1(a).

\[
\text{Table 4.1: Collection of box tuples}
\]

<table>
<thead>
<tr>
<th>box(a, 0.0, 0.0, 5.8, 4.6)</th>
<th>box(f, 2.8, 2.1, 3.5, 3.5)</th>
<th>box(j, 4.4, 2.7, 5.1, 3.6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>box(b, 0.3, 0.7, 3.6, 3.6)</td>
<td>box(g, 2.5, 1.3, 3.3, 2.9)</td>
<td>box(k, 4.0, 1.2, 4.5, 2.3)</td>
</tr>
<tr>
<td>box(c, 2.1, 0.4, 5.5, 4.3)</td>
<td>box(h, 1.1, 1.3, 1.8, 2.2)</td>
<td>box(l, 4.7, 0.6, 5.4, 2.2)</td>
</tr>
<tr>
<td>box(d, 0.7, 2.8, 1.1, 3.4)</td>
<td>box(i, 2.9, 2.3, 3.1, 2.7)</td>
<td>box(m, 4.9, 1.2, 5.3, 2.0)</td>
</tr>
<tr>
<td>box(e, 0.9, 1.1, 1.9, 2.4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A typical query may be to find all boxes in a given region, for example, in Figure 4.1(a) the region defined by the thick outlined box \(u\) with corners \((2.1, 1.9)\) and \((3.6, 3.0)\). This could be formulated in the SQL relational query language [Cod70] as
4.1. ANCESTOR SEARCH USING INTERSECTION OF DESCENDANTS

select * from box where \( x_1 \geq 2.1 \) and \( y_1 \geq 1.9 \) and \( x_2 \leq 3.6 \) and \( y_2 \leq 3.0 \)

As discussed in chapter 1, if there are \( N \) boxes, search in a relational databases will compare \( N \) boxes. A hierarchy provides an efficient intuitive solution to the box problem. The boxes in Figure 4.1(a) can be arranged into the hierarchy Figure 4.1(b) induced by the inclusion relationship over boxes defined in chapter 1.

Consider the search for boxes contained in the box \( u \) in Figure 4.1(a). This example illustrates ancestor search where ancestor boxes overlap or have multiple inheritance. The overlap \( w \) between boxes \( v \) and \( u \) is

\[
((x_{1w}, y_{1w}), (x_{2w}, y_{2w})) = ((\max(x_{1v}, x_{1u}), \max(y_{1v}, y_{1u})), (\min(x_{2v}, x_{2u}), \min(y_{2v}, y_{2u})))
\]

There is an overlap if \( x_{1w} < x_{2w} \land y_{1w} < y_{2w} \). In previous methods for search for ancestors [Lev84, Lev92], the boxes \( a, b, c, d, e, f, g, j, k \) and \( l \) would be compared. In Figure 4.1 the fact that box \( b \) includes the query \( u \) and box \( c \) also contains \( u \), implies that \( u \) is in the overlap of \( b \) and \( c \). Further, boxes which are included in \( u \) are in the intersection of the descendants of \( b \) and \( c \), where \( \text{Descendants}_u(b, P) \) \( \cap \) \( \text{Descendants}_u(c, P) = \{b, d, e, f, g, h, i, \bot\} \cap \{c, f, g, i, j, k, l, m, \bot\} = \{f, g, i, \bot\} \). We define \( \text{Focus} \) as the intersection of the descendants of all known ancestors of \( u \) in \( P \). Thus after \( b \) and \( c \) are found \( \text{Focus} = \{f, g, i, \bot\} \). The box \( d \) would normally be compared, since all ancestors of \( d \), in this case \( b \) and \( a \), match the query \( u \). That is, all boxes which contain box \( d \) contain the query box \( u \). However, \( d \) has no descendants other than \( \bot \) in common with the descendants of \( b \) and \( c \), that is, \( \text{Descendants}_u(d, P) \) \( \cap \) \( \text{Focus} = \{\bot\} \). Thus \( d \) does not need to be compared to \( u \). Similarly, boxes \( e, j, k, \) and \( l \) can also be pruned. Boxes \( f \) and \( g \) are not pruned and are compared to see if they contain \( u \).

The precondition that all ancestors of an object \( v \) must match \( u \) if \( v \) is to match \( u \) can be strengthened. Assume there is some common descendant in \( P \) other than \( \bot \) of all ancestors of \( u \) in \( P \). This is a weaker assumption than \( u \) or some descendant of \( u \) is in \( P \). Given this assumption, \( v \equiv u \) only if \( v \) has some descendant in the intersection of the descendants of all known ancestors of \( u \). Thus if there are descendants of \( u \) in \( P \), this test will not prune ancestors of \( u \).
It should be made clear that this method relies on a common descendant other than ⊥ of all u’s ancestors being present to return u’s ancestors. If there are no common descendants present, then a subset of ancestors of u are returned. The method will always return at least one of the parents of u, but not necessarily all. In the worst case, the descendants of the single parent of u can be searched using descendants topological algorithm in chapter 2. The ancestors encoded algorithm in Figure 4.5 prunes twigs of the ancestor space of u. The twigs are nodes of the fringe of the ancestor space (see chapter 2 for formal definition) which have all their parents in the ancestor space of u, these include the shoots (defined in chapter 2, definition 2.2.1) which are nodes with a single parent and the parent is an ancestor of u.

Consider the search for u in Figure 4.2 which is similar to the hierarchy in Figure 4.1 except the common descendants, {f, g, i}, of b and c have been removed. In this example, initially the Focus = \{a, b, c, d, e, h, j, k, l, m, ⊥\}. The object a is compared since it has a descendant in common with the Focus other than ⊥. Since a is found to be an ancestor of u, Focus = \{a, b, c, d, e, h, j, k, l, m, ⊥\}. Similarly, object b is compared and hence Focus = \{b, d, e, h, ⊥\}. The object c is pruned from the search since Focus ∩ Descendants_= (c, P) = \{b, d, e, h, ⊥\} ∩ \{c, j, k, l, m, ⊥\} = \{⊥\}. Search terminates with one parent b in this example. To determine if there are any descendants of u, the descendants of b are searched. In this example b, d, e and h are compared. Later in the chapter we show a
4.1. ANCESTOR SEARCH USING INTERSECTION OF DESCENDANTS

Figure 4.3: An encoded aerial view of a city

descendants-unification search which would avoid the comparison of \( h \) due to \( e \) not overlapping with \( u \). The ancestors-encoded method will be useful in domains where the number of twigs avoided is larger than the possible extra comparisons in the descendants-topological search to determine there are no descendants of \( u \) in the poset. The ancestors-encoded method is useful if there are solutions to find, but not useful if search is likely to fail.

It is not necessary to actually compute the intersection of boxes \( b \) and \( c \) in Figure 4.3. Rather by mapping the hierarchy of boxes into a description space such as Prolog Herbrand terms it is possible to check for overlap by doing unification of small terms. Consider the mapping of the hierarchy in Figure 4.3(a) into a term space in Figure 4.3(b). The boxes \( b \) and \( c \) are mapped into the terms \(<1>\) and \(<\_1>\) respectively (In chapter 5 we give methods for encoding hierarchies). The “\(_\_\)” character is an unnamed variable. The terms are like open lists of variables with possible instantiations to integers and are padded to the right with variables. The two terms do unify to produce \(<1,1>\). The ordering over the terms obeys the ordering over the boxes: the box \( e \) includes box \( h \), and the term of \( e \), \(<1,3>\), is more general than the term of \( h \), \(<1,3,1>\), since for every position in the term of \( e \) if the variable in that position is instantiated to some integer, then the variable in the same position in the term of \( h \) is instantiated to that same integer. The important point about this technique is that it is independent of the object complexity, but is simply based on the topology of the hierarchy of objects. The codes of a hierarchy of boxes will be the same as the codes of a hierarchy of program specifications which has the
same shape. The ancestors_encoded method introduced below provides increasing benefits in proportion with the increasing complexity of the objects in the hierarchy. We now discuss the search algorithm in detail and prove some properties used in the search and in the proof of correctness of the algorithm.

**Proposition 4.1.1** If $S$ is a non-empty subset of $\text{Ancestors}_=(u, P)$, then $\text{Descendants}_=(u, P) \subseteq \bigcap_{v \in S} \text{Descendants}_=(v, P)$.

**Proof:** Let $w \in \text{Descendants}_=(u, P)$. Therefore, from the definition of $\text{Descendants}_=\cdot u \sqsupseteq w$. Let $v \in S$. Therefore $v \sqsupseteq u$ and so $v \sqsupseteq w$ by transitivity of $\sqsupseteq$. Therefore $w \in \text{Descendants}_=(v, P)$. Since $v$ is an arbitrary member of $S$, then $w \in \bigcap_{v \in S} \text{Descendants}_=(v, P)$. □

Thus an object $v$ is not an ancestor of $u$ if $v$'s descendants are disjoint from the intersection of the descendants of known ancestors of $u$, given there are objects which are descendants of all the ancestors of $u$ in $P$. The Focus (intersection of descendants= of known ancestors= of $u$) is a constraint on search, pruning twigs (defined in chapter 2, definition 2.2.1 as the nodes whose parents are all ancestors of $u$, but are not ancestors of $u$ themselves) which diverge from the solutions (see Figure 4.4). We use the following proposition in the proof of correctness of the ancestors_encoded algorithm.

**Proposition 4.1.2** If $w \neq \bot$ and \{\bot\} $\subseteq \bigcap_{v \in \text{Ancestors}_=(u, P)} \text{Descendants}_=(v, P)$, and $S$ is a non-empty subset of $\text{Ancestors}_=(u, P)$, then

$$((\text{Descendants}_=(w, P) \cap \bigcap_{v \in S} \text{Descendants}_=(v, P)) = \{\bot\}) \Rightarrow w \not\sqsupseteq u.$$

**Proof:** We assume

$$(\text{Descendants}_=(w, P) \cap \bigcap_{v \in S} \text{Descendants}_=(v, P)) = \{\bot\} \land w \sqsupseteq u$$ and prove a contradiction.

Firstly, since $S \subseteq \text{Ancestors}_=(u, P)$ and

$$(\text{Descendants}_=(w, P) \cap \bigcap_{v \in S} \text{Descendants}_=(v, P)) = \{\bot\}$$ implies
(Descendants$(w, P) \cap \bigcap_{v: \text{ancestors}(u, P)} \text{Descendants}^{-}(v, P)) = \bot.

Secondly, $\{\bot\} \subset \bigcap_{v: \text{ancestors}(u, P)} \text{Descendants}^{-}(v, P)$, so there is some $x : P$ such that $x \neq \bot$ and $\forall v : \text{Ancestors}(u, P) \bullet v \sqsupset x$. Since $w \sqsupseteq u$ then $w \sqsupseteq x$. Therefore $x \in \text{Descendants}^{-}(w, P)$, leading to a contradiction. $\Box$

The new ancestors-encoded algorithm, first reported in [Ell93a], finds the ancestors of $u$ in the poset $P$, if there is a common descendant of all the ancestors of $u$ in $P$. A modified version of the ancestors-topological algorithm is given in Figure 4.5. The line $\text{Ancestors} \leftarrow \text{ancestors-brute}(u, L)$ of the algorithm in Figure 2.21 in chapter 2 is replaced with the line $\text{Ancestors} \leftarrow \text{ancestors-layer}(u, \text{Focus}, L, P)$. The ancestors-layer algorithm finds the ancestors of $u$ in the layer $L$. The line in Figure 4.6 $w \leftarrow \text{choose}(N)$ selects a node of the layer $N$. The ancestors of $u$ are accumulated in $M$.

Using Levinson’s [Lev92] ancestors-topological algorithm described in chapter 2 to find the ancestors of $u$ in Figure 4.7, nodes $a, b, c, d,$ and $e$ are compared where only chemicals $a$ and $b$ are subcompounds. Using ancestors-encoded the node $c$ can be pruned because it has no descendants other than $\bot$ in common with the descendants of the known ancestors ($\{a, b\}$) of $u$. Initially, the Focus is the whole database $\{a, b, c, d, e, f, g, h, \bot\}$. On the first call to ancestors-layer, $L = \{\top\}$ and Focus = $\{a, b, c, d, e, f, g, h, \bot\}$. Since the intersection of Focus and the descendants of $\top$ strictly includes $\bot$, $\top$ is compared and returned. On the next iteration, $L = \{a, b\}$, Focus = $\{a, b, c, d, e, f, g, h, \bot\}$. Since the intersection of the
function ancestors_encoded(u : O, P : PO) returns PO;
begin
    K ← ∅;
    L ← {⊤};
    Focus ← P;
    while L ≠ {⊥} do
        begin
            Ancestors ← ancestors_layer(u, Focus, L, P);
            K ← K ∪ Ancestors;
            L ← Crown(Children(Ancestors, P));
            L ← ParentsFilter(L, K, P);
        end
    return K;
end

Figure 4.5: An algorithm for finding ancestors using topological search and encoding

function ancestors_layer(u : O, i/o Focus : PO, L : PO, P : PO) returns PO;
begin
    M ← ∅;
    N ← L;
    while N ≠ ∅ do
        begin
            w ← choose(N);
            N ← N – {w};
            if ((Descendants_(w, P) ∩ Focus) ≠ {⊥}) ∧ w ⊇ u then
                begin
                    M ← M ∪ {w};
                    Focus ← Focus ∩ Descendants_(w, P);
                end
            end
        return M;
end

Figure 4.6: An algorithm for finding ancestors in a layer
descendants of $a$ with the $Focus$ also strictly includes $\bot$, then $a$ is compared to $u$. Since $a$ is an ancestor of $u$, the new $Focus$ is $\{a, c, d, f, g, h, \bot\}$. Similarly, the intersection of the descendants of $b$ with the $Focus$ strictly includes $\bot$ and $b$ matches to give a new $Focus$ $\{d, g, h, \bot\}$. On the next iteration, $L = \{c, d, e\}$, $Focus = \{d, g, h, \bot\}$. The only descendant of $c$ in common with the $Focus$ is $\bot$, therefore $c$ can be ignored even though its ancestors matched the query. The intersection of the descendants of $d$ with the $Focus$ strictly contains $\bot$, so is compared, but does not match ($d$ is a supercompound). The intersection of descendants of $e$ with the $Focus$ also strictly includes $\bot$, so $e$ is also compared but does not match. Since there is no object on the next layer whose ancestors all match $u$, search for ancestors terminates. The $ancestors_{encoded}$ algorithm returns $Ancestors_{=}(u, P)$ if there is a common descendant of all the ancestors of $u$ other than $\bot$, otherwise $ancestors_{encoded}$ returns some subset of $Ancestors(u, P)$. This routine is expected to be used before a $descendants$ algorithm when the goal is not strictly classification, but finding descendants of the query.

Levinson [Lev92] uses size as a necessary condition for matching, and in this case would prevent the comparison of $d$ and $e$ but not $c$. For domains where size is a necessary condition for matching this can be used. It may be useful to use a
Figure 4.8: An encoded aerial view of a city reorganised by partitioning

number of simple necessary conditions such as size as a filter on full object comparisons. Levinson [Lev85] also developed methods for adding ancestors of objects in the hierarchy to “self-organise” the hierarchy. Figure 4.8 shows the hierarchy in Figure 4.1 with some extra partition boxes x, y and z added. These boxes increase the preconditions on comparing the boxes inside them. In a ancestors-topological search the objects a, b, c, x, y, z, f and g are compared. Boxes inside x and y are not compared.

We first prove the correctness of the ancestors-layer algorithm given in Figure 4.6 as a basis for the proof of the ancestors-encoded algorithm.

**Proposition 4.1.3** The ancestors-layer algorithm is totally correct with respect to the precondition

\[
\begin{align*}
\top, \bot \in P, u \neq \bot, \\
\{\bot\} \subseteq \bigcap_{v : \text{Ancestors}_=(u, P)} \text{Descendants}_=(v, P), \text{Focus} = \bigcap_{v \in S} \text{Descendants}_=(v, P) \text{ and} \\
L \subseteq P \text{ for some } S \text{ which is a non-empty subset of Ancestors}(u, P)
\end{align*}
\]

and post condition

\[
\begin{align*}
M = \text{Ancestors}_=(u, L) \text{ and} \\
\bigcap_{v \in S} \text{Descendants}_=(v, P) \text{ for some } S \text{ which is a non-empty subset of Ancestors}(u, P).
\end{align*}
\]

**Proof:** We first prove in turn that Focus = \(\bigcap_{v \in S} \text{Descendants}_=(v, P)\) and M = Ancestors_=(u, L - N) is invariant where S is some non-empty subset of Ancestors_=(u, P).
4.1. ANCESTOR SEARCH USING INTERSECTION OF DESCENDANTS  111

The Focus invariant is true initially, since it is a precondition of ancestors-layer.
Initially, $M = \emptyset$ and $N = L$. Thus $\emptyset = \text{Ancestors}_= (u, L - L)$.

After each iteration, $w = \text{Choose}(N)$, which selects an element $w$ of $N$, and
$N' = N - \{w\}$. There are two cases to consider.

(i) $(\text{Descendants}_= (w, P) \cap \text{Focus}) \neq \{\bot\} \land w \sqsubseteq u$

In this case the following variables are updated.

$$
M' = M \cup \{w\}
$$

$$
\text{Focus}' = \text{Focus} \cap \text{Descendants}_= (w, P)
$$

Let $S' = S \cup \{w\}$. Thus $S' \subseteq \text{Ancestors}(u, P)$ since $w \in \text{Ancestors}_= (u, P)$
and $S \subseteq \text{Ancestors}(u, P)$ from the assumed invariant. Therefore $\text{Focus}' =
((\bigcap_{v \in S'} \text{Descendants}_= (v, P)) \cap \text{Descendants}_= (w, P)) = \bigcap_{v \in S'} \text{Descendants}_= (v, P)$ as re-
quired.

Next we show $M' = \text{Ancestors}_= (u, L - N')$.

$$
\text{Ancestors}_= (u, L - N') =
$$

$$
= \text{Ancestors}_= (u, L - (N - \{w\})) \text{ (expand } N')
$$

$$
= \text{Ancestors}_= (u, (L - N)) \cup \{w\} \text{ } (w \in \text{Ancestors}_= (u, L))
$$

$$
= M \cup \{w\} \text{ } \text{ (from invariant)}
$$

$$
= M' \text{ } \text{ (contract } M')
$$

(ii) $\neg((\text{Descendants}_= (w, P) \cap \text{Focus}) \neq \{\bot\} \land w \sqsubseteq u)$ :
Since only \( N \) has changed we need to show \( M' = \text{Ancestors}_\leq(u, L - N') \). There are two cases to consider:

(a) \( w \not\subseteq u \):

\[
\text{Ancestors}_\leq(u, L - N') = \\
= \text{Ancestors}_\leq(u, L - (N - \{w\})) \text{ (expand } N') \\
= \text{Ancestors}_\leq(u, L - N) \quad (w \not\subseteq \text{Ancestors}_\leq(u, L)) \\
= M \quad (\text{from invariant}) \\
= M' \quad (\text{contract } M')
\]

(b) \((\text{Descendants}_\leq(w, P) \cap \text{Focus}) = \{\bot\} \land w \not\subseteq u\):

This case cannot occur because of proposition 4.1.2

At termination \( N = \emptyset \) and \( M = \text{Ancestors}_\leq(u, L - N) \). Therefore
\( M = \text{Ancestors}_\leq(u, L) \) as required. Therefore ancestors layer is partially correct.

The ancestors layer terminates since \( N = L \) is a finite set, and on each iteration the size of \( N \) is decreased by 1, so eventually \( N = \emptyset \). Thus ancestors layer is totally correct. \( \square \)

We now prove the ancestors encoded algorithm is correct.

**Proposition 4.1.4** If \( \top, \bot \in P \), \( u \neq \bot \), and
\[
\{\bot\} \subset \bigcap_{v: \text{Ancestors}_\leq(u, P)} \text{Descendants}_\leq(v, P),
\]
then the ancestors encoded algorithm returns \( \text{Ancestors}_\leq(u, P) \).

**Proof:** The ancestors encoded algorithm is similar to ancestors topological except
\[
K \leftarrow K \cup \text{ancestors brute}(u, L)
\]
is replaced with
\[
K \leftarrow K \cup \text{ancestors layer}(u, \text{Focus}, L, P)
\]
We need to show that ancestors layer returns \( \text{Ancestors}_\leq(u, L) \). This is true if the preconditions of ancestors layer are satisfied on each call as shown in
4.2 DESCENDANT SEARCH USING UNIFICATION

In chapter 1, the example given in Figure 1.1 illustrated that we should not search a box \( v \) for boxes contained in box \( u \), if \( v \) and \( u \) do not overlap. The descendant search in Figure 1.1 started inside box \( h \). The box \( i \) was compared to \( u \) but \( i \) is not contained in \( u \), further \( i \) does not overlap with \( u \). Since \( i \) does not overlap with \( u \), boxes inside \( i \) (j) do not have to be compared to \( u \), because they cannot be contained in \( u \). The overlap \( w \) between boxes \( v \) and \( u \) is

\[
((x_{1w}, y_{1w}), (x_{2w}, y_{2w})) = ((\max(x_{1v}, x_{1u}), \max(y_{1v}, y_{1u})), (\min(x_{2v}, x_{2u}), \min(y_{2v}, y_{2u})))
\]

There is an overlap if \( x_{1w} < x_{2w} \land y_{1w} < y_{2w} \). The cost of this operation is slightly more than comparing two boxes. The cost of the unification operation can be traded off with the cost of all the comparison operations avoided. It may not be necessary to use a full unification test, but rather some weaker necessary condition for unification that is computationally efficient.

Figure 4.9 gives the new descendants\_unification algorithm. The filter \( Clashes(u, L) = \{ v : L \mid u \cap v = \bot \} \) can be added to the descendants\_topological filter on each layer of objects in the search. Where \( u \cap v \) denotes the greatest lower bound or most general unifier of objects \( u \) and \( v \).
function descendants_unification(Parents(u, P) : PO, u : O)
returns PO;
begin
Focus ← \bigcap_{v:Parents(u, P)} Descendants(v, P);
K ← \emptyset; C ← \emptyset;
L ← Crown(Children(Parents(u, P), P));
while L \neq \{\bot\} do
begin
K ← K \cup \text{Descendants}_{=}\left(\text{descendants}_{\text{brute}}(u, L \cap Focus), P\right);
C ← C \cup \text{Descendants}_{=}\left(\text{Clashes}(u, (L \cap Focus) - K), P\right);
L ← Crown(Children(L, P));
L ← (L - K) - C;
end
return K;
end

Figure 4.9: A topological algorithm for finding descendants using unification operations

Restricting \(L\) to \((L - K) - C\) removes from the next layer the known descendants of \(u\) and known nodes which do not unify. To determine the clash nodes in a layer we use \(\text{Clashes}(u, (L \cap Focus) - K)\). That is, we only consider nodes in the \(\text{Focus}\) which are known not to be descendants of \(u\). Any descendants of clash nodes also clash with \(u\), so they are added to \(C\). This unification approach restricts the \(\text{Focus}\) intersection area, by removing nodes known not to unify with the query and hence cannot be descendants of \(u\).

4.3 Factored Search

Search can be improved by analysing a hierarchy into modules. An algorithm is given for a class of hierarchies, known as term lattices, for testing if an object is a member of the hierarchy. This algorithm does \(O(k \log_k n)\) comparisons in contrast to the \(O(k \log_k^2 n)\) comparisons done by a traditional depth-first algorithm.

Figure 4.11 gives an algorithm for \(\text{member}(u, P)\) which tests if \(u\) is a member of a poset \(P\) (that is, if there is a path from some known ancestor \(v\) to \(u\)). The node \(\top\)
Figure 4.10: Comparison of path and modulated search for an node in a term lattice $TL_{3,3}$

```plaintext
function member_path(u : O, P : PO) returns boolean;
begin
    v ← T;
    while ancestoreq_member(u, Children(v, P), w) do
        v ← w;
    return u ⊇ v;
end
```

Figure 4.11: An depth-first algorithm for checking an node is a member of a poset
is a known ancestor of any \( v \), but if other closer ancestors of \( u \) are known they can be used. The \textit{member.path} algorithm selects an ancestor \( w \) of \( u \) from the children of \( v \), then tests if there is a path from \( w \) to \( u \). The program \textit{ancestreq.member}(\( u, S, w \)) returns TRUE if there is a \( w \) in \( A_\equiv(u, S) \) and selects the first such \( w \), otherwise returns FALSE. A path \( w_1 \sqsupseteq w_2 \sqsupseteq \ldots \sqsupseteq w_n \) is constructed where \( w_1 = v \) and \( w_n = u \) if \( u \) is in \( P \). This algorithm is similar to an earlier version published in [El89b].

Consider the search for \( u \) \( (<2,2,2>) \) in the left hierarchy in Figure 4.10 using the \textit{member.path} algorithm. The search order is \( \top = <0,0,0> \) (assumed ancestor), \( <1,0,0> \) (not ancestor), \( <0,1,0> \) (not ancestor), \( <0,0,1> \) (not ancestor), \( <0,0,2> \) (ancestor), \( <1,0,2> \) (not compared, known non-ancestor), \( <0,1,2> \) (not compared, known non-ancestor), \( <0,2,2> \) (ancestor), \( <1,2,2> \) (not compared, known non-ancestor), \( <2,2,2> \) (ancestor, actually \( u \)). Nodes with filled circles like \( <0,2,2> \) are objects which are ancestors of the query \( u \). Nodes with filled boxes like \( <1,0,0> \) are compared and found to be non-ancestors \( u \). Nodes with nonfilled boxes like \( <1,0,2> \) are known non-ancestor objects which were checked but not compared if non-ancestor information is propagated. We prove the correctness of the algorithm by first proving the main auxiliary function \textit{member.path} is correct.

**Proposition 4.3.1** Given \( \top \in A_\equiv(u, P) \), the \textit{member.path} algorithm returns the truth value of \( u \in P \).

**Proof:** First we show that \( v \in A_\equiv(u, P) \) is invariant.

Initially, this is a requirement of \( \top \) so we need only show \( v' \in A_\equiv(u, P) \) at the end of each iteration. After each iteration,

\( w' \in \text{Children}(v, P) \land w' \in A_\equiv(u, P) \)

and so \( v' \in A_\equiv(u, P) \) as required.

If the program terminates, then \( (\text{Children}(v, P) \cap A_\equiv(u, P)) = \emptyset \) and \( v \in A_\equiv(u, P) \). There are two cases:

(i) \( u \sqsupseteq v \): Thus \( u = v \) by the antisymmetric property of \( \sqsupseteq \). Further \( u \in P \) and the program returns TRUE. 
\(4.3. \text{FACTORED SEARCH}\)

(ii) \(u \not\supseteq v\): Hence \(v \supseteq u\). Assume \(u \in P\). Then
\[\exists w: P \land w \in \text{Children}(v, P) \land w \notin \text{Ancestors}(u, P)\]
resulting in a contradiction. Hence \(u \notin P\) and FALSE is returned. Hence \text{member\_path} is partially correct.

We now show termination. Initially \(v \in \text{Ancestors}(u, P)\) and after each iteration \(v' \in \text{Children}(v, P)\). Since \(P\) is finite and acyclic \(\text{Ancestors}(u, P) \cap \text{Children}(v, P)\) must eventually be empty and hence \text{member\_path} terminates. Thus \text{member\_path} is totally correct. \(\square\)

We now analyse the number of objects visited.

**Proposition 4.3.2** The algorithm \text{member\_path} visits \(O(k \log_k^2 n)\) objects in a term lattice \(TL_{k,d}\).

