Loughborough University

This item is held in Loughborough University's Institutional Repository (https://dspace.lboro.ac.uk/) and was harvested from the British Library's EThOS service (http://www.ethos.bl.uk/). It is made available under the following Creative Commons Licence conditions.

creative commons

COMMONS DEED

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

GOAL DRIVEN THEOREM PROVING USING CONCEPTUAL GRAPHS AND PEIRCE LOGIC

by

John Edward Heaton

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of
Doctor of Philosophy of the Loughborough University of Technology

October 1994

© by John Edward Heaton 1994

ACKNOWLEDGEMENTS

I wish to acknowledge the encouragement and advice given to me by my Director of Research, Professor Ernest Edmonds, and by my Supervisor, Pavel Kocura. With their help I have been able to refine this thesis into what it now is, both with respect to its scientific content and to its literary style.

I also wish to acknowledge the help and encouragement given to me by my wife, Elizabeth, who sat through many hours of agony whilst I expounded my theories to her. Her ability to listen and criticise helped me to refine the theory to its present state.

ABSTRACT

The thesis describes a rational reconstruction of Sowa's theory of Conceptual Graphs. The reconstruction produces a theory with a firmer logical foundation than was previously the case and which is suitable for computation whilst retaining the expressiveness of the original theory. Also, several areas of incompleteness are addressed. These mainly concern the scope of operations on conceptual graphs of different types but include extensions for logics of higher orders than first order. An important innovation is the placing of negation onto a sound representational basis.

A comparison of theorem proving techniques is made from which the principles of theorem proving in Peirce logic are identified. As a result, a set of derived inference rules, suitable for a goal driven approach to theorem proving, is developed from Peirce's beta rules. These derived rules, the first of their kind for Peirce logic and conceptual graphs, allow the development of a novel theorem proving approach which has some similarities to a combined semantic tableau and resolution methodology. With this methodology it is shown that a logically complete yet tractable system is possible. An important result is the identification of domain independent heuristics which follow directly from the methodology. In addition to the theorem prover, an efficient system for the detection of selectional constraint violations is developed.

The proof techniques are used to build a working knowledge base system in Prolog which can accept arbitrary statements represented by conceptual graphs and test their semantic and logical consistency against a dynamic knowledge base. The same proof techniques are used to find solutions to arbitrary queries. Since the system is logically complete it can maintain the integrity of its knowledge base and answer queries in a fully automated manner. Thus the system is completely declarative and does not require any programming whatever by a user with the result that all interaction with a user is conversational. Finally, the system is compared with other theorem proving systems which are based upon Conceptual Graphs and conclusions about the effectiveness of the methodology are drawn.

KEYWORDS

Conceptual Graphs, Expert Systems, Knowledge Processing, Knowledge Representation, Peirce Logic, Predicate Logic, Theorem Proving.