**Proof:** The algorithm is a depth-first search and in the worst case visits all of the nodes of \(\text{Children}(v, P)\) and may have to search to the bottom of the hierarchy. The depth of a term lattice \(TL_{k,d}\) is \(d = \log_k n\), and at worst there are \((k - 1) \cdot \log_k n\) children of an element. Therefore at worst \(O(k \log_k^2 n)\) objects are visited. \(\square\)

In Figure 4.10, only 6 objects are compared which would indicate an upper bound of \(O(k \log_k n)\) and is probably the case assuming non-ancestor information is propagated. Thus even though \(O(k \log_k^2 n)\) objects are visited many less are compared if search information is propagated. The \text{member\_path} algorithm given in Figure 4.11 does not propagate this information. For example the objects marked by non-filled boxes \(<1,0,2>, <0,1,2>\) and \(<1,2,2>\) would be compared to \(u\) even though some of their ancestors are known non-ancestors of \(u\). There is a question of whether the non-ancestor information can be propagated efficiently. If \(v\) is a non-ancestor of \(u\), it would be possible to mark all descendants of \(v\) as non-ancestors of \(u\), but this is expensive for \(v\) with large numbers of descendants such as nodes in the primitive levels high in the hierarchy. Inheritance hierarchies fan out on descent, in which case it makes more sense to look backwards at ancestors which will converge.

It is possible to design efficient algorithms for hierarchies such as term lattices because of their regular structure. We propose a method for member search which
Figure 4.12: A module can be treated as a single node in a hierarchy

Figure 4.13: Modulation and factorisation of a hierarchy

Figure 4.14: Hierarchy of factors

Figure 4.15: Cover tree of a modulated term lattice $TL_{3,3}$
makes use of these regularities. We first decompose the hierarchy into modules. Hassan Aït-Kaci et al. defined a module [AKBLN89] as a group of nodes which can be encircled such that they share a unique greatest node, known as local top, and a unique least node, known as local bottom, where for every node \( u \) in the module if \( u \) has a path to some descendant \( v \) outside the module it must be through the local bottom. If there is a path from some ancestor \( v \) outside the module to some node \( u \) inside the module, it must be through local top. The intuition is that an ellipsis can be drawn around local top and local bottom that will contain all and only nodes in the module. Figure 4.12 (a) gives an example of a module and shows that at the outer level of the hierarchy the module can be considered as a single object. This concept of a module seems restricted for search purposes, since it rules out multiple inheritance across module boundaries as in Figure 4.12 (b).

Wille [Wil92] and Lehmann [Leh93] have done work on factorising hierarchies into dimensions. These dimensions may be structure hierarchies, type or other label hierarchies, attribute hierarchies and others. Figure 4.13 (b) shows a factorisation of a hierarchy. The term lattice on the left can be seen as the product of two simple trees with nodes 0, 1 and 2. Figure 4.13 (a) shows the recursive view of this product. Figure 4.13 (c) shows the possible variables and their assignments which reflects the term nature of the hierarchy. General term lattices \( TL_{k,d} \) can be recursively factored into simple trees with \( k – 1 \) branches. Each module is partitioned into \( k \) submodules. The module top \( (0) \) maps into each of the other modules \( 1 \ldots (k – 1) \) which pairwise have no relationships between them. It may be possible to classify each factor module into a hierarchy of module factors like that in Figure 4.14. Algorithms could then be written for each kind of factor module structure. This is useful if module patterns occur frequently. In chapter 5 we illustrated how an arbitrary poset can be mapped into a term lattice encoding, given this mapping it may be possible to structure the search in the same manner. Simple tree-like factors used in term lattices occur frequently. They represent mutually exclusive values that can fill a slot. A default search can be used for subhierarchies which do not fit any of the factor patterns.

Figure 4.16 gives a factored path algorithm for testing membership in term
function factored_path(u : O, M : PO) returns boolean;
begin
if null(M) then
    return FALSE;
elseif top(M[1]) ⊇ u then
    memb ← factored_path(u, M[1]); i = 1;
    ...;
elseif top(M[k − 1]) ⊇ u then
    memb ← factored_path(u, M[k − 1]); i = k − 1;
else
    memb ← factored_path(u, M[0]); i = 0;
if end then return memb;
else end ← TRUE; return u ⊇ top(M[i]);
end

Figure 4.16: An algorithm for searching for an node using factored path search

lattices T_{L_{k,d}} which makes use of the factor information. We assume that initially \top ⊇ u. Figure 4.15 (a) shows a term lattice T_{L_{3,3}} and the search for <2,2,2>. We use the notation M[2] to identify the elements in the outer module 1: <2,0,0>, <2,1,0>, ... <2,2,2>, that is, refers to all nodes which have a 1 in the first entry of the term code. As we recurse into the term lattice we select submodules within the current module, for example, in the search for <2,2,2>, if we select M[2] in the top-level M[2] module, then we restrict search to the module containing <2,2,0>, <2,2,1> and <2,2,2>. The module M may be empty null(M) which is a terminating module. For example, if we now select the submodule M[2], then there are no term codes which have <2,2,2> as a proper prefix. In the worst case every node of a module may be compared to u, but only one of the submodules is searched. The search order of a module is 1..(k − 1),0. If the top of module i : 1..(k − 1), is an ancestor of u, then no node in module j = 1..(k − 1) − {i} can be an ancestor of u, and paths from module 0 which lead to module i can be ignored. This is in contrast to depth-first search where the module structure is not used. The factored_path method effectively computes a tree covering of a factored graph such as in Figure 4.15.
4.3. FACTORED SEARCH

The end condition checks which if any of the factor nodes were ancestors of \( u \). If so, then either \( u \) is not in \( M \) or this factor is the terminating factor. We only test to see if the matching factor \( M[i] \) is subsumed by \( u \) and hence is \( u \) if there are no subfactors which contain ancestors of \( u \) (\( end \) is a global variable initially set to FALSE and set to TRUE when no ancestor of \( u \) is found in a module being searched). Consider the factored path search in the right hierarchy in Figure 4.10 for the object \( u = <2,2,2> \). Initially \( M = <> \). The search order is \( <1,0,0> \) (no match), \( <2,0,0> \) (match), \( M = <2> \), \( <2,1,0> \) (no match), \( <2,2,0> \) (match), \( M = <2,2> \), \( <2,2,1> \) (no match), \( <2,2,2> \) (match), \( M = <2,2,2> \). We now analyse factored path and analyse the complexity of the algorithm.

**Proposition 4.3.3** The number of objects compared in a factored path search of a term lattice \( T L_{k,d} \) of \( n \) nodes where \( n = k^d \) is \( O(k \log_k n) \).

**Proof:** On each call all of the \( k \) objects in module \( M \) will be tested for subsumption \( \sqsupseteq \) of the query \( u \). We ignore the test \( u \sqsupseteq \text{top}(M[i]) \), since this will only be done on the final node in the path. Each module \( M[i] \) contains a \( k \)th of the nodes in module \( M \). At most one of the submodules \( M[i] \) is recursively searched. Thus the recurrence relation for the number of nodes compared in factored path is \( \#\text{Nodes Compared}(n) = \#\text{Nodes Compared}(n/k) + k \). We prove that \( \#\text{Nodes Compared}(n) \leq c.k \log_k n \) where \( c > 0 \). We assume \( \#\text{Nodes Compared}(n/k) \leq c.k \log_k \frac{n}{k} \) and substitute this into the recurrence to yield

\[
\#\text{Nodes Compared}(n) \leq c.k(\log_k \frac{n}{k}) + k \\
= c.k((\log_k n) - 1) + k \\
= c.k \log_k n - c.k + k \\
\leq c.k \log_k n \text{ for } c \geq 1
\]

square box

The factored path algorithm improves on the depth-first member path algorithm in terms of nodes visited. In the simple algorithm member path this would also translate into a reduction in the objects compared. As noted before member path
can be improved by propagating non-ancestor information. The overhead involved will be significant, whereas the \textit{factored path} avoids most of this overhead at runtime by compiling this information into the program.

### 4.3.1 Experiments

In this section, we give the results of experiments of the intersection search method for finding descendants of an object used in the box intersection example in Figure 4.3. We tested the intersection algorithm on databases of chess patterns used in the Morph adaptive pattern-oriented chess learning system\cite{LS91} which are described in section 2.9.5 in chapter 2.

In the experiment shown in Table 4.2 we used the same objects in the files “A without”, “A with basis”, “B without”, and “B with basis” as queries (rather than assertions) on the databases constructed out of these files. This was used as a test of the pruning capabilities of intersection search, which requires some descendant of all the ancestors of the query to exist. In this case a descendant of all the ancestors of the query do exist (at least itself). Figure 1.6 in chapter 2 shows such a query on the database “A with basis”. The graphs shown are the graphs compared to the query (circled) during normal topological search, that is, the ancestors and twigs of the ancestors. The graphs that were compared using encoded search have dots (matched) or diamonds (did not match) next to them. The graphs with diamonds correspond to the twigs of the ancestors of the query. Encoding pruned 18 graphs (twigs) in this case. Objects such as \texttt{[Piece(Rank=>7)]} were not compared because none of the descendants of the query mention a piece on rank 7. Similarly there is no mention of white knights attacking black knights, so \texttt{[WhiteKnight]->(Direct)->[BlackKnight]} is not compared. \texttt{[Piece(Rank=>5)]} is compared even though it is not in the query, because it is mentioned in a descendant of the query.

The four “Encoded” rows indicate the improvement over normal topological search. Specifically the “\% of DB” entries indicate the decrease in the percentage of the database compared to find an object in the database. Encoded intersection search reduced the comparisons by over 50 \% and over 25 \% after adding the basis.
The “Time” column gives the minutes:seconds taken to classify the whole database onto itself. The slowness of the encoded search can be attributed to the large codes generated by the naive “compact” encoding method we used which is given in [AKBLN89] and discussed in section 5.2.4 in chapter 5. The compact encoding method is based on defining an object in terms of its children, and so is influenced by the number of join-irreducibles (see definition 2.2.1 in chapter 2) in the hierarchy. The number of join-irreducibles is at least as large as the number of coatoms. In “B without” for example there are 2958 coatoms out of a total of 3104 graphs. Chapter 5 discusses new methods of encoding which dramatically reduce the code lengths. We expect the speed of the encoded search to improve by using these more sophisticated encoding methods.

Encoded search does fewer comparisons than depth-first search, even though this is the best case for depth-first, since only a single path through the hierarchy needs to be taken to the exact graph equivalent to the query.
4.4 Summary

The algorithms in this chapter illustrate there are more refinements which can be made to the general search algorithms in chapter 2. The algorithms make more use of transitive information or structural information in hierarchies. The new ancestors algorithm \textit{ancestors\_encoded} uses the information that an object $v$ cannot be an ancestor of the query $u$, unless $v$ has descendants in common with the descendants of all the known ancestors of $u$, given there is some common descendant of all the ancestors of $u$. If there is no common descendant of all the ancestors of $u$, then the algorithm returns a subset of the ancestors of $u$. The \textit{descendants\_unification} algorithm uses unification of objects to determine at the earliest stage whether an object or its specialisations can contain specialisations of the query. The \textit{factored\_path} uses the information that hierarchies such as term lattices can be factorised into regular structures. Programs can be written which exploit this regularity for efficient search.
Chapter 5

Implementing Order and Lattice Operations

... when controversies arise, it will not be a work of learned disputation between two philosophers, but between two computists. It will be enough for them to take pen in hand, sit at the abacus, and say to each other, as friends: 'Let us calculate!'
- Leibniz

5.1 Applications of Lattice Encoding Techniques

Operations on complex object hierarchies can be efficiently implemented by mapping them to operations on simple (code) object hierarchies. Operations supported include subsumption (reachability), least upper bound (antiunification), and greatest lower bound (unification). In chapter 4, we showed how hierarchies of codes can be used to prune comparison of objects in complex object hierarchies. In this chapter, we examine how to compute simple codes for a given complex object database or order. In a number of interesting classes of orders, the term encoding method which we are developing produces codes more compact than previous encoding methods.

In chapter 4 we showed how lattice operations could be used for searching for descendants of objects in hierarchies. We used the box world problem given by Stonebraker [Sto89]. We showed how boxes could be pruned from the search for boxes inside box u in Figure 4.1(a) using box overlap operations. We showed how
rather than box overlap operations, intersection of the descendant sets in the hierarchy Figure 4.1(b) could be used. We indicated it is not an advantage to trade box containment operations for box intersection operations or descendant set intersection operations in this example, since these alternatives are more complex than containment operations on boxes. However, it may be more advantageous to trade box containment operations for simple code operations. Figure 4.3(b) shows how the set of boxes in Figure 4.3(a) can be mapped to a hierarchy of codes. The hierarchy of codes reflect the shape of the hierarchy of boxes. Here the codes or terms are open lists of integers, where an underscore represents a variable. Trailing at the end of each list is a (infinite length) list of variables. A term lattice is defined in chapter 2, definition 2.9.3. For example, some examples of term subsumption $\sqsubseteq$ include

1. $<> \sqsubseteq <1>$
2. $<\_1> \sqsubseteq <1,1>$
3. counter example $<\_1,2> \not\sqsubseteq <1,1>$
4. counter example $<1,1> \not\sqsubseteq <2,1>$

Intersection or term unification $\sqcap$ examples include

1. $<1,\_3> \sqcap <\_2,\_4> = <1,2,3,4>$
2. $<1,3> \sqcap <\_2> = \bot.$

In Figure 4.3(a), the search for boxes inside $u$ can be pruned using the hierarchy of codes in Figure 4.3(b). When box $a$ is compared, the intersection or approximate code of the query is computed as $<\_>$. The box $b$ is compared because its code $<1>$ unifies with $<\_>$, $<1> \sqcap <\_> = <1>$. The box $c$ is compared because its code $<\_1>$ unifies with $<1>$, $<\_1> \sqcap <1> = <1,1>$. The box $d$ is not compared because its code $<1,2>$ does not unify with $<1,1>$, $<1,2> \sqcap <1,1> = \bot$. Codes not unifying corresponds to boxes not having subboxes in common. Similarly, $e$, $j$, $k$, and $l$ are not compared, since their codes $<1,3>$, $<2,1>$, $<3,1>$, $<4,1>$ do not unify with $<1,1>$. Boxes $f$ and $g$ are compared, because their codes $<1,\_1>$ and $<1,1,1>$ unify with $<1,1>$. The code $<1,1>$ is an approximate encoding of $u$. 
In the second phase, children of the query box $u$ must be in the intersection of parents of $u$, $a$ and $b$. Thus children codes must be subsumed by $<1,1>$. The only descendants of $a$ and $b$ whose codes are subsumed are $f$, $g$ and $i$. Each of these is compared and $i$ is found to be the only child of $u$.

### 5.1.1 Using Codes for Indexing Very Complex Objects

These methods are suitable for arbitrarily complex objects. The codes produced by the more compact methods discussed later are dependent on the characteristics of the hierarchy (the order relationships), rather than directly dependent on the objects being indexed. For example, complex objects such as images with fractal geometries can be index based on the ordering relationship recursive subpattern and arranged into a hierarchy such as Figure 1.3(a) in chapter 1. In Figure 1.3, the hierarchy of images (a) can be mapped to the hierarchy of codes (b), that is, “Blank” maps to $<>$, “Sierpinski Triangle” to $<1>$, “Diamond” to $<\_1>$, “Sparkling Diamond” to $<\_1,1>$ and “Sierpinski Triangle and Sparkling Diamond ” to $<1,1,1>$. Notice that the number of positions used in the term “Sierpinski Triangle and Sparkling Diamond” corresponds to the number of descriptors in the name. Suppose the query image has a Sierpinski Triangle and a Diamond, then the code of the query would be $<1,1>$ and any image whose code does not unify with $<1,1>$ can be ignored, since it cannot occur in superimages of the query if there are superimages in the database.

In chapter 4 and [ELR94], we have also shown that these methods using codes can be used for indexing databases of complex objects such as program specification libraries and chemical libraries.

### 5.1.2 Using Codes to Implement Type Operations

Type (class) hierarchies in program languages are encoded for supporting operations on types such as subtype, type unification, and type anti-unification [AKBLN89, LM92, Cas93]. These techniques are used as the basis of primitive operations on classes in object-oriented languages: CLOS [Ste90], LOOPS [Ste90], LIFE [AKP93], LAURE [Cas93].

Operations on hierarchies with no explicit objects can also be implemented using
simple object hierarchies. For example, in programming languages it is common to
declare one type as a subtype of another type, without explicitly giving a definition
that differentiates the two types. Figure 5.1 shows a hierarchy of types on the left
and a possible term encoding on the right. The type **Cat** is a subtype of **PetCat**.
The code for **Cat**, <1>, is a subcode of the code for **PetCat**, <1,1>. **Pet** and **Cat**
have a least common subtype (unifier) of **PetCat**. The code for **PetCat**, <1,1>-corresponds to the unification of codes of **Cat** and **Pet**, <1> and <1>. The types
**Cat** and **Dog** do not unify, since <1> ∩ <2> = ⊥.

Hierarchical encoding methods have been practically applied to a wide variety
of areas including object databases and type hierarchies [GM93, Mel90, Mas88,
GNT87a]; compactly representing the transitive closure of relations in databases
[ABJ89, Jag89]; representing and supporting operations on virtual time and causality
in distributed systems [Mat89]; and theoretical work on representing orders
[Jr.75] [GNT87b, GNT90].

In section 5.2 we survey previous encoding methods. In section 5.3 we introduce
a method for supporting greatest lower bound operations using a decision tree over
kinds of codes produced by previous methods. In section 5.4 we define terms used
in later sections. In section 5.5 we describe a new encoding method called term
encoding. In sections 5.6 we analyse the complexity of the term encoding method
and compare it to previous methods.

We emphasis here that we do not describe the complexity of the algorithms for
producing the encodings, but only the complexity of the resulting encoding space.
5.2 Previous Encoding Methods

Encoding methods map one partially ordered set into another partially ordered set of codes (usually a lattice). To discuss the kinds of encodings and the codes, we first define some of the operations such as greatest lower bound, least upper bound that an encoding must support.

**Definition 5.2.1** The following terms describe sets in a poset \( \mathcal{P} = (P, \sqsubseteq) \). We assume a subposet \( \mathcal{S} = (S, \sqsubseteq) \), such that \( S \subseteq P \).

- **UpperBounds** \((u, v, S) = (\text{Ancestors}_=(u, S) \cap \text{Ancestors}_=(v, S))\)
- **LowerBounds** \((u, v, S) = (\text{Descendants}_=(u, S) \cap \text{Descendants}_=(v, S))\)
- **UpperBoundsBase** \((u, v, S) = \text{Base}(\text{UpperBounds}(u, v, S))\)
- **LowerBoundsCrown** \((u, v, S) = \text{Crown}(\text{LowerBounds}(u, v, S))\)
- **Least Upper Bound:** If \( \text{UpperBoundsBase}(u, v, S) = \{w\} \), for some \( w \), then \( \text{LUB}(u, v, S) \) is defined and \( \text{LUB}(u, v, S) = w \). \( \text{LUB}(u, v, S) \) is also denoted as \( u \sqcup_{\mathcal{S}} v \).
- **Greatest Lower Bound:** If \( \text{LowerBoundsCrown}(u, v, S) = \{w\} \), for some \( w \), then \( \text{GLB}(u, v, S) \) is defined and \( \text{GLB}(u, v, S) = w \). \( \text{GLB}(u, v, S) \) is also denoted as \( u \sqcap_{\mathcal{S}} v \).
- A non-empty ordered set \( \mathcal{S} \) is called a lattice if \( u \sqcup_{\mathcal{S}} v \) and \( u \sqcap_{\mathcal{S}} v \) exist for all \( u, v \in S \).
- A finite lattice \( \mathcal{S} \) has one minimal element and one maximal element denoted respectively \( \top_{\mathcal{S}} \) (top) and \( \bot_{\mathcal{S}} \) (bottom).

For example, in Figure 4.3 the top element is \( a \) and the bottom element is \( \bot \). The parents of \( \bot \) are the boxes which do not contain any other boxes: \( d, h, i, j, k \) and \( m \). The **UpperBounds** set of \( d \) and \( g \) is \( \{a, b\} \) and the **LowerBounds** set is \( \{\bot\} \), corresponding to the common ancestors and common descendants respectively. The **UpperBounds** set of \( f \) and \( g \) is \( \{a, b, c\} \) and the **LowerBounds** set is \( \{i, j, k\} \).
The $UpperBoundsBase$ of $f$ and $g$ is $\{b, c\}$ and the $LowerBoundsCrown$ is $\{i\}$. In the boxes example, we let $B$ stand for the set of boxes illustrated. In the example, $d \sqcup g = b$, since $UpperBoundsBase(d, g, B) = \{b\}$, and $d \sqcap g = \bot$, since $LowerBoundsCrown(d, g, B) = \{\bot\}$. The boxes hierarchy is not a lattice, since neither $f \sqcup g$ nor $b \sqcap c$ is defined, because $UpperBoundsBase(f, g, B) = \{a, b\}$ and $LowerBoundsCrown(a, b, B) = \{f, g\}$.

As Figure 5.2 suggests, the encoding process involves mapping elements $u$ and $v$ of some poset $P$ to elements $\tau(u)$ and $\tau(v)$ in a poset of codes $G$ to support some operation, in this case GLB. In the complex object database search problem illustrated in Figure 4.3, the boxes $a$ and $b$ are mapped to codes $<1>$ and $<\bot1>$. The box intersection operation in the boxes hierarchy is replaced with a term unification operation in the term hierarchy. Thus instead of doing a box intersection operation on $a$ and $b$, a term unification operation is performed on $<1>$ and $<\bot1>$.

Encodings can be classified based on the operations they support. Two kinds of encodings we examine in this chapter are subsumption encodings and term encodings. We define each in turn giving examples.

**Definition 5.2.2** Subsumption Encoding [Cas93]

*Given an object poset $\mathcal{P} = (P, \sqsupseteq_P)$ and a code poset $\mathcal{G} = (G, \sqsupseteq_G)$, an encoding $\tau : P \mapsto G$ is a subsumption encoding of $P$ if, for all $u$ and $v$ in $P$, $u \sqsupseteq_P v$ if and only if $\tau(u) \sqsupseteq_G \tau(v)$.***

We will omit the subscript on $\sqsupseteq$ when it is clear from the context which poset is being discussed. The encoding of the boxes hierarchy with terms in Figure 4.3 is a subsumption encoding since for every box $u$ and $v$, box $u$ contains box $v$ if and only if the term $\tau(u)$ subsumes $\tau(v)$. For example, $b$ contains $h$ and $<1> \sqsupseteq <1,3,1>$. 
All of the encoding methods we will examine are subsumption encodings. We now define a term encoding which supports a stronger relationship GLB.

**Definition 5.2.3** Term Encoding [DF93]

Given an object poset $\mathcal{P} = (P, \sqsubseteq_P)$ and a code poset $\mathcal{G} = (G, \sqsubseteq_G)$, an encoding $\tau : P \rightarrow G$ is a term encoding of $P$ if, for all $u, v, u_1, \ldots, u_n$ in $P$

1. if $\tau(u) = \tau(v)$ then $u = v$

2. $\tau(\bot_P) = \bot_G$

3. $u \sqsubseteq_P \prod_P (u_1, \ldots, u_n)$ iff $\tau(u) \sqsubseteq_G \prod_G (\tau(u_1), \ldots, \tau(u_n))$

where $\prod_P$ and $\prod_G$ represent the greatest lower bound operation applied in the object poset $P$ and term poset $G$, respectively.

The term encoding of boxes in Figure 4.3 is a kind of term encoding. The boxes $b$ and $c$ have common descendants (boxes in their overlap) $f, g$ and $i$. A poset can be mapped into a lattice encoding by treating each object in the hierarchy as a singleton set containing that object, then inducing an order through union of sets representing sets of descendants and ordering by subset[AKBLN89]. The code $\tau(\text{Descendants}_{=}(b, B)) \sqcap \tau(\text{Descendants}_{=}(c, B)) = <1> \sqcap <2,1> = <1,1>$ where $\tau^{-1}(<1,1>) = \{f, g, i, \bot\}$. For this example $\tau^{-1}(\tau)$ returns the set of objects whose code is subsumed by $\tau$. Boxes $d$ and $e$ do not have any boxes in their overlap (other than the empty box $\bot$) $(\text{Descendants}_{=}(d, B) \cap \text{Descendants}_{=}(e, B) = \{\bot\})$. Also $\tau(d) \sqcap \tau(e) = <1,2> \sqcap <1,3> = \bot$, where $\tau(\{\bot\}) = \bot$. The codes $\tau(\text{Descendants}_{=}(k, B))$ and $\tau(\text{Descendants}_{=}(l, B))$ do not unify, hence the boxes $k$ and $l$ do not have any boxes in their overlap. Only some encodings we examine are term encodings. Any term encoding is also a subsumption encoding, though not every subsumption encoding is a term encoding.

When designing encoding methods the first decision is to choose a code type. Here we consider some code types as diverse as descendant sets, products of primes, bit-vectors, lists of integers (terms), and lists of integers (time vectors). The selection of a code type is determined by a number of factors including: operations to be supported; compactness of codes; and efficiency of operations. The most popular
method of encoding type and object classes in programming languages is bit-vector encoding. We will show that term encoding using lists of integers is more compact than bit-vector encoding for interesting classes of hierarchies. We will also show that time vector (also list of integers) codes are more compact for hierarchies than term codes for deep narrow hierarchies and suggest research should be directed towards the integration of these two orthogonal views of lists of integers. Further, more expressive term languages such as conceptual graphs (with abstraction) could be used for compact representation of complex orders.

Figure 5.3 shows some of the possible representations of an ordering given in Figure 5.3(a):

(b) node $u$ is represented as the descendants $\preceq u$ of $u$;

(c) node $u$ is represented as a term;

(d) node $u$ is a vector time stamp (in section 5.2.9, we define time vectors) where the width of the code corresponds to the number of processes in a distributive system;
(e) the transitive closure of the relation in (a);

(f) the boolean vector representation of (a) in a boolean space $B_6$ of 6 bits;

(g) the representation of (a) in a boolean space of 3 bits.

An aim of hierarchical encoding methods is to find a compact encoding of a hierarchy with efficient code operations. In this example, (c), (d) and (g) are competitive for subsumption. Vector Time encoding is not a term encoding, since there is no finite concept of bottom $\bot$.

Figure 5.3(g)(i) gives a recursive picture of a boolean space $B_3$. The arc between the upper module 1 and the lower module 0 indicates that there are arcs between each of the corresponding values, 11, 10, 01, 00 in each of the modules. This can be seen in Figure 5.3(g)(ii) where the arcs have been made explicit.

Since bit-vector encoding is the benchmark for encoding because of its common use in type encoding in program languages, we define a bit-vector and some ordering relationships over bit-vectors.

**Definition 5.2.4** A bit-vector $t$ is a sequence $t_1 \ldots t_n$ where each $t_i$ is either 0 or 1. Given two bit-vectors $s = s_1 \ldots s_n$ and $t = t_1 \ldots t_m$ with $p = \max(n, m)$:

- **GLB** $s \land t = u = u_1 \ldots u_p$, where $\forall i : 1..p \cdot u_i = 1$ if $s_i = t_i$ otherwise $u_i = 0$.

- **LUB** $s \lor t = u = u_1 \ldots u_p$, where $\forall i : 1..p \cdot u_i = 1$ if $(s_i = 1) \lor (t_i = 1)$ otherwise $u_i = 0$.

**Subsumption** $s \sqsupseteq t$ if and only if $(s \land t) = t$.

The operation $\land$ is referred to as **bit-vector AND**. The operation $\lor$ is referred to as **bit-vector OR**. The test $a \sqsupseteq d$ is true in each of the encodings in Figure 5.3:

(a) since $d$ occurs in a walk of the descendants $s_m$ of $a$;

(b) since descendants $s_m$ of $a$ include the descendants $s_m$ of $d$: $\{a, b, c, d, e, \bot\} \supset \{d, \bot\}$;

(c) since term $<> \sqsupseteq <1,2>$;
(d) since time $[0,0]$ occurs before $[2,1]$.  

(e) since bit-vector $11111 \land 000101 = 000101$.  

(g) since bit-vector $111 \land 100 = 100$.  

The codes which we will concentrate on are term codes (c) and bit-vector codes (g). The bit-vector operation will involve two machine operations: a bit-wise AND and a bit-wise equality check. The term code operation will involve, for each slot in the code, simple unification. In the example above, bit-vector encoding is faster and more compact than term encoding. Bit-vector codes which are longer than a machine word are usually implemented as a list of machine words and corresponding operations are implemented by applying machine word operations to each word in the list. We will show that for large hierarchies where term codes are more compact, term encoding is more efficient than bit-vector encoding in real machine operations performed.

Consider the computation of $b \cap c$ in each of the encodings in Figure 5.3:  

(a) topologically walk the descendants of $b$ and $c$ to find the first common descendants  

(b) $\{b, d, e, \bot\} \cap \{c, e, \bot\} = \{e, \bot\}$  

(c) terms $<1> \cap <\bot> = <1,1>$  

(d) time stamps $[1,0] \cap [0,1] = [1,2]$  

(e) bit-vector $010111 \land 001011 = 000011$  

(g) bit-vector $101 \land 011 = 001$.  

In each case we produce the code of $e$. Here the bit-vector operation (g) requires a single bit-vector operation, but we will see that for supporting GLB operations, the decoding operation (determining if $\bot$ usually) is a dominant cost. The term code operation (c) requires operations for each slot subterm. If the subterms are different atoms, then the operation fails, otherwise if a subterm is an atom, the atom is returned or if both are variable, a variable is returned. This subterm unification operation could be implemented in 3-4 machine operations. Alternatively the slot
5.2. PREVIOUS ENCODING METHODS

comparison operation could be implemented in a special purpose machine operation. In either case it would be more expensive than the bit-vector operation, but we will show that because of the vector length, term encoding executes potentially less instructions in a GLB operation.

Encoding methods are usually based on some form of inheritance. We use the term gene to describe a basic unit of inheritance. A code consists of a set of genes. In the case of bit-vectors the genes are single bits. In a term code each atom/integer in a slot represents a gene. Since the codes are usually implemented in machines using bit-vectors, an encoding can be characterised by the size of the bit-vectors used in the final implementation. Alternatively, we can characterise encodings based on the number of genes used and then discuss the size of those genes. For example, in Figure 5.3 the term code (c) uses 2 genes and 3 bits, since the largest code (τ(d)) <1,2> (two genes) can be represented as the bit-vector 101. The bit-vector of τ(a) in Figure 5.3(g) has 3 genes and uses 3 bits 111. The time vector encoding uses 2 genes and 4 bits 11 01.

Bit-vector encodings have been targeted recently because of their support by computers at the machine level. However, we will show that other coding techniques such as term codes may be more compact and their implementation more efficient than bit-vectors and bit-wise operations. In the following sections we will discuss previous encoding methods in the order they were developed, starting in the year 1679.

5.2.1 Leibniz’s Universal Characteristic

Leibniz’s Universal Characteristic [(1603) (1679)] was the first mechanical solution to type subsumption or reachability in a hierarchy. Leibniz was influenced by Ramon Llull’s Ars Magna [Llu15, Sow84] (Thirteenth Century) which has a system of disks inscribed with primitive concepts, which can be combined in various ways by rotating the disks to produce compound concepts. Leibniz represented primitive concepts by prime numbers and compound concepts by product of primes ([Sow84], p. 14).

Leibniz [(1603] envisioned a universal dictionary for mapping concepts to numbers and a calculus of reasoning that would automate the syllogism. “All u is v” is
tested by checking number(u) is divisible by number(v). Testing “All u is v” is a kind of reachability or subsumption query. Consider the hierarchy in Figure 5.4 (a). The primitive concepts are Pet, Cat, Dog, and Bird. They are allocated the first 4 prime numbers 2, 3, 5, and 7 respectively. Every PetCat is a Cat since 6 is divisible by 3. But, every PetCat is not a Dog since 6 is not divisible by 5. To simplify the computations Leibniz invented a calculating machine that could do multiplication and division.

Leibniz’s method supports a least upper bound operation (type anti-unification) as well as subsumption. The least common supertype of PetCat and PetDog is computed by decoding the greatest common divisor of 6 and 10 which is 2. The number 2 corresponds to the type Pet. Leibniz’s Universal Characteristic is a calculus for subsumption and anti-unification. The least common multiple operation in the Universal Characteristic does not support type unification (a greatest lower bound operation) since it allows arbitrary combination of types. For example, in Figure 5.4 (a) Cat ∩ Dog would equal CatDog since 3*5 = 15. In this example, it is necessary to ensure no code number is divisible by 15, 21 or 35. The shaded regions in Figures 5.4 (c), (d) and (e) are the absurd space. Each represents impossible combinations of concepts and codes. The absurd space corresponds to the absurd type in Figure 5.4 (a). Leibniz’s method is a subsumption encoding, but not a term
encoding.

The above characterisation of Leibniz's *Universal Characteristic* is a (distributive) lattice ([Sow84], p.383):

- Each primitive atom is represented as a single prime number.
- \( u \sqsupseteq v \iff \text{divides}(\tau(v), \tau(u)) \).
- \( \tau(u \sqcup v) = \text{GCD}(\tau(u), \tau(v)) \) (Greatest Common Divisor).
- \( \tau(u \sqcap v) = \text{LCM}(\tau(u), \tau(v)) \) (Least Common Multiple).
- Top \( \top \) is 1.
- The bottom \( \bot \) is the product of the first \( p \) primes, where \( p \) is the total number of primitive atoms.

The above is actually a simplification of Leibniz's method, since Leibniz's method also encoded multi-sets or plurals. For example, Leibniz would represent the plural of two cats as \( 4 = 2 \times 2 \), and a stone and two birds as \( 539 = 11 \times 7 \times 7 \) (given a the primitive stone is assigned the prime 11).

The compactness of the codes can be improved. The code of \( \bot \) is the product of the first \( n \) primes. For example, in Figure 5.4 (c) PetCatDogBird (\( \bot \)) is represented by \( 210 = 2 \times 3 \times 5 \times 7 \). The number 210 can be represented using 8 bits, 11010010, however, as Figure 5.4 (e) illustrates, the type hierarchy in Figure 5.4 (a) can be represented using 4 bits, by mapping it directly into the boolean space \( B_4 \). Each bit position corresponds to a primitive concept. The bit-vector encoding is a subsumption encoding since \( u \sqsupseteq v \) iff \( \tau(u) \lor \tau(v) = \tau(v) \). For example, Pet \( \sqsubseteq \) PetCat, since 0010 \( \lor \) 1010 = 1010 = \( \tau(\text{PetCat}) \). Further \( \tau(u \sqcup v) = \tau(u) \land \tau(v) \). For example, \( \tau(\text{PetCat} \sqcup \text{PetDog}) = \tau(\text{Pet}) = 1010 \land 1001 = 1000 \). However, this bit-vector encoding is not a term encoding since \( \tau(\text{Cat} \sqcap \text{Dog}) = \tau(\text{Absurd}) = 1111 \neq \tau(\text{Cat}) \land \tau(\text{Dog}) = 0010 \lor 0001 = 0010 \).

Table 5.1 shows the number of bits needed to represent the product of the first \( n \)th primes up to \( n = 11 \). The \( n \)th prime is approximated by \( n \ln n \) [CLR90]. The product of the first \( n \) primes is approximated by \( \prod_{i=1}^{n} i \ln i \). The number of bits
Table 5.1: The first \( n \) primes (up to 11) and the bits need to encode them

<table>
<thead>
<tr>
<th>( n )</th>
<th>nth prime</th>
<th>( \pi ) product of first ( n ) primes</th>
<th>bits to encode ( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>210</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>2310</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>30030</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>510510</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td>9699690</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>223092800</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
<td>6469691000</td>
<td>33</td>
</tr>
</tbody>
</table>

needed to encode using Leibniz’s method is thus approximated by

\[
\lceil \log_2 \left( \prod_{i=1}^{n} i \ln i \right) \rceil
\]

which is greater than \( n \ln n \) number of bits and less than \( n^2 \log_2 n \). Although the method does not produce very compact codes, it was a remarkable achievement for its time. We now consider more compact bit-vector based encoding methods.

### 5.2.2 Transitive Closure Boolean Matrix

Aït-Kaci, Boyer, Lincoln and Nasr [AKBLN89] gave an algorithm for computing the transitive closure of a partial order which performed the following encoding. Given the sequence of objects in the hierarchy in increasing topological (level) order starting with \( \bot \), the code for the \( p \)'th entry \( u \) is

\[
\tau(u) \leftarrow 2^p \lor \bigvee_{v \in \text{Children}(u, p)} \tau(v)
\]

where \( \lor \) is bit-vector OR. Each object \( u \) is defined as the inheritance of the genes of its children plus a unique gene (the \( p \)th bit).

Figure 5.3 (e) gives the boolean transitive closure matrix of the hierarchy in Figure 5.3 (a) using this method. The intersection of descendants of two elements corresponds to ANDing the rows of the elements in the boolean transitive closure matrix. The empty set corresponds to the bottom element \( \bot \) which has a code 000001 in this example. Also, for example,
\[
d \cap c = \tau^{-1}(\tau(d) \cap \tau(c)) \\
= \tau^{-1}(000101 \land 001011) \\
= \tau^{-1}(000001) \\
= \bot.
\]

The last row and column could be removed since the bottom element is redundant if coded as 00000. The subsumption test \( u \sqsupseteq v \) can be evaluated by checking if the \( u \)th bit in the \( v \)th column is set to 1. For example, \( b \sqsupseteq e \) since the \( b \)th bit in column \( d \) is set to 1. This is possible in a transitive closure matrix, since each object is assigned a unique bit.

If there are \( n \) elements in \( P \), then each element is allocated its own gene, hence the code length is \( n \). The transitive closure matrix is a term encoding and hence a subsumption encoding. If the poset \( P \) has size \( N \), then the size of each code in a transitive closure matrix is \( N \), and the poset is coded with \( N^2 \) bits. Hence, a transitive closure matrix is not a realistic method for large posets, such as large object-databases or class hierarchies.

### 5.2.3 Discussion of Some Poset Characteristics

Different coding methods are affected by different poset characteristics. In chapter 2 we defined some characteristics such as width, depth, meet-irreducibles and join-irreducibles which can be used to measure the “size” of a poset. For example, in Figure 4.3 the width of the boxes hierarchy is 7, since the maximal cochain is \{d, e, f, g, j, k, l\}. The depth of the hierarchy is 3, since a maximal chain is \{a, b, e, h\}. The \textit{MeetIrreducibles} of the hierarchy are \{b, c, d, e, h, j, k, l, m\}. The \textit{JoinIrreducibles} are \{d, e, f, g, j, k, l, h, m\}. In Figure 1.3, the width is 2, where of the maximal cochains are \{Sierpinski Triangle, Diamond\} and \{Sierpinski Triangle, Sparkling Diamond\}. The depth is 3, where the maximal chain is \{Blank, Diamond, Sparkling Diamond, Triangle and Sparkling Diamond\}. The \textit{MeetIrreducibles} are \{Sierpinski Triangle, Diamond, Sparkling Diamond\}. The \textit{JoinIrreducibles} are \{Sierpinski Triangle, Diamond, Sparkling Diamond\}.

The \textit{MeetIrreducibles} correspond to the genes in a hierarchy. In an image
hierarchy *MeetIrreducibles* correspond to the atomic subimages or attributes of an image: Triangle (atomic subimage), Diamond (atomic subimage) or Sparkling (attribute of an image). The *JoinIrreducibles* correspond to genes which only occur once. In an image hierarchy, a join-irreducible is a subimage which occurs in only one image or is a complete image. A library of many complete images can be constructed from a few subimage components and attributes. Hence for image libraries #*MeetIrreducibles* is usually much smaller than #*JoinIrreducibles*. A similar argument can be given for knowledge bases. Many different facts, situations, and rules (join-irreducibles) can be constructed from a few concepts, attributes, relations, and logical connectives (meet-irreducibles).

Since *depth* corresponds to chains of inheritance, *depth* is a measure of the inherited information in an object. The *width* of a hierarchy corresponds to the largest cochain in the hierarchy, that is, the largest set of unrelated objects. The largest cochain in many hierarchies is usually the collection of instances or instance classes at the bottom of the hierarchy. So the *width* is a measure of the diversity of objects in the hierarchy.

### 5.2.4 Compact Encoding

Compact encoding [AKBLN89] is based on transitive closure encoding, but recognises that allocating a separate gene (or bit) to each object in the poset *P* is too conservative. Compact encoding tries to allocate genes to join-irreducibles only. Encodings are usually constructed in two phases. The first phase constructs an initial encoding and the second phase removes conflicts introduced in the first phase. If an element *u*’s descendants are a subset of the descendants of an incomparable element *v* there is potential for conflict, since *u* will inherit a subset of genes and hence the code *τ(u)* will subsume *τ(v)*. Conflicts are usually repaired by adding extra genes to each of the conflicting codes, so they no longer conflict.

The first phase of compact encoding allocates genes to the meet-irreducibles and computes the codes of non-meet-irreducibles. The meet-irreducibles are topologically ordered starting with ⊥ and is hence known as a bottom-up encoding, since codes of *u* are computed from *u*’s children codes. The *i*th meet-irreducible *u* is
encoded using the rule
\[ \tau(u) \leftarrow 2^i \lor \tau(\text{Child}(u, S)) \]
The non-meet-irreducibles are also processed at the same time in topological order from \( \perp \), using the rule
\[ \tau(u) \leftarrow \bigvee_{v \in \text{Children}(u, S)} \tau(v) \]

This method produces codes which tend to be related to \(#JoinIrreducibles(S)\). In databases we have experimented with, we have found that the code length is closely related to or exactly the same as \(#JoinIrreducibles(S)\) (see Table 5.6). In inheritance hierarchies and object databases, \(#MeetIrreducibles\) is usually much smaller than \(#JoinIrreducibles(S)\). This is because meet-irreducibles closely correspond to genes or pieces of information inherited, and join-irreducibles closely correspond to instances or instance classes. There are usually more instances and instance classes than genes or methods inherited.

When \(#MeetIrreducibles\) is smaller than \(#JoinIrreducibles\), we propose to encode top-down rather than bottom-up. Top-down encoding makes sense for inheritance hierarchies, since this is also the direction of inheritance in the objects. In the first phase of top-down compact encoding, the join-irreducibles are encoded in topological order starting from \( \top \) using the following rule for the \( i \)th join-irreducible \( u \)
\[ \tau(u) \leftarrow 2^i \lor \tau(\text{Parent}(u, S)) \]
Non-join-irreducibles are encoded in top-down topological order using the following rule for encoding the non-irreducible \( u \)
\[ \tau(u) \leftarrow \bigvee_{v \in \text{Parents}(u, S)} \tau(v) \]
A conflict could occur if an element \( u \)'s ancestors are a subset of the ancestors of \( v \), so that \( u \) would inherit a subset of the genes of \( v \) and hence \( \tau(u) \) would subsume \( \tau(v) \).

Bottom-up compact encoding is dependent on the number of join-irreducibles, whereas top-down encoding is dependent on the number of meet-irreducibles. In binary decision trees \( DT_{2,d} \) the leaves are join-irreducibles, whereas every node ex-
Figure 5.5: A compact encoding of the decision tree $DT_{2,3}$

cept the root is a meet-irreducible, hence there are roughly twice as many meet-
irreducibles as join-irreducibles in a binary decision tree. Compact encoding bottom-
up is better than top-down for binary decision trees. In the example in Figure 5.5,
bottom-up compact encoding uses 8 bits, and top-down compact encoding uses 15
bits. Habib and Nourine [HN94] showed that a lattice $L$ can be encoded in
\[ \min(\#\text{MeetIrreducibles}(S), \#\text{JoinIrreducibles}(S)) \]
genesis. Though this result is for lattices it is also a useful indicator for posets, depending on how closely the
poset is related to lattice completion of the poset. The number of genes allocated
by compact encoding is usually equal to this minimum value reflecting the choice of
top-down or bottom-up encoding.

Bottom-up compact encoding is a term encoding (and hence a subsumption
encoding) since it does uniquely identify bottom and maintain the $\sqcap$ operation. For
example, in Figure 5.5

\[
d \sqcap c = \tau^{-1}(\tau(d) \sqcap \tau(c)) \\
= \tau^{-1}(11000000 \land 00001111) \\
= \tau^{-1}(00000000) \\
= \bot.
\]

This is because each leaf is at most one bit away from $\bot$ and hence the bottom
space can be uniquely identified as being all zeroes. However, compact bottom-up
encoding does not uniquely encode $\top$

\[
d \sqcup c = \tau^{-1}(\tau(d) \sqcup \tau(c)) \\
= \tau^{-1}(11000000 \lor 00001111) \\
= \tau^{-1}(11001111)
\]
Figure 5.6: A modulated encoding of the decision tree $DT_{2,3}$

$\neq T$.

### 5.2.5 Modulated Encoding

Aït-Kaci, Boyer, Lincoln and Nasr [AKBLN89] proposed a divide and conquer encoding method based on recursively breaking up the poset into small encodable modules which are then encoded using compact encoding. A module is a subposet of $P$ that shares a common module top and module bottom element and every element of the module is only related to elements outside the module (in $P - module$) transitively through the module top and bottom. Modules do not necessarily have to have a unique top and bottom, but can have a crown as top and a base as bottom. Figure 4.12 in chapter 4 shows that such a module can be treated as a single node in an outer hierarchy. Habib, Huchard and Spinrad [HHS95] have developed a linear time algorithm for decomposing hierarchies into modules.

The code of a module is known as the group code, and the code of an module element is known as a local code. Let $\sqsubseteq_m$ be the ordering on two modulated codes $X$ and $Y$ where $X_g$ is $X$’s group code and $X_l$ is $X$’s local code, and $Y_g$ is $Y$’s group code and $Y_l$ is $Y$’s local code. Let $A_g$ be the bitwise AND of $X_g$ and $Y_g$. Let $A_l$ be the bitwise AND of $X_l$ and $Y_l$. Within any group, we use $T_g$ (in [AKBLN89], Hassan Aït-Kaci et al. used $-1$) as a shorthand for the topmost element’s code.
\[ X \preceq_m Y \text{ iff } \begin{cases} X_g = Y_g, & \text{and } X_l \sqsubseteq Y_l; \\ X_g \sqsubset Y_g; \end{cases} \tag{5.1} \]

This definition says that \( X \) is greater than \( Y \) if and only if \( X \) and \( Y \) are in the same group, and \( X_l \sqsubseteq Y_l \) (using the previously defined \( \sqsubseteq \) on codes), or \( X \)'s group subsumes \( Y \)'s group.

Here the GLB of \( X \) and \( Y \) (with \( X = (X_g, X_l) \) and \( Y = (Y_g, Y_l) \), \( A_g = X_g \land Y_g \), and \( A_l = X_l \land Y_l \)) is:

\[
X \sqcap Y = \begin{cases} 
(A_g, A_l) & \text{if } X_g = Y_g; \\
(X_g, X_l) & \text{if } X_g \sqsubset Y_g; \\
(Y_g, Y_l) & \text{if } X_g \sqsubset Y_g; \\
(A_g, \top \land A_g) & \text{otherwise} 
\end{cases}
\]

The first possibility is that \( X \) and \( Y \) are in the same group, and so the result is that group’s code paired with the AND of their local codes. The second possibility is that \( X \)'s group is subsumed by \( Y \)'s. Thus the result is simply \( X \)'s original code. The third possibility is that \( Y \)'s group is subsumed by \( X \)'s. Then the result is simply \( Y \)'s original code. The last possibility is that neither group subsumes the other, and thus the result is the topmost element in the group which is the greatest lower bound of the two groups.

Figure 5.6 shows an example of a modulated encoding of a small decision tree. The tree can be broken into two modules \( \{b, d, e, h, i, j, k\} \) and \( \{c, f, g, l, m, n, o\} \), both these modules can then be broken up into further modules. The first can be broken down recursively into two modules \( \{b, \{d, \{h, i\}\}, \{e, \{j, k\}\}\} \). Using the modulated code \( h \sqcap l = \tau^{-1}((10, (10, 10)) \sqcap (01, (10, 10))) = \tau^{-1}(00, (11, 11)) = \bot \).

Here the local codes 10 and 01 AND to get 00, since the local codes are incomparable, the fourth case in the \( \sqcap \) rule applies, hence the top local code (11, 11) of the group 00 is given as the local code. Notice the local code is itself composed of a group and local code.

The definition of a module is an unnecessarily restricted concept. Caseau [Cas93] has found that instances of this module concept occur infrequently in real posets and that the modulated encoding method usually degenerates to the compact encoding
method described earlier. The reason is that it only takes one element in the interior of the module to be directly attached to an element outside the module to cause the module to grow to include that outside element (and usually much of the structure around it). The new elements included may further require more structure being included in the module such that the expands to all or much of the hierarchy. The module concept corresponds to a separate subdomain which has no cross-links with other domains, other than super and sub-domains. If a cross-link is found between two different domains, they are no longer modulatable. The main use of modulation is at the top-level: breaking up separate domains into separate encodable hierarchies. For this method to be useful in multiple inheritance hierarchies, a more robust definition of a module must be found.

In chapter 4, we use a more general concept of a module in a divide and conquer search algorithm on hierarchies. A module can be a subposet that has some structure which is partially repeated in the poset. This structure can be exploited for search and encoding. For example, a boolean space such as $B_6$ has only one module fitting the definition of Hassan Aït-Kaci et al., but the diagram of $B_6$ in Figure 5.6 is a recursive modulated view of a boolean space. In these modules, a submodule has a corresponding element in a supermodule. For example, in the module 10, the element $\mathbf{b}$ corresponds to element $\mathbf{a}$ in the supermodule 11, because each is the respective top element of their modules. The module 11 can be seen to project into the module 10, similarly it also projects into the module 01 and transitively 00. The module concept is worthy of further analysis, because it represents the structure in the poset that is being exploited in efficient encoding and search methods.

In a logic programming language, the types used in programs can be encoded at compile time. A program execution can include a set of type unification operations. Using the above modulated code scheme, type unification would be implemented using the procedure $X \cap Y$ discussed above. Aït-Kaci et al. [AKBLN89] say that only at the time of writing solutions is decoding $\tau^{-1}$ of an answer needed. However, to decide if two types unify, failure must be detected. The resultant code computed in the $X \cap Y$ operation must be checked to see if it is bottom or in the absurd (bottom) space. The shaded area in Figure 5.6 is the absurd space in this binary tree
example. The operation which detects bottom is essentially a decoding operation followed by a check that the decoded term is bottom in the original type hierarchy.

5.2.6 More Poset Characteristics

The poset characteristics used so far to estimate the size of possible encoding methods are crude. In chapter 2, definition 2.2.1 the concepts shoots, atoms and coatoms were defined. In the following examples $S$ is the entire poset $P$. In Figure 4.3 the $\text{Shoots}(\text{Ancestors}=i, S) = \text{Shoots}\{a, b, c, f, g, i\}, S) = \{d, e, j, k, l\}$, and $\text{Shoots}(\text{Ancestors}=h, S) = \text{Shoots}\{a, b, e, h\}, S) = \{c, d\}$. The shoots of an ancestor space are the closest set of meet-irreducible objects which are not ancestors or descendants. In the boxes domain, the shoots of box $u$ correspond to the closest boxes to $u$ which are directly included in only one box and are incomparable to $u$. In the image database in Figure 1.3(a) $\text{Shoots}(\text{Ancestors}=\text{SparklingDiamond}, S) = \text{Shoots}\{\text{Blank, Diamond, SparklingDiamond}\}, S) = \{\text{Sierpinski Triangle}\}$. Shoots of the ancestors of $u$ include all those distinct primitive images and attributes of images which are added to or modify subimages of $u$. In Figure 5.6, $\text{Shoots}(\text{Ancestors}=h, S), S) = \text{Shoots}\{a, b, d, h\}) = \{c, e, i\}$.

In Figure 4.3 $\text{Atoms}(S) = \{b, c\}$ assuming $a$ is $\top$. The atoms correspond to the primitive concepts or base classes in a knowledge base. In Figure 1.3(a), $\text{Atoms}(S) = \{\text{Sierpinski Triangle, Diamond}\}$, that is, $\text{Atoms}$ are the primitive subimages assuming the Blank image is $\top$. In Figure 5.6, $\text{Atoms}(S) = \{b, c\}$ assuming $a$ is $\top$.

In Figure 4.3 $\text{CoAtoms}(S) = \{d, h, i, j, k, m\}$. The coatoms usually correspond to the instances or instance classes in a class hierarchy. In Figure 1.3(a) $\text{CoAtoms}(S) = \{\text{Triangle and Sparkling Diamond}\}$, that is, coatoms are the final composite images. In Figure 5.6 $\text{CoAtoms}(S) = \{h, i, j, k, l, m, n, o\}$, that is, the leaves of a tree are coatoms assuming leaves are implicitly connected to $\bot$.

We need more detailed characteristics to evaluate the encoding methods discussed in later sections.

**Definition 5.2.5** The following terms describe sets in a poset $\overline{P} = (P, \sqsupseteq)$. We assume a subposet $\overline{S} = (S, \sqsupseteq)$, such that $S \subseteq P$. 
5.2. PREVIOUS ENCODING METHODS

- \( \text{Ideal}(u, S) = \text{Ancestors}(u, S) \)

- \( \text{IdealAndShoots}(u, S) = \{ v : S \mid v \in (\text{Ideal}(u, S) \cup \text{Shoots(\text{Ideal}(u, S), S))) \} \)

- \( \text{CoAtomIdealAndShoots}(S) = \)
  \[ \{ I : PS \mid \exists u : \text{CoAtoms}(S) \cdot I = \text{IdealAndShoots}(u, S) \} \]

- \( \text{CoAtomIdealAndShootsMeets}(S) = \)
  \[ \{ M : PS \mid \exists I : \text{CoAtomIdealAndShoots}(S) \cdot M = \text{MeetIrreducibles}(I) \} \]

- \( \text{CoAtomIdealAndShootsMeetSizes}(S) = \)
  \[ \{ n : N \mid \exists M : \text{CoAtomIdealShootsMeets}(S) \cdot n = \# M \} \]

The \( \text{IdealAndShoots}(u, S) \) include the ancestors of \( u \) and meet-irreducibles in the children of the ancestor space. This set could be thought of as the ancestors and their immediate families. For example, in Figure 4.3 the Ideal of \( i \) (ancestors and \( i \)) are \( \{a, b, c, f, g, i\} \) and the shoots are \( \{d, e, j, k, l\} \). Hence \( \text{IdealAndShoots}(i, S) \) is \( \{a, b, c, d, e, f, g, j, k, l, i\} \). We sometimes use the synonym \( \text{AncestorsImmediateFamilies}(u, S) \) for \( \text{IdealAndShoots}(u, S) \).

The \( \text{CoAtomIdealAndShoots}(S) \) are the \( \text{IdealAndShoots} \) of each coatom in poset \( S \). For example, in Figure 4.3 each coatom in \( \text{CoAtom}\{d, h, i, j, k, m\} \) corresponds to an \( \text{IdealAndTwig} \) set in \( \text{CoAtomIdealAndShoots}(S) = \{\{a, b, c, d, e\}, \{a, b, c, d, e, h\}, \{a, b, c, d, e, f, g, j, k, l, i\}, \{a, b, c, j, k, l\}, \{a, b, c, j, k, l, m\}\} \). These sets correspond to the ancestors’ immediate families of each coatom or instance.