CONTENTS

Acknowledgements

Abstract

Keywords

Contents

Chapter 1 - Knowledge Representation Formalisms

1.1 Introduction

1.2 Knowledge Representation Formalisms

1.2.1 Early Knowledge Representations

1.2.1.1 Representation And Semantic Ambiguity

1.2.1.2 Computation

1.2.2 Later Knowledge Representations

1.2.3 The Future

1.3 Conceptual Graphs

1.4 Implementations

1.4.1 General Observations

1.4.2 A Brief Review Of Some Existing Systems

1.4.2.1 Systems Designed To Use Operations On Conceptual Graphs

1.4.2.2 Other Approaches

1.4.3 Further Considerations

1.5 Motivation And Aims Of The Thesis

1.6 Plan Of The Thesis

Chapter 2 - Conceptual Graphs And Knowledge Representation

2.1 Introduction

2.2 Essential Concepts

2.3 Conceptual Graphs

2.4 The Concept Node

2.4.1 The Referent

2.4.1.1 Individual Marker - #n

2.4.1.2 Definite Article - #

2.4.1.3 Name - 'String'

2.4.1.4 Generic Referent - *

2.4.1.5 Coreference Marker - *x
2.4.1.6 Measure - @ 3 Miles.
2.4.1.7 Set (Various Types) - { ... }
2.4.1.8 Universal Quantifier - "
2.4.1.9 Graph - { Graph }
2.4.1.10 Summary Of Referent Semantics
2.4.2 The Type Label
2.4.3 Lines Of Identity And Higher Order Logic
2.4.4 Formal Definitions Relating To The Concept Node
2.5 The Relation Node
2.5.1 Relation Label
2.5.2 Direction Of Relations
2.5.3 Formal Definition Of Relation
2.5.4 Canonical Graphs
2.5.5 Reasoning With Selectional Constraints
2.5.6 The Nature Of Canonical Graphs
2.6 Type And Relational Hierarchies
2.7 Conformity Relations
2.8 Logical Primitives Of Conceptual Graphs
2.8.1 Conjunction
2.8.2 Negation
2.8.2.1 Negation And Logical Context
2.8.2.2 What Is A Proposition?
2.8.2.3 Logical Considerations
2.8.2.4 A Solution
2.8.2.5 Further Examples
2.8.3 Extending The Definition Of Graph
2.9 Conceptual Graphs And Standard Logic
2.9.1 Simple Graphs
2.9.1.1 Simple Concepts
2.9.1.2 Modal Concepts
2.9.1.3 Identity
2.9.2 Negated Contexts
2.9.3 The Formula Operator ϕ
2.10 Abstraction And Definition
2.10.1 Type And Relational Definitions
2.10.1.1 Type Definitions
2.10.1.2 Relational Definitions

2.10.2 Lambda Abstractions Within The Type And Relation Fields
2.10.3 Individuals
2.10.4 Schemata And Prototypes
2.10.5 Actors
2.11 Other Forms Of Reasoning
2.12 Summary
2.12.1 Summary Of Conceptual Graphs Structures
2.12.2 Model Theoretic Semantics
2.12.2.1 Canonical Basis
2.12.2.2 The True Set
2.12.2.3 The False Set
2.12.3 Type Lattice And Relational Lattice
2.12.4 Conformity Relations Table
2.13 Conclusions Of Chapter 2

Chapter 3 - Conceptual Graphs And Logic

3.1 Introduction
3.2 Canonical Formation Rules
3.2.1 Copy
3.2.2 Join
3.2.3 Restrict
3.2.4 Simplify
3.2.5 Detachment
3.3 Generalisation And Specialisation
3.3.1 Generalisation
3.3.2 Common Generalisation
3.3.3 Specialisation
3.3.4 Common Specialisation
3.3.5 Maximal Join
3.4 Abstraction And Definition
3.4.1 Lambda Calculus
3.4.2 Operations With Schemata And Prototypes
3.5 Aggregation And Individuation
3.6 Logical Operations - Peirce's Rules
3.6.1 Survey Of The Complete Set Of Rules As Given By Sowa
3.6.2 Analysis Of Peirce's Rules
3.6.3 Derived Rules Of Inference And Duplication Of Operations
3.6.3.1 Canonical Formation Rules

3.6.3.2 Generalisation
3.6.3.3 Common Generalisation
3.6.3.4 Specialisation
3.6.3.5 Common Specialisation
3.6.3.6 Maximal Join
3.6.4 Iteration And Linear Notation
3.7 Projection
3.7.1 Projection Of Simple Graphs
3.7.2 Projection Of Compound Graphs
3.7.2.1 Projection And The Rules Of Peirce Logic
3.7.2.2 Rules For Projection Of Compound Graphs
3.7.3. Projection Of Modal Concepts
3.8 Conclusions Of Chapter 3