The \( \text{CoAtomIdealAndShootsMeets}(S) \) are the set of meet-irreducibles in the \( \text{IdealAndShoots} \) of each of the coatoms. For example, in Figure 4.3 the corresponding sets of meets are \( \{\{b, c, e\}, \{b, c, d, e, h\}, \{b, c, d, e, j, k, l\}, \{b, c, j, k, l\}, \{b, c, j, k, l, m\}\} \). These sets correspond to the genes of the ancestors’ immediate families of each coatom or instance.

The \( \text{CoAtomIdealAndShootsMeetSizes}(S) \) is the set of sizes of the previous sets: \( \{3, 5, 6, 7\} \), that is, an approximation to the number of genes of the ancestors’ immediate families for each coatom.
Figure 5.7: A compact hierarchical encoding of the decision tree $DT_{2,3}$

Figure 5.8: Compact hierarchical encoding of a hierarchy

### 5.2.7 Compact Hierarchical Encoding

Caseau [Cas93] has developed an subsumption encoding method based on the top-down compact encoding method called *compact hierarchical encoding*. Caseau noticed that allocating a separate gene to every meet-irreducible in top-down compact encoding was too generous, and it was usually sufficient to allocate a gene to differentiate a meet-irreducible from its ancestors immediate families and its descendants.

Compact hierarchical encoding is a top-down level encoding which applies the following rule to a meet-irreducible $u$ in the first phase

$$\tau(u) \leftarrow \tau(\text{Parent}(u, S)) \lor \text{select gene not in}(\tau(\text{Ancestors Immediate Families}(u, S)))$$

and the usual rule to encode a non-meet-irreducible $u$

$$\tau(u) \leftarrow \bigvee_{v \in \text{Parent}(u, S)} \tau(v)$$

The method repairs any conflicts created by this allocation in a second phase.
Figure 5.9: Compact hierarchical encoding of the boxes hierarchy in Figure 4.3

Figure 5.7 shows a compact hierarchical encoding of a tree. In the case of the decision tree $DT_{2,3}$ the code is determined by $[\text{depth}(S) \times \text{averageBranching}(S)]$. In this case, $\text{depth}(S) = 3$ and $\text{averageBranching}(S) = 2$, hence the code length is 6. This can be seen by noting there are $\text{depth}(S)$ meet-irreducibles in the ideal of a coatom and there are $[\text{depth}(S) \times (\text{averageBranching}(S) - 1)]$ twigs. The hierarchy is mapped upside-down in $B_6$ to reflect that the coding operations are opposite to those used in the bottom-up encodings. Here ⊕ is implemented by bit-vector OR, whereas in a bottom-up encoding ⊕ is implemented using bit-vector AND.

The size of a compact hierarchical encoding is closely related to $\text{max}(\text{CoAtomIdealAndShootsMeetsSizes}(S))$, since when encoding a coatom meet-irreducible it is sufficient to consider only the genes of the ancestors’ immediate families (ancestor and shoot meet-irreducibles). Figure 5.8 (b) is like an aerial view of a LAURE class subhierarchy [Cas93]. Figure 5.8 (a) is some of the primitive classes in Caseau’s class hierarchy. In Figure 5.8 (c) shows that the code size of the hierarchy is 7, which is the same size of the meet-irreducibles in the ideal and shoots of $o$ in Figure 5.8 (d).

Figure 5.9(a) shows the box hierarchy in Figure 4.3(a) with an extra box added; the intersection of boxes $b$ and $c$, named $b \cdot c$. The box $b \cdot c$ is added as part of the ideal completion [Cas93] of a poset undertaken before using compact hierarchical encoding. Figure 5.9(b) shows the compact hierarchical encoding of this box hierarchy. The size of the code is dominated by the size of the $\text{IdealAndShootsMeets}(i, S)$ which is $\{b, c, d, e, f, g, j, k, l\}$, hence the code size is 9 bits. The character-
The characteristic \( \max(\text{CoAtomIdealAndShootsMeetSizes}(S)) \) is a lower bound on the size of compact hierarchical encoding, but it is not a lower bound on possible encoding methods. Figure 4.3 shows the same hierarchy coded with a combination of modulated encoding and compact hierarchical encoding. In this encoding 7 bits are used. The poset is broken into 4 modules \( \{a, b, c, b, c\}, \{b, d, e, h\}, \{b, c, f, g, i\}, \{c, j, k, l, m\} \), and then encoded. In Figure 4.3, the largest term code is \( <4,1,1> \) which requires 5 bits. The encoding given in Figure 5.10 is similar to the term encoding of the same hierarchy in Figure 4.3.

Compact hierarchical encoding is a subsumption encoding that does not directly support \( \sqcap \) and \( \sqcup \) with bit-vector operations. Compact hierarchical encoding intuitively encodes an object with the genes of its ancestors' immediate families. It seems more intuitive to only use the genes of ancestors, so there seems to be room for improvement on compact hierarchical encoding. Caseau [Cas93] also suggested a more compact encoding which we call modulated compact hierarchical which handles large branching more efficiently. Modulated compact hierarchical method will be discussed later.

Caseau uses a method which we call level ancestor encoding for tree hierarchies with no inheritance. The method uses the knowledge

\[
u \sqsupseteq v \Leftrightarrow u \in \text{Ancestors}(v, P, \text{depth}(u, P)).\]

This can be tested efficiently for trees since every element \( v \) except \( \top \) has only one ancestor at the depth of element \( u \). Thus \( u \) subsumes \( v \) if and only if
\[ \text{Ancestors}(v)[\text{depth}(u, P)] = u. \] The actual codes tested are unique integers representing each element \( u \).

### 5.2.8 Tree Encoding

Habib and Nourine [HN94] developed a technique based on identifying the meet (join) irreducibles of a lattice completion of a poset and encoding each object \( u \) in the lattice as the set of meet (join) irreducibles which are not ancestors (descendants) of \( u \). These sets can then be readily converted into bit codes using a bit to represent each meet (join) irreducible. The technique is called \textit{tree encoding} because of its origins in a tree representation of a poset, but would more meaningfully be called (meet) \textit{irreducible encoding}. Tree encoding encodes each object with the set of nonancestor meet-irreducibles:

\[
\tau'(u) \leftarrow (\text{MeetIrreducibles}(S) - \text{Ancestors}(u, S))
\]

The bit-vector code \( \tau \) is generated from the set of nonancestor meet-irreducibles \( \tau' \) by uniquely numbering each meet-irreducible and using that number as a bit-vector position:

\[
\tau(u) = \text{bitcodes}(\tau'(u))
\]

where \textit{bitcodes} sets the ith bit in bit-vector to 1 if i is in the set \( \tau'(u) \), otherwise to 0.

Figure 5.11 shows a tree encoding of a poset based on encoding in terms of nonancestor meet-irreducibles. Figure 5.11(b) gives the \( \tau' \) encoding of the poset in
Figure 5.11(a) and Figure 5.11(c) gives the $\tau$ encoding. There are 8 meet-irreducibles in this example, namely \{a, b, c, d, j, m, o, p\}.

Figure 5.11(c) uses 8 bits, which is more than the 7 bits required for compact hierarchical encoding in Figure 5.8. Tree encoding is a subsumption encoding. Consider the operation $a \sqcap f = \tau^{-1}(\tau(a) \land \tau(f)) = \tau^{-1}(01111111 \land 10011111) = \tau^{-1}(00011111)$ which subsumes the code for $\bot$, namely 00000000.

Habib and Nourine [HN94] showed that a lattice $\overline{L} = (L, \sqsupset)$ can be encoded with a bit-vector of size

$$min(#MeetIrreducibles(L), #JoinIrreducibles(L)).$$

Tree encoding uses exactly $min(#MeetIrreducibles(S), #JoinIrreducibles)$ bits. Tree encoding should be applied top-down if $#MeetIrreducibles(S)$ is smaller and bottom-up if $#JoinIrreducibles$ is smaller. In the bottom-up case, tree encoding of an object is in terms of nondescendant join-irreducibles.

In inheritance hierarchies $#MeetIrreducibles(S)$ is usually much smaller than $#JoinIrreducibles$, since $JoinIrreducibles(S)$ include all of the coatoms which correspond to the instances or instance classes and $MeetIrreducibles(S)$ corresponds to the methods or attributes in the hierarchy. Thus the size of a tree encoding of an inheritance hierarchy is close to $#MeetIrreducibles(S)$. Tree encoding encodes an object with the genes of the whole population while compact hierarchical encoding encodes an object with the genes of its ancestors' immediate families. Tree encoding is typically not as compact as compact hierarchical encoding for inheritance hierarchies.

### 5.2.9 Vector Time Encoding Causality Spaces

Fidge [Fid88, Fid89] and Mattern [Mat89] proposed a generalised non-standard model of time which consists of vector clocks. Clock-vectors are partially ordered and form a lattice. Structure of causality of process states and events in distributed systems is represented by using time stamps and a simple clock update mechanism. Vector time encoding is interesting because it is intended for narrow deep hierarchies, whereas the inheritance encodings we have examined so far are intended for wide
shallow hierarchies. There may be a useful cross fertilisation of ideas to produce an encoding technique for general posets regardless of shape.

Vector time encoding methods time stamp events in a set of $n$ communicating processes, where time stamps reflect the causality partial order. An event $u$ in process $i$ is time stamped with $\tau(u) : \mathbb{N}^n$ such that

- $\tau(u)[i]$ is the number of events which have occurred in process $i$ up to and including event $u$.

- for each process $j$ in $1..n - \{i\}

\[ \tau(u)[j] \] is the number of events known by process $i$ at event $u$ to have occurred in process $j$.

When process $i$ at event $u$ (with previous event $v$ in process $i$) receives a message from event $w$ in some process (where $v$ is possibly equal to $w$), the time stamp on $u$ is $\tau(u) = \tau(v) \cap \tau(w)$, where

- $\tau(u)[i] = \tau(v)[i] + 1$

- for every $j$ in $1..n - \{i\}$ $\tau(u)[j] = max(\tau(v)[j], \tau(w)[j])$.

The vector time encoding method supports causality of events. For every event
$u$ and $v$ with time stamps $\tau(u)$ and $\tau(v)$, respectively,

$$u \subseteq v \iff \tau(u) \subseteq \tau(v)$$

where subsumption in $u \subseteq v$ refers to event causality, and subsumption in $\tau(u) \subseteq \tau(v)$ refers to time vector ordering defined as

$$\tau(u) \subseteq \tau(v) \iff \forall i : 1..n \cdot \tau(u)[i] \leq \tau(v)[i].$$

That is, event $u$ can be causally affected by event $v$ if and only if $u$ has a larger time stamp than $v$.

Figure 5.12(b) illustrates the vector time stamping method on the process event graph in Figure 5.12(a). For example, the time stamp of event $h$ in process $P2$ is $\tau(h) = \tau(c) \cap \tau(f)$ which is $[1,1,0,0] \cap [0,0,1,2] = \max(0,1), (1+1), \max(0,1), \max(0,2)] = [1,2,1,2]$. Notice that process $P3$ has three events and the time stamp of the latest event $\tau(j)$ is $[1,3,3,2]$ which has a 3 in the third position.

Time-vector encoding is a subsumption encoding. Time vectors are a record of the number of events which have occurred before an event. For example, the code $\tau(j) = [1,3,3,2]$ records that 9 events have occurred before the event $j$ which could have affected event $j$. The event totals are partitioned into events for each process. Previous time stamp encoding methods used the number of events that occurred before an event as the time stamp, resulting in a loss of information.

Charron-Bost [CB90] has proven that for $n$ processes the time vector must be of length $n$ to capture the causality information. This is because these process hierarchies are possibly infinite in depth and any two events in different processes may in the future join, that is, one event’s descendant event in a process may communicate with a descendant event of the other process’ event. This means each process must have a record of its own events and hence a local clock (integer) for each process.

If there are $n$ processes and each process event chain is of average length $d$, then the size of the average vector is at least $n \lfloor \log_2 d \rfloor$ bits, since the local clock $\tau(u)[i]$ is a count of the number of events up to and including $u$ and hence can be represented using the usual binary representation of integers. A set of process event chains over a causally (partially) ordered set of events $S$ is a chain partition of $S$ (Figure 5.13 (b)
Figure 5.13: Vector time encoding of the hierarchy in Figure 5.8

shows the chain covering used to generate the time stamps). The size of the chain partition is at least as large as $W$, the width or maximal cochain of $S$. Hence, vector time encoding of a poset $S$ is approximately $O(W, \log_2 d)$, that is, width times the log of the depth. Vector time encoding is suitable for narrow deep hierarchies. This contrasts with the inheritance encodings examined earlier which are depth times the log of the width ($O(d \cdot \log W)$), and hence suitable for wide shallow hierarchies. For example, in Figure 5.12(b) the vector time method produced time stamps of length 4, term encoding (to be discussed in section 5.5) would produce a term code of length 8. Compact hierarchical encoding would produce a bit-vector of length 8 bits. However, Figure 5.13(a) shows that vector time encoding is not compact for inheritance hierarchies. Here the time vector is of length 7 (integers), whereas compact hierarchical encoding uses 7 bits, and in section 5.5 term encoding takes 3 (small integers).

Vector time encoding is not a term encoding since it has no concept of impossible causality. Thus vector time encoding can not be used to efficiently search in a complex object database or implement type unification in programming languages.

The $\cap$ operation for vector time encoding is efficiently computable and is computed as events occur, that is, as objects are added to the hierarchy. A time stamp of $u$ is calculated directly from its parent event $v$ and the time stamp on the message from event $w$. This begs the question “is it possible to encode inheritance hierarchies with a very simple procedure?”. Inheritance encoding methods are based on
the rule

$$\tau(u) \leftarrow \bigvee_{v \in \text{parents}(u, S)} \tau(v)$$

to compute the code of \( u \) from the codes of \( u \)'s parents. Conflict maintenance complicates the procedure for inserting objects in hierarchies.

Inheritance encoding may offer time vector encoding a way of recording the unrelatedness of events and processes. New research on vector time methods is on partitioning a process network into subdomains where full causality information is maintained only within each subdomain. It may be possible to make a closed-world assumption about the process network: any process that has not affected another process is not going to affect that process. When a message violates this noncausal restriction, the restriction is lifted, requiring the recomputation of time stamps which are in conflict. It may be possible to produce time stamps logarithmic in the number of processes using this assumption.

## 5.3 Supporting Lattice Operations with Subsumption Encodings

One reason why the previous subsumption encoding techniques are not term encodings is their inability to detect when a code resulting from a \( \tau(u) \cap \tau(v) \) operation
is in the absurd space, that is, when \( \tau^{-1}(\tau(u) \sqcap \tau(v)) = \bot \), where \( \tau^{-1} \) is a decode operation which we will describe here. We now define the \textit{absurd space} of a code space.

**Definition 5.3.1** Given an object poset \( P = (P, \sqsubseteq_P) \), a code poset \( G = (G, \sqsubseteq_G) \) and a subsumption encoding \( \tau: P \mapsto G \), then the \textit{absurd space} is defined as

\[
\text{Absurd}(P, G, \tau) = \{ x : G \mid \exists u, v : P \cdot u \sqcap_P v = \bot_P \wedge (\tau(u) \sqsubseteq_G \tau(v)) = x \}
\]

Habib and Nourine [HN94] gave a solution to the decoding problem. Given \( x = \tau(u) \sqcap \tau(v) \), they classify the code \( x \) by taking a path from \( \tau(\top) \) to a parent code \( y \) of \( x \). A code \( y \) is a parent of code \( x \), if there are no codes below \( y \) which subsume \( x \). If the parent code \( y \) of \( x \) is the code of a coatom in the object poset being encoded, then \( x \) is in the absurd space. This may not work in general, as the compact hierarchical encoding in Figure 5.8 (c) illustrates. The operation \( \tau(a) \sqcap \tau(c) \) gives \( 1000000 \lor 0010000 = 1010000 \). The parents of the code are \( \tau(a) \) and \( \tau(c) \) where \( a \) and \( c \) are not coatoms, but \( a \sqcap c = \bot \). Their method of classification of codes does approximately \textit{depth}(P) \ast \text{MaxDegree}(P) \) code operations.

If \( \tau(u \sqcap v) \) is subsumed by some element of \( \text{Crown}(\text{Absurd}(P, G, \tau)) \) then \( u \sqcap v \) is \( \bot_P \). For example, the absurd space of Leibniz’s universal characteristic encoding in Figure 5.4 (c) is \{15, 21, 35, 42, 60, 70, 105, 210\}, and the crown is \{15, 21, 35\}, that is, CatDog, CatBird, and DogBird. If any code produced by a \( \sqcap \) operation on code numbers is divisible by 15, 21 or 25, then the code is in the absurd space which corresponds to the Absurd type in the original type hierarchy in Figure 5.4 (a). For example, PetCat \( \sqcap \) Bird = \( \tau^{-1}(\tau(\text{PetCat}) \sqcap \tau(\text{Bird})) = \tau^{-1}(6 \ast 7) = \tau^{-1}(42) = \text{Absurd}(\text{Absurd is the} \bot \text{type in Figure 5.4 (a)}, \text{since} 42 \text{is divisible by 21}.

Given there are \( n \) objects in \( P \), then in the worst case there are \( O(n^2) \) codes in \( \text{Crown}(\text{Absurd}(P, G, \tau)) \). This is because the resultant codes from \( \sqcap \)'s can be “new” codes which are not codes of elements in \( P \). Any code which is in the crown or below the crown is absurd. Any code strictly below the code of a coatom of \( P \) is considered absurd. The absurd space defined above would include the pairwise \( \sqcap \)'s of codes of coatoms. It is possible to reduce the size of the crown tested by replacing all those codes below codes of coatoms with the coatom codes but using
a proper subsumption comparison \( \sqsubseteq \). We need the following definitions to suggest a new technique for detecting absurdity.

**Definition 5.3.2** The coatom codes set is defined as 
\[
CoAtomCodes(P, G, \tau) = \{ z \in G \mid \exists u : CoAtoms(P) \cdot \tau(u) = z \}.
\]

**Definition 5.3.3** The non-coatom absurd crown is defined as 
\[
NonCoAtomAbsurdCrown(P, G, \tau) = 
\text{Crown} \left( \{ y : \text{Absurd}(P, G, \tau) \mid \neg \exists z : CoAtomsCodes(P, G, \tau) \cdot z \sqsubseteq y \} \right).
\]

We suggest the construction of a decision graph [KS90] over 
\( NonCoAtomAbsurdCrown(P, G, \tau) \cup CoAtomCodes(P, G, \tau) \) which efficiently classifies codes to determine absurdity. We call this graph the absurd space decision graph. Figure 5.14 shows an absurd space decision graph for the compact hierarchical encoding in Figure 5.8 (c). Figure 5.14 (a) shows the hierarchy of patterns used to differentiate the codes, (b) shows the decisions used, and (c) shows a possible compilation of the decision graph into computer instructions. For a discussion of how to implement decision graphs see [KS90]. Decision graphs unlike decision trees [Col90] are necessarily linear in the pattern space they discriminate [KS90]. The decision graph is linear in the size of 
\( NonCoAtomAbsurdCrown(P, G, \tau) \cup CoAtomCodes(P, G, \tau) \), since each code corresponds to exactly one leaf in the decision graph. The depth of the decision graph is dependent on the size of the codes, since a bit of the codes is tested at each level and only those bits which are necessary for discrimination are tested.

In the case of compact hierarchical encoding the decisions in the graph in Figure 5.14 (b) correspond to the \( i \)th bit (\( X_i \)) set to 0 or 1. Each decision partitions the absurd set, so at most \( p \) bits are tested in a path where \( p \) is the length of the encoding. If a decision graph leaf is a code in \( CoAtomCodes(P, G, \tau) \), then the query code must be strictly subsumed by the coatom code, otherwise (in \( NonCoAtomAbsurdCrown(P, G, \tau) \)) the code is of a clash in the original poset and only needs to subsume the query code to be absurd.
Proposition 5.3.4 Given the \((\text{NonCoAtom AbsurdCrown}(P, G, \tau) \cup \text{CoAtomCodes}(P, G, \tau))\) space the absurd space decision graph can detect whether a code \(x\) is in that space in the order of one code operation \(#x\).

Proof: The cost of a decision, which is constant, is denoted as Decision. The size of the largest code in \(\text{NonCoAtom AbsurdCrown}(P, G, \tau) \cup \text{CoAtomCodes}(P, G, \tau)\) is denoted as \(p\). The complexity of the decision process is \((\text{Decision} \times p) + p\), since each decision corresponds to one bit in the code being tested and in the worst case all the bits are needed for discrimination. Hence the decision process is \(O(p)\), that is, in the order of one code operation.

\(\square\)

The absurd space decision graph method compares favourably in speed with Habib and Nourine's code classification method for absurd detection [HN94] which does \(\text{depth}(L) \times \text{MaxDegree}(L)\) code operations.

5.4 More Poset Definitions

We introduce some more concepts for finer measurement of the previous methods and for use in new methods.

Definition 5.4.1 The following terms describe sets in a poset \(\overline{P} = (P, \supseteq)\). We assume a subposet \(\overline{S} = (S, \supseteq)\), such that \(S \subseteq P\).

- \(\text{Clash}(u, S) = \{ v : S | u \cap v = \bot \}\)
- \(\text{ClashCrown}(u, S) = \text{Crown}(\text{Clash}(u, S))\)
- \(\text{SimpleClashCrown}(u, S) = \text{ClashCrown}(u, S) - \bigcup_{v:\text{Ancestor}(u, S)} \text{ClashCrown}(v, S)\)
- \(\text{globpartition}(S)\) is a GLB partition of \(S\) which is maximal.
- \(\text{SimpleAllClashCrown}(u, S) = \text{globpartition}(\text{SimpleClashCrown}(u, S))\)
Figure 5.15: Encoding a clashing and unifying hierarchy

- \( \text{ClashGroups}(S) = \{ T : \mathbb{IP}S \quad \exists u : S \cdot \exists R : \text{SimpleAllClashCrown}(u, S) \cdot T = \{u\} \cup R\} \).

- \( \text{group} : \text{Var} \xrightarrow{\text{bij}} \text{ClashGroups(MeetIrreducibles}(S)) \) is an onto function where \( \text{Var} \) is a set of variables.

- \( \psi : S \rightarrow 1..\#S \) is a one to one function from elements of \( S \) to numbers in the range 1..\#S.

- \( \text{same} : \text{dom(group)} \leftrightarrow \text{dom(group)} \)

\[
\forall X, Y : \text{dom(group)} \implies \text{same}(X, Y) \iff \\
\quad \text{there are numberings of group}(X) \text{ and group}(Y), \psi_X \text{ and } \psi_Y, \text{ respectively, such that} \\
\quad \forall u : \text{group}(X), v : \text{group}(Y) \implies \psi_X(u) \neq \psi_Y(v) \iff u \sqcap v = \bot
\]

Further same is transitive

\[
\forall X, Y, Z : \text{dom(group)} \implies \text{same}(X, Y) \wedge \text{same}(Y, Z) \implies \text{same}(X, Z)
\]

The clash set \( \text{Clash}(i, P) \) in Figure 4.3 is \( \{d, e, h, j, k, l, m\} \), that is, the set of elements which do not have a common descendant with \( i \) (other than the default \( \bot \)). The \( \text{ClashCrown}(i, P) \) is \( \{d, e, j, k, l\} \). The clash crown of \( u \) corresponds to the set
of maximal objects which do not have descendants in common with \( u \). In compact hierarchical encoding, when allocating a gene to \( u \), the gene must be different from \( \text{Twigs}(\text{Ancestors}(u, S)) \), that is the first layer away from the ancestor space. In term encoding \( u \) is allocated a gene which clashes with (not only incomparable to) the first layer of clashing objects not already differentiated by ancestor codes which is usually a subset of the twigs. The simple clash crown of \( u \) corresponds to the set of objects which have no common descendants with \( u \) (other than \( \bot \)) which do have common descendants with all of \( u \)'s ancestors. \( \text{SimpleClashCrown}(i, P) = \emptyset \), since \( f \) is an ancestor of \( i \) and their clash crowns are the same, that is, there is no element which clashes with \( i \) and does not clash with some of \( i \)'s ancestors. Figure 5.15(e) gives each of the sets discussed here for the hierarchy in Figure 5.15 (a). For example \( g \) clashes with object \( d, f \) and \( h \), the crown of which is \( \{d, f\} \), but \( g \)'s ancestor \( c \) also clashes with these elements. Object \( c \) has a simple clash crown of \( \{d, f\} \) since they all clash with \( d \) and they unify with all of \( c \)'s ancestors.

A global partition of the subset of objects \( \{d, f\} \) in Figure 5.15 (a) is \( \{\{d\}, \{f\}\} \).

\( \text{SimpleAllClashCrown}(u, S) \) breaks the clash sets in \( \text{SimpleClashCrown}(u, S) \) into maximal subsets where elements in the subset pairwise clash. The motivation for this set is that we wish to allocate conflicting genes to \( u \) and subsets of elements of \( \text{SimpleAllClashCrown}(u, S) \), so pairwise elements in \( \text{SimpleAllClashCrown}(u, S) \) must also clash. In the boxes hierarchy example this has no affect, since the simple clash crowns already satisfy this. Figure 5.15(a) gives a hierarchy where this distinction can be made. The objects \( d \) and \( f \) are the maximal clashes of object \( c \). If clashing genes were allocated to \( c, d \) and \( f \), then the codes of \( d \) and \( f \) would not unify, but the objects \( d \) and \( f \) do unify. Hence different clashing genes must be allocated to \( \{c, d\} \) and \( \{c, f\} \).

The \( \text{ClashGroups}(\text{MeetIrreducibles}(S)) \) is a grouping of the set \( S \) into the minimal sets which clash. For example, the clash groups of Figure 4.3 are \( \{\{b, j, k, l\}, \{c, d, e\}, \{f\}, \{g\}, \{h\} \{m\}\} \). In Figure 5.15 (e), the clash groups of the meet-irreducibles is the last column \( \{\{a\}, \{b\}, \{c, d\}, \{c, f\}, \{e, f\}, \{d, e\}\} \).

The function \( \text{group} \) maps variables to clash groups. For example, in Figure 5.9 \( \{X_1 \rightarrow \{b, j, k, l\}, X_2 \rightarrow \{c, d, e\}, X_3 \rightarrow \{f\}, X_4 \rightarrow \{g\}, X_5 \rightarrow \{h\} X_6 \rightarrow \{m\}\} \).
In Figure 5.15 (e), \( \{X_1 \rightarrow \{a\}, X_2 \rightarrow \{b\}, X_3 \rightarrow \{c, d\}, X_5 \rightarrow \{e, f\}, X_4 \rightarrow \{e, f\}, X_6 \rightarrow \{d, e\}\} \).

The relationship \textit{same}(X, Y) is true for variables X and Y if and only if there are compatible numberings of their respective clash groups so that if elements of different clash groups are assigned different numbers only if they clash. For example, in Figure 5.15 (e) \textit{same} = \{(X_3, X_5), (X_4, X_6)\}.

**Definition 5.4.2** The following terms describe sets in a poset \( P = (P, \sqsubseteq) \). We assume a subposet \( S = (S, \sqsubseteq) \), such that \( S \subseteq P \).

- \( \text{CoAtomIdeals}(S) = \{I : PS \mid \exists u \in \text{CoAtoms}(S) \bullet I = \text{Ideal}(u, S)\} \)

- \( \text{CoAtomIdealMeets}(S) = \{M : PS \mid \exists I : \text{CoAtomIdeals}(S) \bullet M = \text{MeetIrreducibles}(I)\} \)

- \( \text{MaxCoAtomIdealMeets}(S) = \max(\{n : N \mid n = \#\text{CoAtomIdealMeets}(S)\}) \)

The CoAtomIdeals(S) is the set of sets of coatom ancestors of the poset. The size of an encoding of a poset is related to the size of one of these sets (not necessarily the largest). In Figure 5.9, the CoAtomIdeals is the set \( \{\{a, b, d\}, \{a, b, e, h\}, \{a, b, c, b, c, f, g, i\}, \{a, c, j\}, \{a, c, k\}, \{a, c, l, m\}\} \). The intuition is that the more ancestors a class has, the more inherited attributes the class has. The set \( \text{CoAtomIdealMeets}(S) \) represents the new attributes or methods that are inherited for each coatom, since each meet-irreducible corresponds to a method or attribute. For example, CoAtomIdealMeets(S) is \( \{\{b, d\}, \{b, e, h\}, \{b, c, f, g\}, \{c, j\}, \{c, k\}, \{c, l, m\}\} \). Notice that the size of each of these sets corresponds (exactly in this case) to the number of instantiated variables in the codes of the corresponding coatoms in Figure 4.3.

\( \text{MaxCoAtomIdealMeets}(S) \) is the maximum size of sets in \( \text{CoAtomIdealMeets}(S) \). For example, in Figure 5.9 the maximal size is 4 given the set of meet-irreducibles of the coatom \( i \), \( \{b, c, f, g\} \). According to this measure \( i \) would have the largest description, since it is a coatom in the poset with the largest number of meet-irreducibles as ancestors and will inherit the largest number of genes.
5.5 Term Encoding a Lattice

Term encoding of hierarchies has been proposed earlier [MD88, FD93, Mel88, Mel90], but as yet no general algorithms have been developed. We have not yet developed a general algorithm for term encoding, but show some of the process and illustrate the concept on some small examples and later discuss general methods on some restricted kinds of hierarchies. The term encoding method is based on the observation that an object should only inherit methods or attributes from its ancestors. The code size of an object is dependent to some extent on the ancestor meet-irreducibles and hence the size of the encoding of the poset $S$ is influenced by $\text{MaxCoAtomIdealMeets}(S)$. Consider the hierarchy in Figure 5.16, the poset can be decomposed into a set of CoAtomIdeals. Each of these subposets can be encoded with 3 genes using the
Figure 5.18: Coding a term graph

Figure 5.19: A recursive view of the term encoding in Figure 5.18
variables \{X_1, X_2, X_3\} rather than the previous best for the whole poset of 7 (bit) genes using compact hierarchical encoding. For larger posets with more diversity of knowledge (more branching) the savings can be significant. Figure 5.17 illustrates the general problem of a term encoding. A gene must be allocated to each (meet-irreducible) \(u\) to differentiate \(u\) from \(u\)'s ancestors and to make \(u\) clash with the codes of \(u\)'s minimal set of clashes.

Figure 5.18 shows the steps required to term encode a poset. The first step (b) is to identify the meet-irreducibles and groups them into ClashGroups. This can be done as illustrated in the refinement of the initial set of clashes of each meet-irreducible in Table 5.2. This step is dominated by finding the clash set of each element of the poset. If the size of the poset \(S\) is \(n\), then this process involves \(O(n^2)\) steps. The next step, Figure 5.18 (c), is to allocate a separate variable to each of the groups (determining the group function defined earlier). Step (d) involves checking which groups can be allocated the same variable. This test involves finding compatible numberings \(\psi\) for each group sharing the same variable. In this example, \(X_3\) and \(X_4\) can be the same. In step (e) each element of group is assigned a unique number 1..\#ClashGroup determined in step (d) to the ClashGroup variable. Step (f) is a labelling of the graph with the codes implied by step (e). Step (g) is a top-down topological walk where the code of each element is determined by unifying the codes of its parents to get the final encoding of the lattice. Figure 5.19 shows a recursive or modulated view of the encoding of the hierarchy in Figure 5.18. The mapping of the hierarchy in Figure 5.18 into a term lattice is shown (this term lattice is not uniform since some levels have branching of 3 while others have branching 2).

In the case of boolean trees such as Figure 5.20 term encoding provides compact
encodings. The number of genes allocated is the same as the length of the paths in the tree. Figure 5.20(a) shows the ClashGroups of the binary balanced tree. The graph in (b) is of the same relation. That is each level can use the same variable. The instantiation of each variable is done by numbering the elements in each ClashGroup from 1..#ClashGroup, in this example 1 or 2. The table in (c) gives the encoding of the tree. The encoding can be seen as mapping the CoAtomIdeals (a chain) into the boolean lattice $B_3$ as in (d).

Term encoding has an advantage over bit-vector encoding on trees when branching increases as indicated by Figure 5.21. Bit-vector encoding must ensure that the bit-vectors allocated to each branch element are pairwise incomparable. To encode $k$ branches, naive bit-vector encoding allocates a bit to each element and so uses $k$ bits to achieve this. The notation $0^*$ indicates trailing zeroes. The largest set of incomparable bit-vectors is the maximal cochain in a boolean lattice. Given the boolean lattice $B_n$, where $n = \lceil \log k \rceil$ the maximal cochain in a boolean lattice corresponds to $\binom{n}{\lceil \log k \rceil}$ which is approximated by $\frac{2^n}{\sqrt{\pi n}}(1 + O(\frac{1}{n}))$ [CLR90] (p. 104). Term encoding uses most if not all of a boolean lattice rather than a single (maximal cochain) level. The $k$th branch element is allocated the integer $k$ which can be
encoded in $\log_2 k$ bits. This means that bit-vector encoding uses $\frac{\sqrt{\pi k}}{(1+O(\frac{1}{n}))}$ times the number of bits used by term encoding on branches.

### 5.5.1 Implementing Term Code Operations

The purpose of encoding methods is to implement very fast operations over compact codes which implement hierarchical and lattice operations. This allows complex object comparisons to be replaced with simple machine operations on bit-vectors. The bit-vector encoding methods produce compact bit-vector codes which support subsumption using fast parallel bit-wise operations. GLB and LUB operations are not directly supported by bit-vector operations on bit-vector codes. In section 5.3, we described a new method for supporting GLB using a decision graph over the crown of the absurd space. This decision graph process is hard to parallelise, since it relies on the dependencies between decisions made.

How can term codes be implemented efficiently? A term code could be translated to a bit-vector by translating all variables to 0 and every number greater than 0 to its binary representation. For example, $<1,5>$ is translated into 1 101 and $<1,1>$ to 1 1. This translation does not directly support subsumption nor GLB. Consider $<1,1> \nsubseteq <1,5>$, but $11 \lor 1101 = 1101$. Alternatively, term codes could be translated into a subsumption bit-vector encoding similar to that produced by compact hierarchical modulated encoding. A variable is translated into a vector of $k$ ones and numbers are translated into vectors where $\binom{k}{\frac{k}{2}}$ bits are set to 1 for some suitably large $k$. The codes would not be as compact as term encoding, but a parallel bit-wise operation could be used for subsumption. GLB and LUB operations would
not be directly supported. Figure 5.22 shows a sample term encoding (a) of a poset expanded to a bit-vector encoding in (b).

The terms which are generated by term encoding are a special kind of term where matching and unification can be implemented very efficiently. There are no repeated variables or nested structures. Each operation on a variable can be implemented very efficiently with a special machine instruction, and a series of operations could be handled in one parallel machine instruction.

Unlike the sequential decision graph process on bit-vectors for supporting GLB, a term code can be computed in parallel. Each gene component of a term can be tested independently. If we wish to compute \( x \cap y \) for the terms \( x \) and \( y \), we can test the genes \( X_i \) and \( Y_i \) for each \( i \). The whole term operation fails if for any \( i \) the variables \( X_i \) and \( Y_i \) are instantiated with different integers.

### 5.6 Analysis of Term Encoding Efficiency

In the following sections we compare encoding methods in terms of compactness of codes on some partially ordered set benchmarks and then show how term encoding performs relative to previous methods on actual object databases and class hierarchies.

In chapter 2 benchmark hierarchies such as chains, decision trees, term lattices and boolean lattices are discussed. The attributes of these are discussed in the context of classification in object hierarchies. We now use the same benchmarks for encoding techniques.

In Table 2.2 in chapter 2, for each kind of hierarchy (column) the minimum of the value in the \#MeetIrreducibles and \#JoinIrreducibles rows gives an upper bound for the number of bits required for encoding the corresponding hierarchy since all the posets are lattices [HN94]. This is a very conservative upper bound, since compact hierarchical encoding is closer to \#MaxCoAtomIdealMeets + \#Twigs(MaxCoatomIdeals). Caseau [Cas93] has shown that for trees compact hierarchical encoding uses max \{sum of degrees of a chain\} bits. This can be improved significantly compact hierarchical modulated encoding and further improved
Table 5.3: The characteristics of some example spaces

<table>
<thead>
<tr>
<th></th>
<th>Chain</th>
<th>Decision Tree</th>
<th>Term Lattice</th>
<th>Boolean Lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxCoAtomIdealMeets + #Twigs(MaxCoAtomIdeals)</td>
<td>n-1</td>
<td>k. (log₄n)</td>
<td>k-1.( log₄ n)</td>
<td>log n</td>
</tr>
<tr>
<td>MaxCoAtomIdealMeets</td>
<td>n-1</td>
<td>log₄n</td>
<td>log₄n</td>
<td>log n</td>
</tr>
<tr>
<td>max(#SimpleClashCrown(u, P))</td>
<td>1</td>
<td>k</td>
<td>k-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5.23: Comparison of term encoding with compact hierarchical encoding on a $TL_{2,8}$

using term encoding. The number of genes used in term encoding is related to MaxCoAtomIdealMeets and is likely to be a lower bound on an inheritance encoding. The dimension $\max(#\text{SimpleClashCrown}(u, P))$ indicates the likely size of the integers assigned to variables in a term encoding.

Table 5.4 gives the corresponding code lengths in bits for each of the encoding methods discussed earlier for each poset benchmark. The benchmark spaces are defined in chapter 2. We discuss each benchmark in detail.

Chain: Inheritance encoding methods are thought to be optimal if they are linear in the depth of a hierarchy. For chains, transitive closure, compact bottom-up, compact top-down, tree, modulated, compact hierarchical, compact hierarchical modulated, level ancestor and term encoding all have codes linear in the depth of a chain. Leibniz’s universal characteristic is not quite as compact.

In level ancestor encoding the $i$th element in the chain is encoded with a vector of $i - 1$ integers representing the depths of its ancestors, hence the bottom element is encoded with a vector of $n - 1$ integers. However, in the
Table 5.4: Comparison of encoding methods on various spaces in Table 5.3

<table>
<thead>
<tr>
<th>Encoding Methods (authors, year)</th>
<th>Chain</th>
<th>Decision Tree DT_{k,d}</th>
<th>Term Lattice TL_{k,d}</th>
<th>Boolean Lattice B_{d}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal Characteristic (Leibniz, 1679)</td>
<td>&gt; (n-1)ln(n-1)</td>
<td>&gt; (n-1)ln(n-1)</td>
<td>&gt; (k-1)(log_k n) + (k-1)(log_k n)</td>
<td>&gt; (log n)(ln n)</td>
</tr>
<tr>
<td>Transitive Closure</td>
<td>n-1</td>
<td>n-1</td>
<td>n-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>
| Compact bottom-up (Ait-Kaci, Boyer, Lincoln 
& Naar, 1989) | n-1 | (n+1)/k | (k-1)log_k n | log n |
| Compact top-down (Ellis, 1994) | n-1 | n-1 | (k-1)(log_k n) | log n |
| Tree (Habib & Nourine, 1994) | n-1 | (n+1)/k | (k-1)(log_k n) | log n |
| Modulated (Ait-Kaci et al., 1989) | n-1 | k(log_k n) | (k-1)(log_k n) | log n |
| Compact Hierarchical (Caseau, 1993) | n-1 | k(log_k n) | (k-1)(log_k n) | log n |
| Compact Hierarchical (Modulation) (Caseau, 1993) | n-1 | (log n) \frac{log_k n}{(1 + 20 log_k n)} (log k + 1) log_k n | (log n) \frac{log_k n}{(1 + 20 log_k n)} (log k + 1) log_k n | log n |
| Vector Time (Fidge, 1988) | log n | > (n+1)/k log_k n | > (log n) log_k n | log n |
| Level Ancestor (code compared) | (log n) | (log n) log_k n | (log n)^2 log_k n | n log n |
| Term (Ellis, 1994) | n-1 | (log k + 1) log_k n | (log k + 1) log_k n | log n |
case of chains encoding size can be significantly reduced. The \(i\)th element should be directly encoded with \(i\). The \(i\)th element of the chain subsumes the \(j\)th if and only if \(i \leq j\) as is done in vector time encoding. This results in codes that are logarithmic in the depth of the chain. Depth is not a measure of optimality, since vector time encoding can encode a chain using \(\log n\) bits, effectively encoding the depth with an integer.

**Boolean Lattice:** Most methods are compact for boolean lattices, \(\log n\), however, Leibniz's Universal Characteristic does not perform well. Vector time encoding is related to \(\text{width} \times \log \text{depth}\) and is hence poor at encoding boolean lattices being almost linear in the number elements. Level ancestor encoding also performs badly on boolean lattices.

**Decision Tree:** Term encoding of the decision tree \(DT_{k,d}\) produces a code more compact than the previous best methods. Compact hierarchical encoding with modulation and level ancestor encoding are competitive. Caseau uses level ancestor encoding in preference to compact hierarchical encoding for trees though it is not clear that level ancestor encoding is superior for trees. Term encoding is more compact than level ancestor encoding by a factor of \(\frac{\log n}{\log k} + 1\) for decision trees \(T_{k,d}\) and approximately \(\sqrt{\pi \log k}\) more compact than compact hierarchical encoding with modulation. The only other encoding to perform reasonably on trees is Hassan Aït-Kaci et al.'s modulated encoding method.

Term encoding is more effective than other encoding methods on trees with large branching factors because each branch is encoded with a unique number rather than separate bits used in compact hierarchical. Numbering is also more efficient than the bit-vectors from the maximal cochain of a boolean lattice used in compact hierarchical modulated. Figure 5.23 compares term encoding with compact hierarchical encoding on a term lattice of depth 2 and branching factor of 8. The term codes, compact hierarchical bitcodes and compact hierarchical modulated bitcodes are given for the left most and right most elements on each level. The right most element of the bottom level has a bit-vector encoding of 0000000100000001 (non-modulated), 1100011000 (modulated) and a term code of \(<8,8>\). The term code \(<8,8>\) could ultimately
be implemented using a bit-vector of 00010001.

**Term Lattice:** Most methods perform reasonably well for term lattices ($TL_{k,d}$) due to the multiple inheritance. The level ancestor method which was designed for single inheritance unsurprisingly performs poorly on term lattices. Vector time encoding also perform badly due to the relatively large width of the hierarchy compared to the depth. Term encoding again outperforms the previous best method of compact hierarchical modulated encoding.

Figure 5.23 illustrated the compactness of term encoding on branching in a tree. This argument extends to term lattices. The compact hierarchical bit-vector encoding of $TL_{d,1000000}$ will be 50000 times the size of the term encoding and the compact hierarchical modulated bit-vector encoding of $TL_{d,1000000}$ will be approximately 6 times the size of the term encoding.

### 5.6.1 Results on Real Object Databases

We have not yet completed the design of a term encoding algorithm, but have implemented some metrics which give an indication of the expected size of the term encoding, and use these to compare term encoding to previous methods.
Table 5.6: Further topology statistics of some databases and indications of code sizes in bits

<table>
<thead>
<tr>
<th>Database</th>
<th>Nodes</th>
<th>Join Irreducibles</th>
<th>Meet Irreducibles</th>
<th>#MaxCoAtomIdealMeets + #Shoots(MaxCoAtomIdealMeets)</th>
<th>Max Immediate Family Meets</th>
<th>Bit-Vector Code (Upper Bound approx)</th>
<th>Term Code (Upper Bound approx)</th>
<th>Term Code (Lower Bound approx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A without</td>
<td>1778</td>
<td>1200</td>
<td>360</td>
<td>161</td>
<td>23</td>
<td>51</td>
<td>184</td>
<td>138</td>
</tr>
<tr>
<td>A with basis</td>
<td>1815</td>
<td>1199</td>
<td>67</td>
<td>50</td>
<td>17</td>
<td>10</td>
<td>85</td>
<td>68</td>
</tr>
<tr>
<td>A’ without</td>
<td>1783</td>
<td>1200</td>
<td>260</td>
<td>127</td>
<td>21</td>
<td>29</td>
<td>147</td>
<td>145</td>
</tr>
<tr>
<td>A’ with basis</td>
<td>1820</td>
<td>1199</td>
<td>36</td>
<td>53</td>
<td>21</td>
<td>11</td>
<td>126</td>
<td>84</td>
</tr>
<tr>
<td>B without</td>
<td>3104</td>
<td>2958</td>
<td>456</td>
<td>152</td>
<td>2</td>
<td>144</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>B with basis</td>
<td>3146</td>
<td>2980</td>
<td>352</td>
<td>41</td>
<td>5</td>
<td>24</td>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>B’ without</td>
<td>3109</td>
<td>2959</td>
<td>149</td>
<td>153</td>
<td>4</td>
<td>148</td>
<td>40</td>
<td>32</td>
</tr>
<tr>
<td>B’ with basis</td>
<td>3151</td>
<td>2980</td>
<td>33</td>
<td>40</td>
<td>7</td>
<td>25</td>
<td>49</td>
<td>35</td>
</tr>
<tr>
<td>Laure Lattice</td>
<td>296</td>
<td>212</td>
<td>274</td>
<td>46</td>
<td>12</td>
<td>7</td>
<td>60</td>
<td>36</td>
</tr>
</tbody>
</table>

The databases we use are hierarchies of chess board schemas used in the Morph [LS91] chess learning system described in Table 2.3 in section 2.9.5 in chapter 2. We added the schemas in Figure 2.35 in chapter 2 to the database “A” to get the database A’. Similarly, the schemas were added to each database “A with basis”, B, and “B with basis”. A third hierarchy used is the class library hierarchy from the Laure language [Cas93]. Table 5.5 shows some topology statistics about each of these databases. In database A there are 1778 nodes, and the hierarchy has depth 4 and 912 coatoms.

In Table 5.6 the methods compact bottom-up, tree, compact top-down, compact
hierarchical, compact hierarchical modulated and term encoding are compared in terms of the number of bits they use on the hierarchies in Table 5.5. We examine each column in detail.

**Nodes:** The Nodes column gives the size of the corresponding hierarchy in terms of number of nodes.

**JoinIrreducibles:** This column illustrates that compact bottom-up encoding is not suitable for encoding complex object databases nor class hierarchies, since it produces codes almost linear in the size of the hierarchy, since compact bottom-up encoding size is lower bounded by the number of join-irreducibles in the hierarchy.

**MeetIrreducibles:** This column gives the number of meet-irreducibles in each poset. The size in this column contrasts with the corresponding element in the JoinIrreducibles column. This shows that by reversing the direction of encoding from bottom-up to top-down, the size of the code can be dramatically decreased. In the case of “B’ with basis” the code is reduced by a factor of 90. This is probably a lower bound on the size of the code produced by compact top-down and Habib and Nourine’s tree encoding. In most cases it is still relatively expensive compared to compact hierarchical and term encoding. Interestingly, by adding 42 basis graphs to database X to get “X with basis” reduced the size of the code by between 22.8 percent and 86.2 percent. Similarly, adding the 5 schema graphs in Figure 2.35 to database X to get X’ resulted in compression of the codes by between 27.8 percent and 67.3 percent. These dramatic changes indicate that reshaping of hierarchies by adding elements is a useful research direction. The effects are continued through to the more compact encoding methods, but not as dramatically.

**#MaxCoAtomIdealMeets + #Shoots(MaxCoAtomIdeals):** This column is an estimate of the size of the code produced by the compact hierarchical (non-modulated) method. The column gives the size of the largest set of all ancestor’s immediate family meet-irreducibles of a coatom. This is a subset of all meet-irreducibles and is hence better than tree encoding or compact
top-down encoding given to the left.

**#MaxCoAtomIdealMeets:** This column approximates the number of variables used in a term encoding, since it gives the largest set of ancestor meet-irreducibles. Interestingly these numbers correspond more closely to the size of the conceptual graphs which are stored in the hierarchies.

**MaxImmediateFamilyMeets:** This column gives the maximum branching factor encountered in the poset. Compact hierarchical modulated encoding allocates a bit-vector from a boolean lattice maximal cochain. For example, 8 bits are needed to encode 51 branches, since 51 is between \binom{7}{3} = 35 and \binom{8}{3} = 70. Term encoding uses 6 bits, since \lfloor \log 51 \rfloor = 6.

**Bit-vector Code (Upper Bound approx):** This approximate upper bound for compact hierarchical modulated encoding is computed by multiplying the #MaxCoAtomIdealMeets column by the number of bits needed by modulated encoding to encode the branches in the MaxImmediateFamilyMeets column.

**Term Code (Upper Bound approx):** A term encoding approximate upper bound is computed by multiplying column #MaxCoAtomIdealMeets by \lfloor \log \#MaxImmediateFamilyMeets \rfloor. The approximate upper bound for term encoding is between 1 and 40 percent smaller than the approximate upper bound for compact hierarchical modulated encoding, showing that their handling of branching can be significant even for these relatively small databases.

**Term Code (Lower Bound approx):** An approximate lower bound for term encoding is log \left[ \frac{\#MaxCoAtomIdealMeets + \#Shoots(MaxCoAtomIdeals)}{\#MaxCoAtomIdealMeets} \right] times #MaxCoAtomIdealMeets. The first part giving a lower approximation on the average number of bits used for each branch.

Term encoding produces smaller codes than the previous best encoding method (compact hierarchical modulated). In terms of actual bits used term encoding could be expected to reduce the bits used in a compact hierarchical modulated encoding by anywhere from 0 up to 40 percent.
Tree encoding can be seen as inheriting genes from the whole population and compact hierarchical encoding inherits genes from ancestors immediate families, while modulated compact hierarchical encoding and term encoding inherit genes from ancestors. Term encoding is scalable since it reflects the objects being stored which also inherit information from their ancestors.

Term encoding will show its real potential on large databases, since the encoding method is effectively independent of database size, but rather on the objects being stored. In Table 5.6 the column #MaxCoAtomIdealMeets for database B gives a result of a term encoding of 2 genes. This should not be surprising since the database is a seed database used in Morph where the seed patterns are of the form shown in the schemas in Figure 2.35. The codes should be as small as the patterns they encode in a database of patterns. Instantiating a variable in a term code of a chess graph reflects restricting Piece to a specific piece type in WhitePawn, BlackPawn, ..., WhiteQueen, BlackQueen in the chess graph. The database A is a database of chess patterns of the form given in Figure 2.34. The variables used in term encoding (#MaxCoAtomIdealMeets column) reflect this.

5.7 Summary

Hierarchical operations such as subsumption, greatest lower bound (GLB), and least upper bound (LUB) are central operations to modern programming languages and knowledge base management systems and hence their efficiency is of primary concern. These operations are normally implemented by encoding hierarchies with bit-vectors and using bit-vector operations on these codes. We have shown that a term encoding we propose can be more efficient than bit-vector encoding methods because of its encoding efficiency for branches.

Bit-vector encoding methods usually do not support GLB operations. We introduced a method using bit-vector codes which efficiently supports GLB operations. The method is based on a bit-vector AND and classifying the resulting code using a decision graph over a subspace of bit-vectors to determine if the GLB operation results in ⊥.
Chapter 6

Compiling and Compressing Conceptual Graphs

Computations involving queries and subpatterns can be shared with superpatterns in a hierarchy. Repeated subpatterns in superpatterns can be removed in a hierarchy of patterns resulting in compression. Formation rules applied to subpatterns to generate superpatterns may be implemented in an abstract machine resulting in compilation.

6.1 Introduction

This chapter addresses problems in conceptual graph subsumption in a taxonomy. A database of conceptual graphs is typically organised as a directed acyclic graph data structure based on the partial order over conceptual graphs discussed in chapter 2. A content addressable memory is a memory which uses the description of the objects or cells stored as the address in the memory. For example, traditional computer memory is a content addressable memory for bit-vectors. The generalisation hierarchy over conceptual graphs is a content addressable memory where the conceptual graphs are addresses. We show how conceptual graphs in this hierarchy can be compiled into instructions which represent specialised cases of the canonical formation rules. This method compiles subsumption of conceptual graphs and compresses knowledge in a knowledge base. Each conceptual graph in the hierarchy is compiled as differences from its parent graphs. The differences represent the rules
used in deriving the graph from the parent graphs. We describe some experiments to illustrate how the method compresses knowledge bases.

Garner and Tsui [GT87] first used the idea of storing conceptual graphs as the differences between parent graphs in the hierarchy. This has the potential to save store. In [GT87], a graph is reconstructed from the differences when traversing the hierarchy. This graph is then compared to the query by using a general subsumption algorithm. Our method differs from this in three important ways. Firstly, we use a different concept of differences. Rather than labelling the incident arcs with the differences, we label the node representing the conceptual graph. Secondly, the differences are between a graph and all of its parent graphs, rather than a single parent graph. This method is especially suited to the topological search method proposed by Levinson [Lev92] and discussed in chapter 2. Thirdly, the differences between graphs represent instructions which are specialised cases of the canonical formation rules of conceptual graph theory. These instructions perform part of the matching of the database graph with the query graph. The canonical formation rules are the basis of the partial ordering defined over conceptual graphs. Rather than just reconstructing the graphs, the differences are applied to the query graph using the projections of the parent graphs into the query graph. In many cases if the parent graphs have already been compared to the query, the differences need only be mapped into the query graph to implement the comparison.

Our method achieves compilation of conceptual graphs in three ways: removal of redundant data, use of simple instructions which ignore redundant checks when performing matching, and by sharing common processing between graphs. Section 6.4 introduces basic conceptual graph theory. Section 6.6 explains what we mean by compilation of conceptual graphs in the generalisation hierarchy and section 6.7 gives descriptions of instructions which are specialised cases of the canonical formation rules. These ideas are used to compile a small example database and a query on the compiled database is examined. Section 6.8 details experiments on knowledge base compression and discusses the impact on compilation.
6.2. Compressing objects in hierarchies

Using image databases, we illustrate how objects can be compressed in hierarchies for efficient storage. Consider the set of images in Figure 6.1. A hierarchy is induced by the subimage relationship or subtree relationship over quadtreees, as in Figure 6.2. The images $a$ and $b$ are subimages of the query $u$. The image $d$ is a superimage of $u$. The image $c$ is neither a subimage nor superimage of $u$, but it does contain the subimage $b$ in common with $u$. The image $u$ could be seen as the union or join of $a$ and $b$. The image $d$ is the union of $a$, $b$ and $c$. Each of these relationships can be determined by a subimage comparison. A typical query on such an image database is to find all images which are "like" the query image $u$. These could be subimages of $u$, or superimages of $u$, or images which contain some image parts in common with $u$.

Hierarchical search will minimise the number of images compared, but it is also
possible to reduce the amount of computation done in comparing images. If the images in a database share common subimages it is possible to remove the common images, so that the images are compressed and common computation is shared rather than duplicated. Figure 6.3 shows how the set of images in Figure 6.2 can be compressed by storing each image as the difference between itself and its parent objects. An image hierarchy can be seen as a kind of inheritance hierarchy where each image is the union of its ancestor images and its own additional subimage.