Chapter 4 - Computation With Peirce Logic

4.1 Introduction
4.2 Fundamental Concepts
4.2.1 Objectives
4.2.2 Knowledge Base Structure
4.2.3 Model Theoretic Considerations
4.2.4 Proof Strategies
4.2.5 Types Of Proof
4.2.5.1 Proof Of Simple Graphs
4.2.5.2 Proof Of Compound Graphs
4.4.3 Factorisation Of Proof Strategies
4.3 Selectional Constraints
4.3.1 Logical Enforcement Of Selectional Constraints
4.3.2 Specialisation Enforcement Of Selectional Constraints
4.3.3 Additional Tests
4.4 Lookup
4.4.1 Lookup And The Open World
4.4.2 The Evaluation Game
4.4.3 Lookup And Knowledge Base Structure
4.4.4 Lookup Heuristics
4.5 Deduction
4.5.1 Typical Deductive Mechanisms
4.5.2 Deduction And Peirce Logic
4.5.3 A Deductive Mechanism

4.6 Theorem Proving
4.6.1 Data Driven Non-uniform Approaches
4.6.1.1 Natural Deduction Proof Methods
4.6.1.2 Sequent Proof Methods
4.6.1.3 Conclusion Of Analysis Of Data Driven Non-uniform Methods
4.6.2 Goal Driven Non-uniform Approaches
4.6.2.1 Semantic Tableau Methods
4.6.2.2 Conclusion Of Analysis Of Goal Driven Non-uniform Methods
4.6.3 Data Driven Uniform Methods
4.6.3.1 Resolution
4.6.3.2 Conclusion Of Analysis Of Data Driven Uniform Methods
4.6.4 Goal Driven Uniform Methods
4.6.4.2 Conclusion Of Analysis Of Goal Driven Uniform Methods
4.6.4.3 General Conclusions Relation To Uniform Methods
4.6.5 Summary Of Review Of Proof Methods
4.7 Peirce Logic And Proof Strategies
4.7.1 The Nature Of Proofs In Peirce Logic
4.7.2 Peirce Logic And The Goal Driven Method Of Proof
4.7.2.1 Reversibilty Of Inference Rules
4.7.2.2 Reversing The Effect Of An Erasure
4.7.2.3 Reversing The Effect Of An Insertion
4.7.3 Further Considerations Related To Iteration
4.7.4 Propositional Proofs And First Order Proofs
4.7.4.1 The Projection Operation
4.7.4.2 The Effect Of Restriction
4.7.5 The Rules Of Inference For The Goal Driven Method
4.7.5 Previous Work
4.8 Proof By Last Resort
4.9 Conclusions Of Chapter

Chapter 5 - From Theory To Implementation

5.1 Introduction
5.2 General Points About The System
5.2.1 Choice Of Platform
5.2.2 Logical Requirements Of The Knowledge Base
5.2.3 General Description Of The System
5.3 Execution Cycle
5.4 Input And Standardisation

5.5 Selectional Constraints And The Lattices
5.5.1 Placing New Type And Relation Labels Within The Lattices
5.5.2 Requesting Definitions
5.5.3 Lattice Structure And Efficiency
5.5.4 Requesting Canonical Graphs
5.5.5 Testing For Selectional Constraint Violations
5.6 Computing Denotations And Truth Values
5.6.1 Overall Strategy
5.6.2 High Level Structure Of The System
5.7 Lookup
5.8 The Deductive Mechanism
5.8.1 Top Level Of The Deductive System
5.8.2 Selection Of Rules
5.8.3 Handling The Results Of A Deduction
5.8.4 Nonterminating Loops
5.8.5 Repetition Of Work
5.8.6 Completeness Of The Deductive Mechanism
5.9 The Theorem Prover
5.9.1 Organisation Of The Theorem Prover
5.9.2 Restriction
5.9.3 Deiteration Against Graphs Within The Formula
5.9.4 Removal Of Double Negations
5.9.5 Empty Clause Elimination
5.9.6 Deiteration Against The Knowledge Base
5.9.7 Iteration
5.9.8 Algorithm Of The Theorem Prover
5.10 Last Resort
5.11 Updating The Knowledge Base
5.12 Evaluation Of CGP
5.12.1 Informal Evaluation
5.12.2 Formal Evaluation
5.12.2.1 Bokkers
5.12.2.2 Lukose
5.13 Conclusions Of Chapter 5