When inserting an image into a hierarchy, if its subimages do not already occur in the database, then these subimages can be added to the hierarchy. This results in further compression, since it encourages sharing of subimages. The images in Figure 6.3 can be further compressed through such a reorganisation of the database as in Figure 6.4. The compression operation, which computes the differences between an image and its parent subimages, removes the repeated subimages. In Figure 6.4, the images a and b have the square in image c in common.

Indexing can be improved by encouraging that enough subimages of an image are in the hierarchy to reconstruct the image entirely from its subimages. This ensures that during classification of a query image, database images are only compared if all the component subimages of the database image are present in the query image. In Figure 6.4, the image f is a subimage of c in Figure 6.3, and is the difference between c and its parent b. The image c can be completely reconstructed by joining images e, b and f. Similarly g is a subimage of d in Figure 6.3 and is the difference
Figure 6.4: Storing images as the difference between adjacent subimages

between d and its parents a and c.

To uncompress an image the differences of the subimages of the object are joined together by a walk of the ancestors of the image. Not only have the images been compressed, the computation has also been compressed, since only the differences need to be compared to the query.

Complex images including those with fractal geometries can be defined as regular languages which can be encoded in Probabilistic Finite Generators (PFGs) [ID93]. PFGs are a kind of finite state machine. Sowa has shown how conceptual graphs can represent finite state machines [Sow93].

In chapter 1 we showed how a hierarchy can be used to store complex images such as fractals. Figure 1.3 gave a hierarchy of fractal images. Further, each image can be represented as a Probabilistic Finite Generator (PFG) using Culik II & Dube's technique [ID93]. Figure 1.4 gave the corresponding hierarchy of PFGs. Each PFG is a compressed representation of its corresponding image, but Figure 6.5 shows that the PFGs can be compressed further by considering the redundancy in the images (and PFGs) with respect to the hierarchy of images (hierarchy of PFGs), respectively. Only new states and transitions used in refining and joining the PFGs are stored. In Figure 6.5, for example, the sparkling diamond can be produced by adding a transition from state 7 to state 0 of the diamond PFG. The triangle and diamond combination can be produced by creating a new start state and adding transitions from the new start state to the start states of the triangle and sparkling
Figure 6.5: Hierarchy of compressed probabilistic finite generators in Figure 1.4

diamond PFGs. These methods of composing images by copying, restricting and
joining PFGs has a correspondence with the graph grammar rules of conceptual
graphs called the *canonical formation rules*. In this chapter we discuss ways of
compiling and compressing conceptual graphs based on representing graphs as the
set of formation rules applied to their subobjects.

### 6.3 A modeling language for complex objects

The conceptual graph knowledge representation aims to express the propositional
content of natural language sentences in as simple and direct a manner as possible
[Sow91]. This section is a sketch of the conceptual graphs language. Sowa’s con-
ceptual graphs [Sow84] have attracted applications including natural language process-
ing, information retrieval, systems analysis and data modeling, software specification
and synthesis, image analysis, planning, and expert systems [Sow92]. Collections
of papers on conceptual graphs include [ELR95, TDS94, MMS93, NNG92, PN93,
Way91, EG90, NN89, Esc88].

Conceptual graphs [Sow84] is a system of logic based on the graphical system
of logic, *existential graphs*, developed by the nineteenth century philosopher and
logician Charles Sanders Peirce’s (1839-1914) [Pei60, Rob73]. After making sig-
nificant contributions to the linear form of logic, Peirce developed his graphical form which he called “the logic of the future”. The complex object storage and retrieval techniques discussed in this thesis are the foundation of the *Peirce* system [EL92a, EL92b, LAML93, EL94b, ELR94] named in his honour. The Peirce system is the beginnings of a realisation of C.S.Pierce’s logic in a computer.

Sowa introduced Peirce’s Existential Graphs to the Artificial Intelligence community [Sow84], adding sorts and abstraction to the language. Conceptual graphs [Sow93] are a synthesis of Peirce’s existential graphs [Rob73] with Tesnière’s dependency graphs [Tes59] and the semantic networks of Artificial Intelligence. Conceptual graphs have the full power of first-order logic, can represent modal and higher-order logic, and have simple and elegant inference rules. Sowa [Sow84] defines an operator $\phi$ which maps conceptual graphs into first-order logic. Using this mapping Sowa gave the semantics of the language and proved the soundness and completeness of conceptual graphs.

Conceptual graphs is an arguably intuitive notation for first-order logic. In this thesis we argue that the graphical nature of the logic formalised in the generalisation operator used for constructing generalisation hierarchies is an advance which allows complex objects to be treated as first-class in the theory and in implementations. Conceptual graphs has an informal mapping into natural language. For example the sentence “A girl is eating fast” could be represented by the conceptual graph

$$[\text{Girl}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Manner}) \rightarrow [\text{Fast}]$$

which could in turn be translated into first-order logic as

$$\exists x \bullet \exists y \bullet \exists z \bullet \text{Girl}(x) \land \text{Agent}(y, x) \land \text{Eat}(y) \land \text{Manner}(y, z) \land \text{Fast}(z).$$

The first-order logic representation contains four conjunctions, whereas the original sentence and the conceptual graph contain none. The differences in ease of mapping to natural language become even more evident for more complex examples including modal operators and plurals. Alternatively, conceptual graphs can be mapped into a modern typed logic

$$\exists x : \text{Girl}, y : \text{Eat}, z : \text{Fast} \bullet \text{Agent}(y, x) \land \text{Manner}(y, z).$$
However, even in this more precise representation there is a conjunction. This ability of conceptual graphs to represent information without using logical connectives is important when it comes to implementation, since pattern matching can be used rather than logical deduction. The point is that a collection of conjunctive facts can be considered as one pattern. This is not unique to conceptual graphs, but conceptual graphs emphasis on viewing a conjunctive collection of predicates as a graph, emphasises that these collections can be analysed as a pattern, and can be indexed and encoded using the methods discussed in this thesis.

6.3.1 Relevance of conceptual graphs to broader research

Conceptual graphs are a graphic system of logic that is as general as predicate calculus, but they are as readable as special purpose diagrams [Sow93]. In 1896, Charles Sanders Peirce developed existential graphs [Rob73] as an alternative to the linear notation for predicate calculus. Sowa [Sow84] combined existential graphs with modern research in semantic networks to develop conceptual graphs. An annual conference on conceptual structures is dedicated to discussing research issues and applications of conceptual graphs. There is also a conceptual graph mailing list with over 300 subscribers.

Sowa [Sow93] showed that many of the existing popular diagrams can be viewed as special cases of conceptual graphs: type hierarchies, entity-relationships diagrams, parse trees, dataflow diagrams, flow charts, state-transition diagrams, and Petri nets. The ability of conceptual graphs to subsume almost all conventional special-purpose diagrams make it a unified diagramming tool for research in many areas of computer science.

The ANSI X3H4 committee on Information Resource Dictionary Services (IRDS) has proposed conceptual graphs as a standard normative schema language for integrating heterogeneous modelling languages, modelling paradigms, and database languages [Pe93]. The ANSI X3T2 Data Interchange and Repositories committee is using the Knowledge Interchange Format (KIF) report [GFB+92] as a base document. Developers of KIF, and conceptual graphs have agreed to adopt the KIF model-theoretic foundations as common semantics. The ANSI X3J21 Formal
Description committee will examine the common logical foundations for Vienna Definition Methodology (VDM) and the Z specification language in order to establish a common basis to be shared with KIF and conceptual graphs.

What else does conceptual graphs offer that traditional first-order logic does not? Conceptual graphs could be discarded in favour of first-order logic with no loss of generality. However, as Shastri [Sha91] argues, a knowledge representation should not only be judged by its representational adequacy but also by its computational effectiveness.

An important feature of conceptual graphs is its ability to structure knowledge. In conceptual graphs the representation of the sentence above would be treated as a single unit (conceptual graph). Conceptual graphs distinguish the conjunctions inside a proposition and conjunctions connecting propositions. Consider the representation of the sentence “A girl is eating fast, and a dog is barking” in conceptual graphs and typed first-order logic:

\[
\begin{align*}
&[\text{Girl}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \leftarrow (\text{Manner}) \leftarrow [\text{Fast}] \\
&(\exists x : \text{Girl}, y : \text{Eat}, z : \text{Fast} \cdot \text{Agent}(y, x) \land \text{Manner}(y, z)) \land \\
&(\exists u : \text{Dog}, v : \text{Bark} \cdot \text{Agent}(v, u)).
\end{align*}
\]

A conceptual graph corresponds to a formula in typed first-order logic that is connected by variables or constants. Conjunction of conceptual graphs is represented by grouping them together.

More dramatic is the ability to abstract complex information into single concepts or relations or individuals. This allows concepts to be deeply modelled while avoiding computation with the detail by working at a higher level of abstraction.

Semantic networks are kinds of logics, just as typed logic is a kind of logic. Typed logic could be considered a trivial variant of first-order logic. There is a simple mapping from typed logic to first-order logic. However, the elegant formulation of problems, and computational effectiveness of typed logic are remarkable. The introduction of sorts (types) into first-order logic can significantly reduce the number of clauses, connectives, and variables in the formulation of a knowledge base. Modified inference rules that use the sortal constraints of arguments prune the search space. The branches that are pruned are useless dead ends. Walther
[Wal84] employed typed logic to give the first mechanical solution to the theorem prover challenge problem:

Schubert’s Steamroller: Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore there is an animal that likes to eat a grain-eating animal.

The success was attributed to the significant reduction in the size of the problem by the typed logic formulation and the correspondingly reduced search space. Figure 6.6 gives a formulation of this problem using conceptual graphs.

We argue that a parallel situation exists between conceptual graphs and typed logic. Conceptual graphs inherit the computational effectiveness of typed logic through the use of sorts (concepts types). As well as sorts, conceptual graphs allows these sorts to be defined intentionally. This allows the sort hierarchy to be deeply modelled, which is useful for early detection of inconsistencies and in the generation of explanations. Further, by abstracting information away, a problem can be greatly reduced in size.

There have been many semantic network systems built in the KL-ONE tradition [Bra77] such as BACK [Pel91], CLASSIC [PSMB+91], KL-TWO [Vil85], K-Rep [MDW91], KRYPTON [BGL85], KRIS [BH91], LOOM [Mac91], NIKL [SM91], SB-ONE [Kob91]. Implementations of semantic networks differ from implementations of first-order logic such as Prolog in at least one major respect: a taxonomy of concepts. With the advent of constraint languages such as LIFE [AKP93], and TAXLOG [Han91], that distinction is disappearing. We argue that what the Prolog-like implementations lack is an associative memory, where assertions are not decomposed but stored as a single object.

Conceptual languages generally include two different languages: a terminological
Figure 6.6: An axiomatisation of Schubert’s steamroller
language (T-Box) for defining concepts intentionally; and an assertional language (A-Box) for making assertions about those concepts. The ordering between two concepts is defined by a subsumption operation. A taxonomy is a construction of the non-transitive subsumption relationship between a collection of concepts. Adding a new concept to a taxonomy is called classification, that is, using the insert operation described in chapter 3.

Conceptual graphs uses the same language for both defining concepts (as well as relations and individuals) and making assertions. Thus subsumption not only defines a partial ordering over concepts in the form of a taxonomy, it takes this process one step further by defining a partial ordering over assertions. The idea of a taxonomy of concepts generalises to a generalisation hierarchy of assertions. The subsumption relationship for atomic conceptual graphs is defined by a subgraph relationship modulo subtyping and individuation. For general conceptual graphs a theorem prover is used to compare formulas.

The generalisation hierarchy is an associative memory. Consider a knowledge base as a collection of sentences. Each sentence is represented by a conceptual graph. The sentences are ordered into a generalisation hierarchy which is a directed acyclic graph representing the non-transitive links in the subsumption relationship defined over conceptual graphs. A query (conceptual graph) may be made on this collection. If we regard solutions to a query as sentences that imply the query sentence, then these can be found by a two phase constrained topological search of the generalisation hierarchy (classification). The search returns the virtual location of the query graph: parents and children of the query graph. The subhierarchies of the children of the query graph contain all of the solutions to the query in an ordered form. Chapters 1 and 2 survey complex object classification methods which significantly prune the number of object comparisons. Construction of the generalisation hierarchy statically compiles many inferences.
6.4 What are Conceptual Graphs?

A conceptual graph [Sow84] is a finite, connected, bipartite graph. The two kinds of
nodes are concepts and conceptual relations. Every conceptual relation has one or
more arcs, each of which must be linked to some concept. A single concept by itself
may form a conceptual graph, but every conceptual relation must be linked to some
concept.

The function type maps concepts into a set T whose elements are type labels.
The function referent maps concepts into a set I = {#1, #2, #3, ...} of individual
markers or the generic marker *. An individual marker is a surrogate for some
individual in the real world, a perceived world, or a hypothetical world. The label
of concept c, lab(c), is the pair type(c) : referent(c). This concept may be displayed
in the linear form as [t : r], where type(c) = t and referent(c) = r. For example,
the concept [Person: *] or more simply [Person] represents an unspecified person,
and may be read A person. A box replaces the square brackets in the graphical
form.

The partial order \( \sqsubseteq \) over the type labels in T, known as the type hierarchy,
forms a lattice, called the type lattice. The type hierarchy makes analytic statements about
types: they must be true by intension. The statement Girl \( \sqsubseteq \) Person is true, because
the properties of a person are also associated with a girl.

The minimal common supertype of a pair of type labels s and t is written \( s \sqcup t \).
The maximal common subtype is written \( s \sqcap t \). There are two primitive type labels:
the universal type \( \top \) and the absurd type \( \bot \). For any type label t, \( \bot \sqsubseteq t \sqsubseteq \top \). In
Figure 6.11, the minimal common supertype of Person and Pie is PhysicalObject.
The maximal common subtype of Animate and PhysicalObject is Animal. The
maximal common subtype of Person and Pie is \( \bot \) (absurd), which means that it is
logically impossible for an entity to be both a person and a pie.

The denotation of type t, written \( \delta t \), is the set of all entities that are instances
of any concept of type t. For extensions, the union \( \delta \text{Person} \sqcup \delta \text{Pie} \) is the set of all
people and pies in the world and nothing else. For intensional type labels, Person
\( \sqcup \) Pie is their minimal common supertype PhysicalObject, which also has subtypes
Animal, Food, and Chair. The type lattice represents categories of thought, and the
lattice of sets and subsets represents collections of existing things. The two lattices are not isomorphic, and the denotation operator that maps one into the other is neither one-to-one nor onto.

The function type also maps conceptual relations to type labels. A relation \( r \) with \( \text{type}(r) = t \) may be written \( (t) \) in the linear form. A ellipse replaces the parenthesis in the graphical form. For two relations to have the same type they must have the same number of arcs. Concepts and conceptual relations have no type in common.

The conformity relation :: relates type labels to individual markers: if \( t :: i \) is true, then \( i \) is said to conform to type \( t \). The conformity relation obeys the following conditions:

- The referent of a concept must conform to its type label: if \( c \) is a concept, \( \text{type}(c) :: \text{referent}(c) \). For example the concept \([\text{Integer}: 1]\) is well-formed, but \([\text{Integer}: 3.14]\) is not.

- If an individual marker conforms to type \( s \), it must also conform to all supertypes of \( s \): if \( s \sqsubseteq t \) and \( s :: i \), then \( t :: i \). For example the number 3 conforms to the type Prime, Prime::3. Hence it also conforms to the supertype Integer, Integer::3.

- If an individual marker conforms to type \( s \) and \( t \), it must also conform to their maximal common subtype: if \( s :: i \) and \( t :: i \), then \( (s \cap t) :: i \). For example since 3 conforms to types Odd and Prime: Odd::3; Prime::3, then 3 also conforms to their maximal common subtype OddPrime, OddPrime::3.

- Every individual marker conforms to the universal type \( \top \); no individual marker conforms to the absurd type \( \bot \): for all \( i \) in \( I \), \( \top :: i \), but not \( \bot :: i \).

- The generic marker * conforms to all type labels: for all type labels \( t \), \( t :: * \).

The operator \( \phi \) maps conceptual graphs into formulas in first order predicate calculus. For the conceptual graph

\[
\text{u} = [\text{Girl: #}\text{3074}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Object}) \rightarrow [\text{Pie}]
\]

the translation is

\[
\phi \text{u} = \exists x \exists y (\text{Girl}(\#3074) \land \text{Agent}(x, \#3074) \land \text{Eat}(x))
\]
\( \land \text{Object}(x,y) \land \text{Pie}(y) \)\)

Generic concepts map to variables, individual concepts map to constants. Alternatively, conceptual graphs could also be mapped into a modern typed logic

\[ \phi' u = \exists x : \text{Eat}, y : \text{Pie} \bullet \text{Agent}(x, \#3074) \land \text{Object}(x,y) \]

### 6.4.1 Canonical Graphs

To distinguish the meaningful graphs that represent real or possible situations in the external world, certain graphs are declared to be canonical. Canonical graphs include basis graphs and those graphs derived by canonical formation rules from basis graphs.

There are five canonical formation rules for deriving a conceptual graph \( w \) from conceptual graphs \( u \) and \( v \) [Sow84, MC93]:

- **copy**\((u)\). \( w \) is an exact copy of \( u \).

- **restrict**\((u, c, l)\). For any concept \( c \) in \( u \), \( \text{type}(c) \) may be replaced by a subtype \( t \); if \( c \) is generic, its referent may be changed to an individual marker \( i \) where \( l = t : i \). These changes are permitted only if \( \text{referent}(c) \) conforms to \( \text{type}(c) \) before and after the change, that is \( t :: i \).

- **simplify**\((u, r, s)\). If conceptual relations \( r \) and \( s \) in the graph \( u \) are duplicates, then one of them may be deleted from \( u \) together with all its arcs.

- **Join**\((u, c, d)\). If a concept \( c \) in \( u \) is identical to a concept \( d \) in \( u \), then \( \text{Join}(u, c, d) \) is the graph obtained by deleting \( d \) and linking to \( c \) all arcs of conceptual relations that had been linked to \( d \).

- **Fuse**\((u, v, c, d)\). Let \( u \) and \( v \) be two disjoint conceptual graphs. If a concept \( c \) in \( u \) is identical to a concept \( d \) in \( v \), then \( \text{Fuse}(u, v, c, d) \) is the graph obtained by deleting \( d \) and linking to \( c \) all arcs of conceptual relations that had been linked to \( d \).

To illustrate the formation rules, Figure 6.7 shows two canonical graphs. Each concept and relation is identified with \( ci \) and \( rj \), respectively, and each graph is also
labeled for reference in the text. The graph b may be read A girl is eating fast; and the graph c, A person, Sue, is eating pie. These are not formal translations of the graphs, but informal verbalisations for discussion of the graphs here.

The graph d in Figure 6.8 shows the result of restricting the concept c3 [Girl] in the graph b in Figure 6.7 to [Girl: Sue]. The graph e is the result of restricting the type Person in the concept c7 in graph c to type Girl. Before doing the restrictions, the conformity relation must be checked to ensure that Girl :: Sue is true.

The identical concepts c3 and c7 [Girl: Sue] in d and e, respectively, in Figure 6.8 can be fused together to form a single graph f in Figure 6.9. Then the identical concepts c1, c5 [Eat] in f can be joined together to produce g.

In Figure 6.9, the graph g can be simplified by removing one of the duplicate

Figure 6.7: Two canonical graphs

Figure 6.8: Restriction of two graphs in Figure 6.7

Figure 6.9: Join of the two graphs in Figure 6.8
relations $r_2$ and $r_4$ (Agent) resulting in graph $h$ in Figure 6.10. Two conceptual relations of the same type are duplicates if for each $i$, the $i$th arc of one is linked to the same concept as the $i$th arc of the other. The graph $h$ may be read *A girl, Sue, is eating pie fast.* The simplification rule corresponds to the rule of logic that $R(x, y) \land R(x, y)$ is equivalent to just $R(x, y)$.

The formation rules are a kind of graph grammar for canonical graphs. Besides defining syntax, they also enforce certain semantic constraints. The formation rules make no guarantee about truth or falsity. However, the formation rules are refutation rules. If we assert that the graph

$$[\text{Person}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Object}) \rightarrow [\text{Pie}]$$

is false (*No person is eating a pie*), then we can use the formation rules to show that

$$[\text{Girl:Sue}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Object}) \rightarrow [\text{Pie}]$$

is false (*Sue is not eating a pie*). That is, if a graph can be derived from a false graph, then it must in turn be false. The formation rules are falsity preserving.

The canon contains the information necessary for deriving a set of canonical graphs. It has four components: a type hierarchy $T$, broken into a concept hierarchy
$T_c$ and relation hierarchy $T_r$; a set of individual markers $I$; a conformity relation $::$ that relates labels in $T$ to markers in $I$; and a finite set of conceptual graphs $B$, called the canonical basis, with all type labels in $T$ and all referents either $*$ or markers in $I$. The canonical graphs are the closure of $B$ under the canonical formation rules. Figure 6.11 shows the canon used in this thesis.

6.4.2 The Relationship between Formation Rules and Subsumption

If a conceptual graph $u$ is canonically derivable from a conceptual graph $v$ (possibly with the join of other conceptual graphs $w_1, \ldots, w_n$), then $u$ is called a specialisation of $v$, written $u \subseteq v$, and $v$ is called a generalisation of $u$. The generalisation relationship $\subseteq$ is a partial order over conceptual graphs. Thus $\subseteq$ is the fundamental operation on conceptual graphs which can be used to manage databases of conceptual graphs using the abstract data type for partially ordered sets discussed in chapters 1, 2, 3 and 4.

Generalisation defines a partial ordering of conceptual graphs called the generalisation hierarchy. The ordering is reflexive, transitive, and antisymmetric. For any conceptual graphs $u$, and $v$, the following properties are true:

- **Subgraph.** If $v$ is a subgraph of $u$, then $u \subseteq v$.

- **Subtypes.** If $u$ is identical to $v$ except that one or more type labels of $v$ are restricted to subtypes in $u$, then $u \subseteq v$.

- **Individuals.** If $u$ is identical to $v$ except that one or more generic concepts of $v$ are restricted to individual concepts of the same type, then $u \subseteq v$.

- **Top.** The graph $[\top]$ is a generalisation of all other conceptual graphs.

The graphs in Figs. 6.7, 6.8, and 6.9 are all generalisations of the graph in Figure 6.10. We call the graphs defined so far atomic conceptual graphs (ACGs). They do not contain logical connectives and hence neither quantification other than the default existential quantification, nor have we considered definition of concepts
and relations. A subsumption test for ACGs can be implemented as subgraph morphism modulo subtyping and individuation.

The generalisation hierarchy is not a partial order over conceptual graphs as stated in [Sow84], rather it is a partial order over equivalence classes of conceptual graphs. Consider the graphs

\[ v = [\text{Person}] \langle \text{Agent} \rangle \rightarrow [\text{Eat}] \]

\[ u = [\text{Person}] \langle (\text{Agent})\rightarrow [\text{Eat}]\rightarrow (\text{Agent}) \rightarrow [\text{Person}] \]

The graph \( v \) is a proper subgraph of \( u \). The graph \( u \) can be derived from \( v \) by joining a copy of \( v \) on the concept \([\text{Eat}]\), thus \( u \subseteq v \). However, \( v \) can be derived from \( u \) by joining the two identical concepts \([\text{Person}]\), then simplifying the duplicate (\( \text{Agent} \)) relations, thus \( v \subseteq u \). Hence \( u = v \) according to the generalisation hierarchy. This property of the generalisation hierarchy has also been noted independently in [Wil90]. Mugnier and Chein [CM92] named these minimal graphs of an equivalence class \textit{irredundant graphs}, since they are graphs which do not have a proper subgraph onto which the graph can be projected. This is a subtle point that doesn’t affect any of the subsequent theory of conceptual graphs. In practice, graphs with redundant branches can always be simplified to derive the smallest one in each equivalence class. Conceptual graphs from now on are assumed irredundant.

If \( u \subseteq v \), a canonical derivation of \( u \) from \( v \) corresponds to the reverse of a proof of the formula \( \phi v \) from the formula \( \phi u \): for any conceptual graphs \( u \) and \( v \), if \( u \subseteq v \), then \( \phi u \supseteq \phi v \). The result that the two graphs given in the paragraph above are equivalent should not be surprising considering their translations into typed logic

\[ \phi' v = \exists x : \text{Person}, y : \text{Eat} \bullet \text{Agent}(y, x) \]

\[ \phi' u = \exists x : \text{Person}, y : \text{Eat}, z : \text{Person} \bullet \text{Agent}(y, x) \land \text{Agent}(y, z) \]

For any conceptual graphs \( u \) and \( v \) where \( u \subseteq v \), there must exist a mapping \( \pi : v \rightarrow u \), where \( \pi v \) is a subgraph of \( u \) called a \textit{projection} of \( v \) in \( u \). The \textit{projection operator} \( \pi \) has the following properties:

- For each concept \( c \) in \( v \), \( \pi c \) is a concept in \( \pi v \) where \( \text{type}(\pi c) \subseteq \text{type}(c) \). If \( c \) is individual, then \( \text{referent}(c) = \text{referent}(\pi c) \).
Figure 6.12: A query graph $u$

- For each conceptual relation $r$ in $v$, $\pi r$ is a conceptual relation in $\pi v$ where $type(\pi r) = type(r)$. If the $i$th arc of $r$ is linked to a concept $c$ in $v$, the $i$th arc of $\pi r$ must be linked to $\pi c$ in $\pi v$.

For example, the projection of graph $c$ in Figure 6.7 into the graph $u$ in Figure 6.12 is \{d5 \rightarrow d4, c6 \rightarrow d3, c7 \rightarrow d1, r3 \rightarrow q3, r4 \rightarrow q1\}, since Person > Girl and Act > Eat.

A graph $v$ can be represented by the set of instances of the canonical formation rules used to construct the graph $v$ from the graphs $w_1, \ldots, w_n$. To test if the graph $v$ subsumes a graph $u$, these rule instances can be applied to the projections of $u$ in $w_1, \ldots, w_n$. The rule instances will only succeed if $v$ subsumes $u$. We use this technique to compile conceptual graphs in a data structure representing the generalisation hierarchy partial ordering.

Now that we have the theory of atomic conceptual graphs we will consider how to store large sets of conceptual graphs and how to retrieve conceptual graphs once stored.

### 6.5 Storing and Retrieving Conceptual Graphs?

The common data structure used to store conceptual graphs is the generalisation hierarchy (discussed in chapter 2); a directed acyclic graph representing the non-transitive links of the partial ordering over conceptual graphs [Min89, Lev92, Ell90a, Ell90b]. Levinson’s earlier work used a similar data structure for organising chemical graphs [Lev85]. The taxonomy over KL-ONE concept descriptions [SL83] is also a hierarchy.
6.5. The Generalisation Hierarchy as a Contents Addressable Memory

The nodes in the generalisation hierarchy are conceptual graphs and the arcs represent the non-transitive ordering between the graphs. In Figure 6.13 the hierarchy is given for the graphs from the previous section. The canonical basis in this example consists of the set of graphs \{B_1, B_2, B_3\}. The arc (b, d) indicates each of the following:

1. A girl, Sue, is eating fast is canonically derivable from A girl is eating fast;

2. A girl, Sue, is eating fast implies A girl is eating fast;

3. A girl is eating fast is a generalisation of A girl, Sue, is eating fast; and

4. A girl, Sue, is eating fast is a specialisation of A girl is eating fast.

In the following sections we examine how to use the hierarchy for searching a set of conceptual graphs and how to construct the hierarchy.
6.5.2 Searching in the Generalisation Hierarchy

The generalisation hierarchy indexes the knowledge base. When we apply a conceptual graph query \( u \) to the knowledge base we search for \( u \) in the hierarchy. The hierarchy is a content addressable memory where the conceptual graphs are the addresses.

Consider a depth-first search of the generalisation space. Any path in the generalisation space can be taken as they all lead to \( u \) in the hierarchy. Consider searching for the query \( u \)

\[
[\text{Girl: Sue}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Object}) \rightarrow [\text{Fast}]?
\]

*Is the girl, Sue, eating fast?*, in the hierarchy in Figure 6.13. The query \( u \) matches the graph \( d \) in the hierarchy. The search starts at the graph \([\top]\). To find the graph we search the children of \([\top]\) for a generalisation of \( u \). The *children* of \( v \) is the set of immediate specialisations of \( v \). The children of \( a, B_1, B_2, B_3, b, c, d, e, \) and \( h \) are \( \{B_1, B_2, B_3\}, \{b\}, \{b, c\}, \{c\}, \{d\}, \{e\}, \{h\}, \{h\} \) and \( \bot \), respectively.

In a depth-first search we could select the first graph in the children which is a generalisation of the query \( u \) as a continuation in the path to \( u \). The basis graph \( B_1 \) is a generalisation in the first children set, so we select it. So are \( B_2 \) and \( B_3 \), so they could equally be chosen. There is only one child of \( B_1, b \), which is also a generalisation, so we select it. We now search the children of \( b \) for a generalisation. The graph \( d \) is the only child and it is a generalisation of \( u \). In fact \( d \) is isomorphic to \( u \). The query graph is matched so the search terminates successfully. In this case there are two solutions \( \{d, h\} \). The English answers to the question *Is the girl, Sue, eating fast?* are \( d \) - yes; and \( h \) - yes, Sue is eating pie fast.

In [Lev92] and chapter 2, topological search was shown to be better than depth-first search for classification of objects. This is also true for classification of conceptual graphs. The two phase search corresponds to finding the immediate generalisations of query conceptual graph \( u \) and the immediate specialisations of \( u \). Topological search enforces the rule that no conceptual graph in the database is compared before any generalisations of it in the database.

The search does not necessarily start from the graph \([\top]\). Indexing techniques
can be used to start further down the generalisation space. The ultimate goal of indexing techniques is to index directly to the top of the specialisation space which includes u (see [LE92] for indexing techniques).

### 6.5.3 Inserting in the Generalisation Hierarchy

To insert a graph u into the hierarchy we need to compute the set of immediate generalisations and the set of immediate specialisations of u in the hierarchy. This information gives us the virtual location for inserting u.

Consider inserting the graph \( u = [\text{Person}] \leftarrow (\text{Agent}) \leftarrow [\text{Eat}] \rightarrow (\text{Object}) \rightarrow [\text{Pie}] \)

read *A person is eating pie*, into the hierarchy in Figure 6.14. The immediate generalisations in this case are \( b \) - *A person is eating*, and \( c \) - *A pie is being eaten*. The immediate specialisations are \( h \) - *A girl, Sue, is eating pie in the kitchen*, and \( e \) - *A girl is eating pie fast*. Notice that \( d \) - *A person is eating food in the kitchen*, and \( u \) are incomparable. To insert u the arcs \((b, e), (c, e), \) and \((c, h)\) are removed, then the arcs \((b, u), (c, u), (u, h)\) and \((u, e)\) are added to get the new hierarchy in Figure 6.15.

We have shown how to prune the search within the database down to the generalisation and specialisation space. In the following section we show how to share matching information gained from subsumption testing between related graphs.
6.6 Compilation in the generalisation hierarchy

How can the efficiency of querying the database be improved? In chapters 1, 2 and 4 we saw a method for minimising the number of database graphs compared to the query graph. In the following sections we look at minimising the cost of each of these comparisons. We examine how to represent conceptual graphs in a generalisation hierarchy to improve individual subsumption tests.

Woods [Woo91] states about his algorithm “No deep insights have been exploited to gain efficiency. For example, in classification, no advantage is taken of what might be learned in the course of one subsumption test that might be redundant with part of another subsumption test.”

Garner and Tsui [GT87] proposed representing graphs as differences between parent graphs in the generalisation hierarchy. Figure 6.16 illustrates how they stored the graph $v = “A$ girl, Sue, eating pie fast” as the difference from the adjacent generalisation $u = “A$ girl eating food”. The difference $c_1 \rightarrow “: Sue”$ means replace the referent of the concept $c_1$ in $u$ with Sue. The difference $c_2 \rightarrow “(Manner)\rightarrow [Fast]”$ means to connect a new binary relation (Manner) to the concept $c_1$ in $u$ and a new concept [Fast]. The difference $c_3 \rightarrow “Pie:”$ means replace the type of the concept $c_3$ in $u$ with the type Pie.

A difference between Garner and Tsui’s method and the method outlined below is that the former method places the difference between parent graphs on the incident
arc, whereas the latter places differences between a graph and all its immediate generalisations in the node representing the graph. The graph differences in Garner and Tsui’s method are the nodes, arcs, and restrictions in the specialisation that are not in the generalisation. The graph differences are treated as data. Graphs are reconstructed by an algorithm which reads the differences by traversing the arcs. Reconstructed graphs are then compared with the query using a general matching algorithm. This method does not compile the graphs into matching instructions, nor does it share common computation in queries.

Storing graphs as differences fulfills our aim of removing redundant data from the database. Another aim of our method is to share common computation. This can be achieved by storing projections between the parent graphs, in conjunction with the differences. As we will see the projections do not have to be stored explicitly, but can be composed. This allows us to fulfill another aim: to represent the differences in such a way that they may be used as instructions for a future conceptual graph abstract machine. A graph may be compared to a query using the projections of generalisations into the query, and instructions which perform small parts of the general matching operation relative to these projections. The following details this alternative to Garner and Tsui’s method.

Consider the query $u$ in Figure 6.12 on the generalisation hierarchy in Figure 6.13.
In the discussion below the notation $\pi^{v \rightarrow u}$ represents the projection $\pi^{v \rightarrow u} : v \rightarrow u$, where $v$ and $u$ are graphs. Let us assume that a subgraph morphism $\pi^{b \rightarrow u}$ of the graph $b$ in Figure 6.13 has been found in the query $u$. In the search for solutions to the query $u$, the search method, outlined in previous sections, takes paths through the generalisation hierarchy that contain more specialised generalisations of the query at each step. The graph $d$ is the only choice in paths from $b$. To traverse this path $d$ must be compared to $u$ to see if it is a generalisation of $u$. Can a full subsumption test be avoided? Notice that the only difference between $d$ and $b$ is that the concept $c_3$ with type Girl is restricted from the generic form to the individual Sue.

To compute a match between the database graph $v$ and the query graph $u$ the projection $\pi^{v \rightarrow u}$ must be computed. Assume there is a generalisation $w$ of $v$ stored in the database. Assume that the projection $\pi^{w \rightarrow u}$ is also stored in the database and $w$ has been mapped into the query $u$, $\pi^{w \rightarrow u}$. For every concept and relation $x$ in $v$ if $\pi^{w \rightarrow u^{-1}}x$ is in $w$ then $(x \rightarrow \pi^{w \rightarrow u} \pi^{w \rightarrow u^{-1}}x)$ is in $\pi^{v \rightarrow u}$, otherwise there must be a match for $x$ in $u$, $y$, that does not violate the rest of the match and $(x \rightarrow y)$ is part of the projection $\pi^{v \rightarrow u}$.

To compute a match between $d$ and $u$ in Figure 6.13 the projection $\pi^{d \rightarrow u}$ must be computed. We can use the projection $\pi^{b \rightarrow d} = \{c_3 \rightarrow c_3, r_2 \rightarrow r_2, c_1 \rightarrow c_1, r_1 \rightarrow r_1, c_2 \rightarrow c_2\}$ and $\pi^{b \rightarrow u} = \{c_3 \rightarrow d_1, r_2 \rightarrow q_1, c_1 \rightarrow d_4, r_1 \rightarrow q_2, c_2 \rightarrow d_2\}$. The projection $\pi^{d \rightarrow u}$ is equal to the projection $\pi^{b \rightarrow u}$, since $|b| = |d|$ and the projection $\pi^{b \rightarrow d}$ is the identity mapping. The difference could then be represented as referent($\pi^{d \rightarrow u}c_3) = Sue$. Thus if referent($\pi^{d \rightarrow u}c_3) = Sue$, $d$ is a generalisation of $u$. In general, this is only possible if there is no symmetry in the graphs involved, that is, there are unique projections between the graphs involved. If this is not the case, then the differences must be applied to each of the possible projections. For many of the domains in which conceptual graphs are used the graphs contain unique morphisms.

Thus differences between graphs can be used, if projections between parent graphs ($\pi^{b \rightarrow d}$ in the previous example) and the current generalisation $b$ and the query $u$ ($\pi^{b \rightarrow u}$) are kept. It is not necessary to store the projection between each
parent graph explicitly. The projection are composed when traversing the generalisation hierarchy. The canonical formation rules construct the projection when constructing the graphs. The copy rule sets up a projection of the whole graph that was copied. The restrict rule does not affect the projection. The join rule computes the union of the projections of the two graphs being joined, it then maps one of the identical concepts to the other. The simplify rule maps a duplicate relation onto another.

Conceptual graphs in a generalisation hierarchy can be replaced with sets of applications of the canonical formation rules. The instances of the rules apply to the immediate generalisations of the graph being represented. Figure 6.17 illustrates this method for the generalisation hierarchy in Figure 6.13.

This method has the potential to reduce the cost of graph comparison by sharing computation already done through projections between adjacent conceptual graphs, and also has the potential to save space in storing conceptual graphs.
6.7 Instructions

We now examine how to use the canonical formation rules in differences between parent graphs in the generalisation hierarchy. Here we will concentrate on the first phase of topological search: searching the generalisation space. The following discussion assumes that only irredundant canonical graphs [CM92] are compared and stored.

In the first phase the aim is to find subgraph morphisms of database graphs in the query. In the second phase the aim is to find subgraph morphisms of the query in the database graphs. In the first phase the database graphs could be thought of as reading from the query graph. In the second phase the database graphs write to the query graph constructing specialised solutions. These modes correspond to the modes of reading and writing in Prolog compiler unification instructions [AK91].

Here we give a specialised interpretation of the canonical formation rules based on the mode of operation: read or write. We only examine the read mode here. The graphs are reconstructed by the instructions, however we only show the operations that construct the projection between the database graphs and the query $u$.

- **copy($v, w$)** - Find some subgraph morphism $\pi^{v \to u}$ inducing a subgraph $w$.
  A copy operation is used to separate the variable name space for incomparable specialisations of the copied graph. A general matcher is used to find a subgraph $w$ induced by the projection $\pi^{v \to u}$.

- **restrict($v, c, t, w$)** - if $\text{type}(\pi^{v \to u} c) \subseteq t$ then $\pi^{w \to u} := \pi^{v \to u}$ else fail.
  For the database graph $w$ to be a generalisation of the query graph $u$, $u$ must have a subtype of the type of the corresponding concept in $v$.

- **restrictRef($v, c, i, w$)** - if $\text{referent}(\pi^{v \to u} c) = i$ then $\pi^{w \to u} := \pi^{v \to u}$ else fail.
  This instruction only handles restriction to individual markers, rather than more complex referents, such as nested graphs, and sets. For $u \subseteq v$ to be true the query $u$ must have the same individual marker $i$ as the one in the corresponding concept in the database graph $v$. 
• join\((v, v', c, d, w)\) - if \(\pi^{v \rightarrow u} c = \pi^{v' \rightarrow u} d\) then \(\pi^{u \rightarrow u} := \pi^{u \rightarrow u} \cup (\pi^{v' \rightarrow u} - (d \rightarrow \pi^{v' \rightarrow u} d))\) else fail.

Joining concepts \(c\) and \(d\) of database graphs \(v\) and \(v'\) respectively in read mode means that \(c\) and \(d\) must already be pointing at the same concept in the query graph \(u\).

• simplify\((v, r, s, w)\) - if \(\pi^{u \rightarrow u} r = \pi^{u \rightarrow u} s\) then \(\pi^{u \rightarrow u} := \pi^{u \rightarrow u} - (s \rightarrow \pi^{u \rightarrow u} s)\) else fail.

Simplifying two duplicate relations in a database graph in read mode means that the two relations must be mapped to the same relation in the query, since the query graph cannot contain duplicates as it is a minimal graph.

These instructions can be separated into more specialised cases. For example, if the input and output graph are the same, then a new projection is not constructed, rather modifications to particular entries in the projection are made.

In Figure 6.17, conceptual graphs have been replaced with instructions. Compare this representation with the generalisation hierarchy in Figure 6.13. Figure 6.12 contains the query graph \(u\) and the generalisation hierarchy in Figure 6.13 contains the solution \(h\). Let us consider what happens in each stage of the topological search of the generalisation hierarchy for the query \(u\). We will examine the process in the middle of the search where subgraph morphisms of \(b\) and \(c\) in \(u\) have been found: \(\pi^{b \rightarrow u} = \{c1 \rightarrow d4, c2 \rightarrow d2, c3 \rightarrow d1, r1 \rightarrow q2, r2 \rightarrow q1\}\) and \(\pi^{c \rightarrow u} = \{c5 \rightarrow d4, c6 \rightarrow d3, c7 \rightarrow d1, r3 \rightarrow q3, r4 \rightarrow q1\}\).

Now we look at \(b\) and \(c\)'s children graphs for generalisations of \(u\). The children graphs are \(d\) and \(e\), respectively. The graph \(d\) is represented by \(\text{restrictRef}(b, c3, \text{Sue, d})\). This instruction translates into “if \(\text{referent}(\pi^{b \rightarrow u} c3) = \text{Sue}\) then \(\pi^{d \rightarrow u} := \pi^{b \rightarrow u}\) else fail.” Since \(\pi^{b \rightarrow u} c3 = d1\) and \(\text{referent}(d1) = \text{Sue}\), \(d\) is a generalisation of \(u\) and \(\pi^{d \rightarrow u} := \pi^{b \rightarrow u}\).

The graph \(e\) is represented by \(\text{restrict}(c, c7, \text{Girl, e})\). This instruction is implemented as: if \(\text{type}(\pi^{c \rightarrow u} c7) \subseteq \text{Girl}\) then \(\pi^{e \rightarrow u} := \pi^{c \rightarrow u}\) else fail. Since \(\pi^{c \rightarrow u} c7 = d1\) and \(\text{type}(d1) = \text{Girl}\), \(e\) is a generalisation of \(u\), and \(\pi^{e \rightarrow u} := \pi^{c \rightarrow u}\).

Now we examine the children graphs of \(d\) and \(e\). The only one in this case is \(h\).
Figure 6.18: Some linear derivations of graphs in the hierarchy in Figure 6.13

The graph \( h \) is represented by three instructions. The first instruction, fuse\( (d, e, c_3, c_7, f) \), means if \( \pi^{d-u} c_3 = \pi^{e-u} c_7 \) then \( \pi^{f-u} := \pi^{d-u} \cup (\pi^{e-u} - (c_7 \rightarrow d_1)) \) else fail. Since \( \pi^{d-u} c_3 = d_1 = \pi^{e-u} c_7 \), we calculate \( \pi^{c_9-u} = \{c_1 \rightarrow d_4, c_2 \rightarrow d_2, c_3 \rightarrow d_1, c_5 \rightarrow d_4, c_6 \rightarrow d_3, r_1 \rightarrow q_2, r_2 \rightarrow q_1, r_3 \rightarrow q_3, r_4 \rightarrow q_1\} \).

The second instruction is join\( (f, c_1, c_5, g) \). Since \( \pi^{f-u} c_1 = d_4 = \pi^{f-u} c_5 \) we get \( \pi^{g-u} = \pi^{f-u} \cup (\pi^{f-u} - (c_5 \rightarrow d_4)) \).

The third instruction in \( h \) is simplify\( (g, r_2, r_4, h) \). Since \( \pi^{g-u} r_2 = \pi^{g-u} r_4 = q_1 \) we have \( \pi^{h-u} := \pi^{g-u} - (r_4 \rightarrow q_1) = \{c_1 \rightarrow d_4, c_2 \rightarrow d_2, c_3 \rightarrow d_1, c_6 \rightarrow d_3, r_1 \rightarrow q_2, r_2 \rightarrow q_1, r_3 \rightarrow q_3\} \). Thus \( h \) is a generalisation of \( u \). In fact \( h = u \). Compare this result with the graphs \( u \) and \( h \) in Figure 6.12 and Figure 6.13, respectively.

Figure 6.18 shows an alternative compilation based on canonical derivations applied to a single parent and joining simple basis relations. This approach follows a new formalisation of conceptual graph theory by Mugnier and Chein [MC93] and has some similarities with Garner and Tsui’s method of representation as differences [GT87].

### 6.8 Experiments
Figure 6.19: The generalisation hierarchy Figure 6.13 compressed using differences from Largest Parent

Figure 6.20: The generalisation hierarchy Figure 6.13 compressed using differences from All Parents
Table 6.1: Results of compressing chess files

<table>
<thead>
<tr>
<th>Database</th>
<th>Original File</th>
<th>Lempel-Ziv Compressed</th>
<th>Largest Parent</th>
<th>All Parents</th>
<th>All Parents Without Parents</th>
<th>Lempel-Ziv Compressed All Parents</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>207838</td>
<td>31526</td>
<td>128518</td>
<td>75254</td>
<td>44720</td>
<td>19774</td>
</tr>
<tr>
<td>B</td>
<td>182452</td>
<td>18644</td>
<td>93744</td>
<td>47493</td>
<td>18826</td>
<td>11923</td>
</tr>
<tr>
<td>A with basis</td>
<td>208549</td>
<td>31760</td>
<td>129012</td>
<td>81324</td>
<td>43743</td>
<td>21838</td>
</tr>
<tr>
<td>B with basis</td>
<td>183163</td>
<td>18936</td>
<td>94124</td>
<td>47797</td>
<td>16823</td>
<td>12221</td>
</tr>
</tbody>
</table>

The compilation methods above are still in the design phase. To examine the usefulness of such methods in conceptual graph databases we implemented algorithms which compressed conceptual graph databases by representing a graph \( u \) as the differences between \( u \) and (i) \( u \)'s largest parent, (ii) all \( u \)'s parents. Method (i), which we shall call the *Largest Parent* method, is a best case in Garner and Tsui's method. This is illustrated in Figure 6.19. Method (ii) corresponds to the compilation method shown in Figure 6.20. We will call this the *All Parents* method. The main problem with All Parents method is that a join of all parents may need to be stored. In the tests below we did not store such joins.

The files which were tested for compression were those in Table 2.3 discussed in section 2.9.5 in chapter 2. They are databases “A”, “A with basis”, “B” and “B with basis” consisting of 1778, 1815, 3104, 3146 conceptual graphs, respectively.

The column “Database” of Table 6.1 lists the names of the files. The column “Original File” lists the size of each ASCII file containing the conceptual graphs in a conceptual graph linear notation [Sow84]. The column “Lempel Ziv Compressed” column gives the size of the files when compressed using a UNIX compression utility. The column “Largest Parent” shows the size of each file when compressed using the Largest Parent method of representing conceptual graphs. The column “All Parents” shows the size of each file when compressed using the All Parents method of representing conceptual graphs. The column “All Parents without Parents” stores the same differences as the All Parents method, but leaves out the list of parents which the differences refer to. The column “Lempel Ziv Compressed All Parents” shows the size of the file generated by the All Parents method after compression.
using a UNIX compression utility.

The main columns to compare are “Largest Parent” and “All Parents”. The “All Parents” method results in smaller files, even though more parents are referred to. The cost of referring to the parents is the difference between “All Parents” and “All Parents without Parents” columns. For example, for database A listing parents cost 30534 bytes. In all cases, the “All Parents” method resulted in smaller files than the files using the “Largest Parent” method. The All Parents method resulted in a compression ratio of between 2.56:1 and 3.84:1 in the four knowledge bases we tested. Potentially a similar reduction could also be achieved in information retrieval times.

6.9 Summary

Compilation of conceptual graphs can be achieved by storing them as derivations from immediate generalisations in a directed acyclic graph representing the generalisation hierarchy partial order over conceptual graphs. A graph can be inserted into the generalisation hierarchy by computing its immediate neighbourhood in the hierarchy, then attaching the newly inserted graph to graphs in the neighbourhood. The neighbourhood is computed by a two phase topological search.

The canonical formation rules distinguish conceptual graphs from other semantic network formalisms. They enforce semantic constraints on the kinds of graphs which can be asserted. Algorithms to process them must be developed.

Conceptual graphs are compiled into instructions which are special cases of the formation rules. The instructions operate on immediate generalisations, and construct a projection between the immediate generalisations and the graph, and hence the query graph during search. Common computation involved in matching database graphs to the query graph is shared through these projections. Further, there is a potential for graph compression by storing only the formation rules which are applied to subgraphs, rather than storing the whole graph. Compression of knowledge using differences has been illustrated on some sample databases. Chapter 1 shows how these methods can be applied to complex object hierarchies such as image
Compilation is effected in three ways: removal of redundant data, use of simple instructions which ignore redundant checks when performing matching, and by sharing common processing between graphs.

In future work, we will examine methods for handling complex conceptual graphs for use in such domains as chemistry. Levinson [Lev94] has recently developed a new tuple and skeleton-based compression technique called Universal Data Structure (UDS). UDS is based on a new compact representation of conceptual graphs which make storage and retrieval more efficient. UDS can be extended so that processing in a hierarchical search can be shared.

Early work suggests that storing possible mappings in matrix form between parents and children in the database may be more efficient for graph matching than compilation for complex graphs. The matrices combine mappings between parents and children, and during a query possible matches between database objects and the query are computed in matrix form. The matrices for possible bindings between parents and a child are combined with matrices between parents and the query using matrix multiplication to give a matrix which approximates the possible mappings of the child into the query. This technique seems to be more useful for propagating binding information gathered in search within a conceptual graph database.
Chapter 7

Typing Complex Objects

A hierarchy of graphs can be decomposed into factors of structure hierarchies and label hierarchies, which can further be factorised. These factors can be encoded in term lattices, and each graph can be encoded with the composition of the codes of its factors.

7.1 Introduction

Querying a conceptual graph knowledge base means comparing a query graph with the conceptual graphs in the knowledge base to see if it is present among, or implied by, the graphs already there. It is necessary to efficiently test whether one descriptive conceptual graph subsumes another. The set of maximal common subgraphs of two graphs may also need to be calculated, e.g., for performing the maximal join of conceptual graphs, or for finding unifying terms in unification-based theorem-proving. Each type-labeled conceptual graph has an unlabeled graph as its skeleton graph. The skeleton graphs form a poset ordered by subgraph inclusion.

This chapter examines how we can replace time-consuming graph comparisons with fast term or code computations. We precalculate a bit-vector (string of bits) for each graph, and subsumption testing is accomplished by parallel bit-wise logic operations on the bit-vectors. This amounts to embedding the set of conceptual graphs ordered by subsumption (graph inclusion) into a boolean lattice of bit-vectors. The knowledge base is thus compiled. Generally these bit-vector comparisons are necessary conditions which act as filters such that a general graph matching operation
is only applied if the filter succeeds. For some graphs the bit-vector operations are sufficient to determine a match. Some subgraph morphism algorithms use similar filters, but the main advantage of our scheme is that the filters are precomputed into an efficient code. In the following discussion we use Caseau’s compact hierarchical bit-vector encoding method for posets [Cas93] on the skeleton graph poset. See chapter 5 for a survey of encoding methods.

When we generalise this method to type-labeled (order-sorted) graphs like conceptual graphs (without negation), the poset of graphs ordered by subsumption is radically altered by the external type hierarchy on type-labels. This algebraic alteration is determined by two “product”-like operators which combine type-posets and graphs, skeleton product for labeled graphs sharing the same skeleton, and fret product for labeled graphs with different skeletons. These skeletons are similar to those used in the hierarchy classification technique developed in [Ell89a]. These operators must take into account the fact that any symmetries present in the compared graphs effect the form of the resulting hierarchy. The fret product combines the skeleton product with the inclusion hierarchy of all possible skeleton graphs; since this latter structure is universally applicable, it need be encoded only once. A current goal is to determine ways of combining encodings of type and graph hierarchies that will preserve comparison operations on the resulting combined codes for the fret product.

In essence we are continuing with the dream of the philosopher G. W. Leibniz (1646-1716) of an Ars Combinatoria, a mathematical encoding of the hierarchy of concepts to enable mechanical reasoning by calculation. Leibniz used prime numbers to represent conceptual primitives and multiplied them together to make composite concepts (analysed in chapter 5, section 5.2.1). He used the divisibility lattice of numbers as his knowledge hierarchy, so that if one concept’s number is divisible by that of a second concept, then the first concept is subsumed by the second (i.e. it is more specific). We believe that this was the wrong formal structure, since it did not properly account for relations, and we offer a encoding scheme as an alternative to Leibniz’s prime numbers. Also, we would like most reasoning by computer to be faster than arithmetic division.
Figure 7.1: The names of the graphs in the poset of unlabeled connected graphs with 5 nodes or less, §5 [Leh93]

7.2 Typing a Graph with its Skeleton Type

We propose to label a graph with its graph type. This graph type can then be used as a filter for matching graphs. We denote the type of the graph $u$, as $\tau(u)$. We can take the idea of a graph class to an extreme by giving each graph its own particular class. This may seem impractical, but by classifying a graph with a number of properties it is possible to achieve this for some useful sets of graphs. Here, we examine the possibility of using the unlabeled graph structure (skeleton) embedded in a graph as the type of the graph. We denote the graph $u$ with its labels removed as $\text{skeleton}(u)$. We denote the hierarchy over all possible unlabeled graphs or skeletons as the skeleton hierarchy, §.

As it happens, this is precisely the poset of the abstraction hierarchy on con-
ceptual graphs without negation (conceptual graphs without negated contexts), in which all individuals are of type “top”, all relations are symmetric, and distinctly named individuals are presumed to be distinct. Every such graph is subsumed by each of its subgraphs.

A serious defect of prior poset and lattice encoding schemes, for the type hierarchies and taxonomies in knowledge bases at least, is the fact that each change in the the type hierarchy requires that the whole hierarchy be recompiled and that new codes be assigned to many or all existing stored objects. Only a few applications have unchanging domain hierarchies; most are frequently updated. Our solution is to map every object to its code in the virtual hierarchy of of all possible descriptions. This is feasible for some structural factors of the hierarchy, particularly those due to the pure graph structure of the conceptual graph of a description (for graphs up to a certain size). Using the virtual poset of all graphs largely obviates the need to recompile the code-hierarchy with each addition, as others have had to do.

Figure 7.1 shows the skeleton hierarchy, §5, over all connected undirected graphs of 5 nodes or less. A dotted line between graphs $u$ and $v$ indicates that $v$ was formed from $u$ by adding a new node and attaching it to some node in $u$ with an arc, which corresponds to an external join in the conceptual graph canonical formation rules in [MC93], and the primitive operation $j_2$ or relative product in Burch’s Peircean Algebraic Logic [Bur92]. A solid line indicates that the more specialised graph was formed by an internal join [MC93] of two nodes in $u$; this is the same as the primitive operation $j_1$ of Burch, the equating of two otherwise already-related concepts.