Chapter 6 - Conclusions And Future Work

6.1 Introduction
6.2 Summary Of The Rational Reconstruction

6.2.1 Representation
6.2.1.1 Referents
6.2.1.2 Isomorphism Of Field Types
6.2.1.3 The Cut
6.2.1.4 Definitions
6.2.1.5 Lattices
6.2.2 Computation
6.2.2.1 Canonical Formation Rules
6.2.2.2 Operations Of Expansion And Contraction
6.2.2.3 Operations Of Logic
6.3 Summary Of The Model And The Prototype
6.4 Future Work
6.4.1 Efficiency Of Computation
6.4.2 Completeness
6.4.3 Open Versus Closed Worlds
6.4.4 Explanation And Theory Building
6.4.4.1 Representation
6.4.4.2 Generation Of Nested Knowledge Bases
6.4.4.3 Reasoning With Modal Concepts
6.4.4.4 Combining Knowledge Bases
6.4.5 Fuzzy Reasoning
6.5 Final Conclusion

References

Appendix I - Theory Generation

Appendix II - CGP User Guide

CHAPTER 1

KNOWLEDGE REPRESENTATION FORMALISMS

1.1 INTRODUCTION

This chapter consists of a necessarily brief review, with respect to representational and computational efficiency and completeness, of a selection of knowledge representation formalisms. These will be chosen with a view to identifying underlying principles of such formalisms and showing possible deficiencies. Each will be illustrated in the forms in which they first appeared, since we may thus gain insights into the way such formalisms arise and the problems they attempt to address. Following this we introduce and review Sowa's Conceptual Graphs theory and show how far it goes in satisfying the perceived requirements of knowledge representation. We then discuss general principles concerning the use of very rich representations and finally, we review some existing computer programs based on conceptual graphs. As a result of these discussions we will be able to state the aims of the thesis.

Since it is not the purpose of this thesis to present a comprehensive review of knowledge representation formalisms and logics the contents of this chapter will be of a very general and somewhat informal nature.

1.2 KNOWLEDGE REPRESENTATION FORMALISMS

Expert systems, knowledge based systems and deductive databases have been around for several years. They represent the practical embodiment of the theoretical work started by Aristotle, with his *categories* and *syllogisms*, and the more recent developments such as predicate logic (in its various forms) and graphical/network formalisms. Many of these knowledge processing systems are advanced and sophisticated (relatively) and might be expected to find wide appeal amongst people who use and interpret information. Nevertheless their

use in the real world, particularly in industry, is not as widespread as perhaps it ought to be. This lack of use may be caused by the following factors, which we discerned during a period of industrial employment:

I Despite their relative sophistication, current expert systems do not adequately model their users' problems either because the theory on which they are based is intrinsically not sufficiently advanced or because they are too complex to use.

II Current computers (in particular the desktop computers on which expert systems are often used) are perhaps not powerful enough to handle realistically large or complex problems of the kinds that occur in industry.

III Some expert systems do not work correctly or are not sufficiently intuitive and these points discourage the use of better systems by the less experienced computer user. As a result, expert systems in general are not seen to be useful.

IV The full power of logic is compromised in many expert systems for reasons of computational complexity, leading to loss of usefulness of the system. This is an intrinsic problem of many logics but maybe the best compromises have not yet been found.

V Many systems must, in effect, be *programmed* by the user. This somewhat defeats the objective of an expert system which, by definition, should not require the user necessarily to be a domain expert and certainly should not require them to be a computer programmer. An example of such a system is the programming language Prolog which is often quoted as being a knowledge processing environment when in reality it is a programming language whose use requires normal programming skills.

Points I, IV and V are the ones that most concern the present work. They indicate the need for systems which allow the user to communicate in as natural

a way as possible and which are automated as much as possible. In simple terms, there remains a need for systems which allow problems to be described in 'everyday' terms and of which arbitrary queries can be made. For this to be possible it is necessary to develop systems which do not need to be programmed in any sense (other than by adding knowledge) and which find solutions to queries by an automated procedure.