Is it feasible to construct § or even a subset of it? In Table 7.1, column 3 shows how large § can get for even small graph sizes. Fortunately, we need not actually construct §. We only have to have an algorithm to calculate all the codes, and the structure will be “virtually reflected” in the operations on the codes. Since the structure § is a universal structure, its coding only needs to be done once. There are 11 716 571 connected graphs of 10 nodes. This may not seem especially useful for larger graphs. Even if this is the case, the method is useful for nested graphs which could be arbitrarily large. Conceptual graphs can be nested, since context concepts such as propositions, situations, and events can have graphs as their referent labels.
Large graphs are not easily computed with, constructed, edited, nor displayed. The problem of representing and processing complex highly related information is likely to be solved by structuring the information into components and modules. We believe nested graphs are a useful structuring tool for this problem. The useful thing about nested graphs is that only the graph structure at one level needs to be examined at a time. We intend for now to use this method for graphs of 10 nodes or less. That is for graphs which model the relationships between 10 objects or less. This is a useful subset, but in general a good cut-off point is the maximum graph size which can be bit-encoded within a word boundary of the relevant computer. It may also be possible to factor out the components in a graph into such smaller graphs which could then be identified using this method.

7.2.1 Indexing into the Poset of Skeletons

The skeleton of $u$ is looked up in the skeleton hierarchy using the name of the skeleton. The entry in the skeleton hierarchy has a type code associated with it. The type code is assigned to $u$. Using the hierarchical traversal methods discussed in chapter 2 would be inefficient. Unlabeled graphs are usually much harder to compute with than labeled graphs. Myaeng [MLL91] used this to support the argument that conceptual graphs are usually much more efficient to compute with than the worst case analysis of subgraph morphism would indicate. We want to avoid doing any actual skeleton comparisons. We denote some canonical property of an unlabeled graph $v$ as the name of $v$, written $name(v)$. We wish to find a canonical form, that is, a sequence which can be used as a “name.” This may not be possible for graphs in general, but is possible for usefully large sets of graphs. If we have some easily computable $name(v)$, then we can build a hash table on the names of all the skeletons in $. We propose to construct (or rather to encode) the skeleton hierarchy $\$ for some reasonable size of graphs. We then construct a hash table on the graph name.

The graph name is used as an index into $. We have done this by hand for some small collection of graphs, but for larger collections we expect to use efficient structure-generating algorithms found in [Gol93], which also list the group-theoretic
Figure 7.2: An encoding of the poset of unlabeled connected graphs §

orbits of each graph which are needed to determine the links in the poset §.

We have to find an easily calculable linear description of graphs suitable to use as a key for indexing into §. If this were possible in general, then the graph isomorphism problem would be tractable (whether graph isomorphism is formally tractable is presently unknown). It may however be possible to find a graph property or a small set of properties which uniquely describes any member of a finite set of graphs in §. Figure 7.1 shows names for skeletons in §₅. A graph is named by its “degree spectrum” or in other words its ordered multi-set of node arities. Lendaris also uses a filter on the arities of nodes [Len92]. However, we are precomputation this information in the hierarchy. For example, the name 1, 1, 1, 3 represents the graph ‡ of 4 nodes in Figure 7.1, three nodes each are connected to the node of arity 3. This graph property is certainly easy to calculate, but it does not uniquely describe the graphs [CK80]. There are two graphs n and p in Figure 7.1 with the label 1, 2, 2, 2, 3, and there are two graphs v and w labeled 2, 2, 2, 3, 3.

The graphs v and w can be distinguished by describing a node by its arity and
the arities of adjacent nodes:

\[(2, \{3, 3\}), (2, \{2, 3\}), (2, \{2, 3\}), (3, \{2, 3\}), (3, \{2, 2, 3\})\]

and

\[(2, \{3, 3\}), (2, \{3, 3\}), (2, \{3, 3\}), (3, \{2, 2, 2\}), (3, \{2, 2\})\]

These descriptions are similar to the node descriptions used in the multi-hierarchical retrieval methods in [Lev92]. Each node is described by its *neighbourhood*. A neighbourhood is a description of a node \(u\) and nodes of some fixed radius from \(u\). For a finite set of graphs it should be possible to pick a small radius to sufficiently describe each graph uniquely.

The node description method works well to fully distinguish most graphs encountered in practice, but it is possible to concoct graphs which cannot be distinguished by this method, such as large, highly regular graphs. Although such graphs are unlikely to represent natural language sentences (the domain conceptual graphs were designed for), the formalism of conceptual graphs is general enough to include them. To help further distinguish most of these, we can refine the classification of graphs with further graph attributes which are efficiently computable from the whole graph rather than being neighbourhood-based. Zickwolff [Zic91] has used the "attribute exploration" program developed at TH Darmstadt to develop a *concept lattice* of graphs and their attributes, based on the Formal Concept Analysis theory of Rudolf Wille [Wil82, Wil92]. A lattice like the one in Figure 7.4 is generated by the program.
Figure 7.4: Zickwollf’s [Zic91] “concept lattice” of graph classes. Each graph represents the class of graphs sharing the attributes appearing above it in the lattice. Based on a series of questions and proffered counterexamples [Wil91]; each graph inherits the attributes occurring above it in the lattice. The program can be used to enlarge the lattice of graph classes automatically as new graph attributes are added. A code for a graph’s most specific class in the lattice can be used as an adjunct in forming its name along with the node description method. Because the attributes are determinable in polynomial-time, so is this extended name of the graph.

7.2.2 Coding the Poset of Skeletons

The skeleton hierarchy $\mathcal{S}$ can be mapped into a hierarchy of codes. In the case of the skeleton hierarchy bit-vectors would be most compact. The skeleton hierarchy in Figure 7.1 is encoded using Caseau’s compact hierarchical method [Cas93] (see chapter 5, section 5.2.7) in Figure 7.2. Each graph in Figure 7.1 is associated with a type value $\tau$ in Figure 7.2.

We propose to implement $\mathcal{S}$ using boolean bit encoding. A boolean encoding assigns a code to each graph such that direct operations on the codes (like comparison) correspond to structural operations (like testing subsumption) in the poset of graphs ordered by generality. So, encoding and decoding must preserve the poset structure. The codes are variable length and can be assumed to be padded to the
Table 7.1: Characteristics of some sets of connected graphs including the number of connected graphs up to 10 nodes [Slo73], sequence 649, p.75.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>31</td>
<td>20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>143</td>
<td>15</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>996</td>
<td>21</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>12 113</td>
<td>28</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>273 193</td>
<td>36</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>45</td>
<td>11 989 764</td>
<td>45</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>190</td>
<td></td>
<td>190</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1 225</td>
<td></td>
<td>1 225</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>4 950</td>
<td></td>
<td>4 950</td>
<td>155</td>
<td></td>
</tr>
</tbody>
</table>

right with zeroes.

Consider the comparison of the query graph u with the database fact graph v in Figure 7.3. The graphs u and v represent block arrangements in some block world. The query graph u is skeletonised and the name of u is computed as an ordered multiset of degrees of nodes of $u = 2, 2, 2, 2, 4$. The graph u is looked up in § by hashing on name(u). The type of u, $\tau(u) = 11111101110111$ is found from the encoding of §. Similarly, the graph type of v is precomputed and stored in the database by using the name of $v = 2, 2, 2, 3, 3$ to get the encoding of $v = 111111111010111101$. The graph u is not a supergraph of the graph v since $\tau(u)$ is incomparable to $\tau(v)$. The size of $\tau(u)$ is smaller than $\tau(v)$, so the only possible relationship between the codes is $\tau(u) \sqsubseteq \tau(v)$, but the 10th slot of $\tau(u)$ is a 1, whereas it is 0 in $\tau(v)$. Thus a potentially expensive subgraph morphism test is replaced with an operations on a compact code. In this example, the type codes fit into one machine word ($\tau(v)$ uses 18 bits). A graph comparison is replaced with one machine word operation.

### 7.2.3 Complexity of Codes

Figure 7.5 illustrates that a lower bound for the length of order-preserving codes encoding the poset of connected graphs of $n$ nodes or less, $\$n$, is $O(n^2)$. Since $\binom{n}{2} = n(n - 1)/2$ is the number of edges in the complete graph of $n$ nodes, then a
Table 7.2: Representing a graph by projecting into a complete graph

<table>
<thead>
<tr>
<th>Complete</th>
<th>Graph</th>
<th>Edges in Complete</th>
<th>Boolean Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>e</td>
<td>{2, 4, 6}</td>
<td>010101</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>{1, 2, 6}</td>
<td>110001</td>
</tr>
</tbody>
</table>

lower bound of the code length is $O(n^2)$. That is, at least $\binom{n}{2}$ bits are needed to encode $S_n$.

If it is possible to encode $S_n$ into a boolean space of less than $\binom{n}{2}$ bits, this means $S_n$ can mapped into $B_{\binom{n}{2}}$ while preserving order. One labelling of the $B_{\binom{n}{2}}$ hierarchy is the poset of all possible subsets of edges of the complete graph of size $n$. Table 7.2 shows how graphs relate to the power set of edges in the complete graph of size 4. The edges of the complete graph of size $n$ are numbered from 1 to $\binom{n}{2}$. Then any graph of size $n$ (or less) for a given orientation can be represented as a set of edges (numbers). This set of numbers can then be represented in a boolean bit string. The poset on the left in Figure 7.5 (where edges have been omitted) represents the poset of all possible graphs by taking subsets of the edges in the complete graph of size 4. This poset maps into $B_6$. We should be able to map the poset of connected unlabelled graphs into the poset of general (possibly unconnected, possibly isomorphic) graphs. The poset on the right in Figure 7.5 is $S_4$. It is not possible to embed $S_4$ into the left poset. Compare columns “Caseau Code Size” and “A Lower Bound Code Size” in Table 7.1. Column “Caseau Code Size” represents results of encoding $S_n$ by hand. Column “A Lower Bound Code Size” represents a lower bound on the number of bits needed. Notice that the “Max $|E|$” (edges) column is the same as the “A Lower Bound Code Size” column which equals $\binom{n}{2}$. We expect given a graph of $n$ nodes and $m$ edges, that the code will be at least as large as $\max(m, n)$.

Work is ongoing into determining the complexity of the type codes with respect to the graphs they encode.
7.3 Factorisation of Graphs

The above work was inspired by the work of Lehmann [Leh93] and was done in collaboration with Lehmann [EL94a]. Lehmann has been working on the the harder problem of encoding type-labelled graphs. The following is a review of that work which also appeared in [EL94a]. The situation becomes more complicated when we generalise to the type-labeled graphs used in conceptual graphs, based on Sowa’s “type lattice”\(^1\). Graph elements may also be labeled with type labels taken from one or more ordered sets (trees, lattices or general posets) of labels. The shape of the type lattice profoundly effects the algebraic (poset) structure of the resulting abstraction hierarchy of type-labeled graphs.

Lehmann’s goal [Leh93] is to factorise a knowledge base into its graph and type hierarchy factors, and further factorise these factors. Using the encoded factors of a graph \( u \), \( u \) can be assigned a code based on the composition of codes derived within each factor. In this way it is possible to replace subgraph morphism tests by efficient code comparisons on composite bit strings. For general relational graphs (with possible symmetries) like conceptual graphs, as opposed to purely functional “feature structures”, the analysis depends on two product-like operators, skeleton product and fret product.

A skeleton product \( \otimes \) is the hierarchy induced by taking the unlabeled graph

\(^1\)Parker-Rhodes also used them in his Inferential Semantics scheme using lattice-ordered “base domains” for labels, Ait-Kaci used them for his abstract type lattices, and various other have used them in the contexts of semantic networks, order-sorted logics and hybrid KL-ONE reasoners [Leh92].
(skeleton) and a set of label hierarchies, and producing a partial ordered set of graphs based on all possible assignments of labels to the skeleton graph. Figure 7.6 shows the hierarchy of all possible triangle graphs with labels from the hierarchy \( \{a, b\}, \{a \sqsupseteq c, b \sqsupseteq c\} \). This hierarchy is in the skeleton multiplication table given in Table 7.3 in the fourth row, third column. Table 7.3 gives the skeleton “multiplication” table for the first four smallest connected graphs as rows, and six of the smallest possible type hierarchies. Thus by grouping together all graphs with the same skeleton, it is possible to reduce a group of graphs to skeleton and type hierarchy factors.

A *fret product* \( \otimes \) is the hierarchy induced by taking a hierarchy of skeletons ordered by graph inclusion, such as \( \sqsubseteq \), and a set of one or more label hierarchies, and producing the partially ordered set of graphs based on all possible assignments of labels to every skeleton graph. Figure 7.7 shows the simplest non-trivial hierarchy that is computed by a fret product, namely \( 2 \otimes 4 \) on the skeleton hierarchy \( 4 \) with the type hierarchy \( \{1, 0\}, \{1 \sqsupseteq 0\} \). This hierarchy corresponds to the second column from the skeleton multiplication \( \otimes \) table.

Although it’s complicated, the structure \( 4 \otimes 2 \) in Figure 7.7 is clearly “fret-factorable” into the simple type lattice \( 2 \) and the universal poset \( 4 \). Each block of Figure 7.7 (enclosed in a dotted capsule) corresponds to a node in \( 4 \) and is in fact
The Simplest Nontrivial Fret Product: $S_n \times 2$

Subgraph inclusion (injective) of undirected graphs, all graph nodes labeled from the two-element poset, for $n \geq 4$.

Poset $2^+ = \{1\}$

The universal structure fret factor $S_4$:

Fritz Lehmann, GRANDAI Software
4282 Sandburg, Irvine, California 92715, USA
(714) 733-0566 fritz@rodin.wustl.edu

Figure 7.7: A Fret $\times$ product
Table 7.3: Lehmann’s Skeleton \$ multiplication table [Leh93]. Graphs are at the left and type posets are along the top. The entries are posets of type-labeled graphs. The large circles occur in nested posets; these are direct products of the inner and outer posets. (The nested notation has been extended to subdirect representations [Wil89].)

<table>
<thead>
<tr>
<th>Poset $ Graphs</th>
<th>{</th>
<th>&gt;</th>
<th>|</th>
<th>\diamond</th>
</tr>
</thead>
<tbody>
<tr>
<td>\cdot \cdot \cdot</td>
<td>{</td>
<td>&gt;</td>
<td>|</td>
<td>\diamond</td>
</tr>
<tr>
<td>\cdot \cdot \cdot</td>
<td>{</td>
<td>&gt;</td>
<td>|</td>
<td>\diamond</td>
</tr>
<tr>
<td>\cdot \cdot \cdot</td>
<td>{</td>
<td>&gt;</td>
<td>|</td>
<td>\diamond</td>
</tr>
<tr>
<td>\cdot \cdot \cdot</td>
<td>{</td>
<td>&gt;</td>
<td>|</td>
<td>\diamond</td>
</tr>
</tbody>
</table>

a skeleton product \( G \# \# 2 \) where \( G \) is the graph of the corresponding \$4 node (What Lehmann calls the fret product was in essence the “context repertory” described by Parker-Rhodes [PR78], Chapter V.).

The skeleton hierarchy \$ assumes that for one graph to subsume another more specific one, it must be a subgraph, that is, there must be an injective mapping of nodes which preserves links. Conceptual graphs as formally defined, and other intensional graph subsumption systems, do not assume that the mapping is injective. Two distinctly named individuals in one graph can project into one individual in the other graph so long as the links are preserved. Thus a long chain of one graph may project into a small cycle of the other graph. Lehmann is currently working on developing the appropriate poset for non-injective mappings which is the counterpart of \$ for injective mappings. To accommodate relations of arbitrary arities, we need to encode the quasi-order (and the poset of homomorphic equivalence classes) of arc-labeled connected directed hypergraphs under non-injective mapping.

Skeletons and skeleton products are combined in the fret product. Work is
ongoing into how codes for these factors can be combined into a general type code
preserving the desired poset operations on the full fret product.

7.4 Summary

We have shown that it is possible to type graphs and encode them based on the
factors of a graph such as graph class, skeleton, and type-labelings. Qualitatively
defined graph classes can be used as a filter on expensive graph comparisons. The
graph class technique can be used for the potentially infinite set of typed conceptual
graphs of, say, around size 10 or less. We envision these graph typing methods
will be used in the hierarchy classification algorithms that have been developed for
conceptual graphs.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

Relational databases do not provide scalable storage and retrieval for complex objects such as chemicals, program specifications, images, and CAD designs because their indexing methods are based on totally ordered simple data. These complex object domains can be handled by indexing on attributes such as substructure, subimage, or subconcept which capture ordering information about structure, behaviour or semantics.

This thesis, we believe, is to date the most extensive study of the problem of complex object management. We have surveyed the algorithms for classification of objects into a hierarchy and made several improvements to these algorithms including management of topological search process; use of unification operations during descendant search; use of code hierarchies in an encoded search for ancestors; and a modulated path search for membership testing. We have shown the theoretical performance of these and other known classification algorithms on four benchmark hierarchies: chains, decision trees, term lattices, and boolean lattices. We illustrated that the topological methods developed by Levinson [Lev84] and refined here outperform hybrid methods based on traditional binary search algorithms. Using the Peirce conceptual graph database system developed by the author we showed that topological search and improved techniques outperformed previous algorithms on databases of chess graphs. We also showed that adding a canon of simple objects to
the database can significantly reduce the cost of object classification.

To support the encoded search algorithms we developed encoding techniques which assign simple code objects to hierarchies based on the shape of the hierarchy. As well as being useful for search in complex object databases, these encoding techniques are also useful for implementing type operations in type hierarchies and class libraries. We extensively surveyed encoding techniques tracing back to 1679 with the development of Leibniz’s Universal Characteristic. Leibniz’s goal was to implement logical deductions with relatively simple calculations such as multiplication and division. We showed that term encoding (which we are currently developing an algorithm for) is more efficient than previous encoding methods on multiple inheritance hierarchies such as those shaped like term lattices and performs as well as other methods on boolean lattice shaped hierarchies. Term encoding is more compact for methods with very large branching factors. We also showed that adding a canon of simple objects to the database can significantly reduce the size of the codes produced.

We showed that compilation and compression was possible within hierarchies by representing objects as the difference between itself and all its parent subpatterns stored in the hierarchy. This resulted in databases which were more compact than databases using previous methods based on storing an object as the difference between itself and one parent. This illustrates that multiple inheritance is a useful paradigm for knowledge compression and compilation.

Finally, we illustrated by hand for graphs of 10 nodes or less, how these graphs can be compared using a single bit-vector AND computer instruction. Types intrinsically have structure, and conversely structures exhibit typedness. In conceptual graphs, the unlabeled graph structure can act as a type: for an irredundant conceptual graph to be a generalisation of another, there must first exist a subgraph relationship of the graphs ignoring the label information. A graph type can be used as a filter on labeled graph matching. Further this process should extend to other combinatorial structures and formulae.
8.2 Recommendations

After analysing orders and the factors which affect object classification and encoding operations we encourage:

**Multiple inheritance:** When adding an object to a database, ensure that any new information which is introduced by the object is covered by some existing subobject within the database. If necessary introduce new subobjects into the database. The topological search uses the rule that an object should not be compared unless all its subobjects (stored in the database) match a query. The goal of a self-organising complex object database then is to ensure that enough subobjects are stored to cover the object, that is, they can be simply joined to produced the object. This is encouraged because the classification algorithms ensure that an object’s subobjects match the query object before attempting to match it. An object’s subobjects differentiate it from other objects in the database.

**Use of canonical databases:** A canonical database is one in which all objects are derived from a small number of primitives and maybe a large number of simple individual markers. Canonical formation rules operate by joining and restricting existing objects in the database and hence encourage multiple inheritance.

**Use conventional indexing techniques where possible:** Conventional indexing techniques should be used on the simple objects in the primitive atomic levels of the hierarchy. Most classification techniques do a linear search of the primitives, since there is no ordering between the primitives. In practice, there will be large numbers of primitives. Existing hashing and indexing techniques should be used for this part of the search.

8.3 Future Work

In the near future we will continue developing more sophisticated scalable algorithms. The current algorithms are scalable, but need further engineering to make
them practical for very large scale object databases. One avenue we will work on in the short term is the multi-level hierarchical retrieval method developed with Levinson [LE92]. The idea behind this method is that a hierarchy of conceptual graphs can be seen as the product of at least four levels of hierarchies of objects: concept and relation, node description units, node descriptions and finally conceptual graphs. A node description consists of a set of adjacent node description units to a certain radius. A conceptual graph consists of a set of node descriptions.

There are many interesting avenues to follow with encoding including compact encoding of general relationships (which would have an impact on relational database technologies); encoding types with lists of types or more general knowledge representation languages such as conceptual graphs; learning of object descriptions from the codes; and implementation of large chunks of logic using codes.

It is interesting to consider why the goal of object encoding is to produce bit-vectors, and our work focuses on encoding with lists of integers. Why not have lists of types or conceptual graphs? One of the problems with Figure 1.3 in chapter 5 is that the bottom concept Sierpinski Triangle and Sparkling Diamond could be encoded with one concept name. That is, by increasing the vocabulary and creating types as abstractions, it is possible to reduce the size of the codes. So instead of a code consisting of a list of integers, it might consist of a list of types. Thus by finding patterns in the codes, these patterns can be encoded into types to give further compression. This can be taken further, by finding (graph) structure in the codes, it
is possible to encode relationships using conceptual graph codes. Indeed this follows the approach taken in knowledge representation to find a concise representation of a situation. Conceptual graphs are codes. The question is how much of the information in a conceptual graph is necessary for computation. Is there a useful time/space tradeoff in converting to simpler (to compute with) codes? A principle of knowledge processing is “to process objects at the highest level of abstraction that gives no loss of accuracy” [Lev94]. Levinson gave the example of chemists reasoning with xOH and xH where x could be large compounds. It is possible that much of the code information is unnecessary for code computation.

Other areas that should be explored include the relationship between the codes and objects they encode in the original domain. For example, consider the completed images hierarchy in Figure 8.1, where the Sierpinski Triangle and Diamond image is added by computing the greatest lower bound of Sierpinski Triangle and Diamond. These objects can be induced from the codes for the position <1,1>.

Our goal is to replace, as much as possible, the algorithmic theorem proving involved in subsumption with code operations. Table 8.1 shows that codes can

<table>
<thead>
<tr>
<th>Classical Logic</th>
<th>Euler Diagram</th>
<th>Conceptual (Existential) Graphs</th>
<th>Hierarchy</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>All P are Q</td>
<td>![Diagram]</td>
<td>![Conceptual Graph]</td>
<td>![Hierarchy]</td>
<td>![Terms]</td>
</tr>
<tr>
<td>Some P are Q</td>
<td>![Diagram]</td>
<td>![Conceptual Graph]</td>
<td>![Hierarchy]</td>
<td>![Terms]</td>
</tr>
<tr>
<td>No P are Q</td>
<td>![Diagram]</td>
<td>![Conceptual Graph]</td>
<td>![Hierarchy]</td>
<td>![Terms]</td>
</tr>
<tr>
<td>Not all P are Q</td>
<td>![Diagram]</td>
<td>![Conceptual Graph]</td>
<td>![Hierarchy]</td>
<td>![Terms]</td>
</tr>
<tr>
<td>if P and Q, then R</td>
<td>![Diagram]</td>
<td>![Conceptual Graph]</td>
<td>![Hierarchy]</td>
<td>![Terms]</td>
</tr>
</tbody>
</table>
Figure 8.2: A relationship participation lattice

Figure 8.3: A hierarchy of quantified schemas of three variables
support monadic logic. Statements such as “All A is B” can be encoded with two genes. A can be encoded with <1> and B with <1,1>, such that code(A) subsumes code(B). This means, that many kinds of proofs can be replaced with simple parallel code operations in the tradition of Leibniz. This can be extended to other kinds of quantified statements. Figure 8.2 shows a lattice of the participations of arguments in binary relationships used in entity-relationship diagrams. This lattice can be encoded using 4 bits. Figure 8.3(a) shows a hierarchy of predicate calculus statements consisting of combinations of universal and existential quantifiers over three variables. Figure 8.3(b) shows an equivalent conceptual graph form and Figure 8.3(c) shows $B_3$ which encodes these hierarchies. The query $\forall x : X \exists y : Y \exists z : Z \circ P(x, y, z)$ is implied by $\forall x : X \exists y : Y \forall z : Z \circ P(x, y, z)$ can be tested by $100 \land 101 = 101$.

These examples illustrate that it is possible to encode formulas and constraints as simple codes and that complex algorithmic matching can be replaced with efficient operations on compact codes.

This area continues to produce many interesting directions for future research.
Bibliography


[Han91] Philipp Hanschke. TAXLOG: A logic programming language with deeply-modelled type hierarchies. draft note from DFKI, 1991.


Index

Ancestors, 44
AncestorsJoin, 129
Atoms, 45
Children, 44
Clash, 159
ClashCrown, 159
ClashGroups, 160
CoAtomIdealAndShoots, 147
CoAtomIdealAndShootsMeetSizes, 147
CoAtomIdealAndShootsMeets, 147
CoAtomIdealMeets, 162
CoAtomIdealMeets(S), 162
CoAtomIdeals, 162
CoAtoms, 45
Comparables, 44
Descendants, 44
DescendantsMeet, 129
Fringe, 45
GreatestLowerBound (GLB), 129
Ideal, 147
IdealAndShoots, 147
Incomparables, 44
JoinIrreducibles, 45
LeastUpperBound (LUB), 129
LowerBoundsCrown, 129
MaxCoAtomIdealMeets, 162
MaxCoAtomIdealMeets(S), 162
MeetIrreducibles, 44
Parents, 44
Shoots, 45
SimpleAllClashCrown, 159
SimpleClashCrown, 159
Twigs, 45
UpperBoundsBase, 129
Ancestors=, 44
Descendants=, 44
delete, 87, 90
depth, 44
diff, 160
glb_partition, 159
group, 160
insert, 87, 89
intersection, 88, 97
member, 87, 88
numbering, 160
same(X, Y), 162
subset, 88, 92
union, 88, 94
width, 44

absurd space, 157
absurd space decision graph, 158
All Parents, 208
ancestor, 43
ancestors, 23
decision tree, 75
anti-chain, 45
descendant, 43
Ars Magna, 135
descendants, 23
Base, 44
Design Method I, 47
bit-vector, 133
DoD, 24
bit-vector AND, 133
domain of discourse, 24
bit-vector intersection method, 66
duplicates, 193
bit-vector OR, 133
Dynamic Memory, 25
Boolean Lattice, 78
existential graphs, 182
bottom, 129
Focus, 103
brute force, 46
focus, 60
canonical formation rules, 182, 191
fret product, 212, 221, 222
cardinality, 44
gene, 135
Chain, 75
generalisation hierarchy, 188
equal cover, 18
chain covering, 68
greatest lower bound, 113, 136
class, 24
group code, 143
classification, 89
Hasse diagram, 43
classified, 35
hierarchy, 43, 50
coatom codes, 158
index objects, 49
code, 135
Inferential Semantics, 221
compact hierarchical encoding, 148
insert, 89
comparable, 43
irreducible encoding, 151
complex object, 24
irredundant graphs, 195
content addressable memory, 177
Largest Parent, 208
crown, 44
lattice, 129
crown, 44
least upper bound, 136
data objects, 49
level ancestor encoding, 150
database unification, 94
local bottom, 119
decision, 75
local code, 143
Decision Tree, 75
local top, 119
term encoding, 131, 137, 163
minimal chain covering, 68
Term Lattice, 76
modulated compact hierarchical, 150
terms, 126
module, 119, 143
time stamp, 153
most general unifier, 113
top, 129
name, 215
topological, 55
neighbourhood, 217
tree encoding, 151
non-coatom absurd crown, 158
Universal Characteristic, 135, 137
parents, 23
virtual address, 42
partially ordered set, 43
Peirce, 183
Power Set, 43
properly subsumed by, 43
properly subsumes, 43
reachability, 136
recursive subpattern, 127
screens, 49
skelton graph, 211
skelton hierarchy, 213
skelton product, 212, 221
subset, 92
subsumed by, 43
subsumes, 43
subsumption, 136
Subsumption Encoding, 130
subsumption encoding, 130, 136, 154
Term Encoding, 131