1.2.1 Early Knowledge Representations

Theories of knowledge representation consist or two major aspects:

* Knowledge representation.

* Knowledge manipulation.

In some senses these two areas are quite separate in that the (known) laws of knowledge manipulation are reasonably well understood (although not yet complete) in terms of classical logic. This logic embodies a basic set of fundamental principles that is universal to all logics, and many logics are little more than notational variants of classical logic. This point is made clear by the observation that many logics (example, the alethic modalities) are built by addition of special predicates to the syntax of predicate logic (which somehow fail to capture the fundamental nature of such predicates to the logic since they require additional semantics to be applied as special cases) or are extensions of predicate logic by means of appropriate additional axioms (example, epistemic modalities and axiom K). Where theories differ the most is in what are the basic objects that make up knowledge, although the apparent differences are not perhaps as great in reality as might at first be thought.

1.2.1.1 Representation And Semantic Ambiguity

Most theories recognised that the world is composed of *entities*. These entities are *instances* of *types*, and can be *concrete* entities, such as physical objects, or *abstract* entities, such as states or events. These theories also recognised that entities exist in *relationships* with other entities. Unfortunately, it is possible and all too common to fail in two ways to make clear the semantics of the representations of these notions. We give simple examples to show this point.

The first example, from predicate logic, is:

elephant(Clyde)
old(Clyde)

In this example, the human reader can distinguish by tacit knowledge the difference in meaning between these two predicates. The first states that 'Clyde *is an* elephant' whereas the second states that 'Clyde *is* old'. There is a clear difference in meaning between these two sentences since the first contains the word *an* and the second does not. Thus the first states the type of which Clyde is an instance and the second gives some additional property of Clyde. In addition to the fundamentally different structure of these two sentences there may be different rules of inference applicable to each interpretation. Those who subscribe to the use of predicate logic generally disregard the difference in reading and semantics between these two cases when writing formulae but implementors of systems must provide special purpose code to allow the recognition of each case.

The second example, given here in predicate logic form, although it is more often encountered in network formalisms, we see a commonly used relation:

isa(elephant,Clyde)
isa(mammal,elephant)
isa(species,elephant)

The first formula is intended to state that 'Clyde *is an* elephant'. The second is intended to state that '*an* elephant *is a* mammal'. From these two we would like to be able to conclude that Clyde is a mammal and so state that the *isa* relation is transitive. Intuitively this is correct since elephants are a kind of mammal and that if an elephant is a mammal then so must be Clyde since Clyde is an elephant. The third formula states that 'elephant *is a* species'. If we use the argument that allowed Clyde, by transitivity of *isa*, to be a mammal then we must also conclude that Clyde is a species. This conclusion is intuitively incorrect and results from the fact that the notation does not take into account the different meanings of each use of the *isa* predicate. There are thus two ways in which the *isa* predicate can be read:

* *p isa q*. A specific individual *p* is a member of the set of all *q*s.

* *a p isa q*. The type *p* is a special kind of the type *q*.

The first reading is not transitive whereas the second is.

Further difficulties with representation arise when the naturalness of form and semantics of the symbols used is less than intuitive. We will, in the next section, consider two further cases from the point of view of computability but the readability and naturalness of the representation should also be noticed.

These two examples have clearly shown that we must distinguish between different kinds of predicate. This will be an important principle in the subject matter of later chapters.

1.2.1.2 Computation

The two examples of semantic ambiguity shown in the previous section are compounded by computational problems which result from two sources:

I Some representations are made very complex but do not lend themselves to mechanised computation.

II Logic is intrinsically computationally expensive and any overheads that result from notational ambiguity only serve to compound this problem.

Problem I is sometimes related to the first case of semantic ambiguity in the previous section, where it may be necessary to provide special cases to recognise syntactically identical but semantically different constructs on the basis of labels that they happen to contain. Other instances of problem I result from the nature of the symbols used in the representation. We briefly cite two examples:

Frege's Begriffsschrift

The *Begriffsschrift* is a system of diagrammatic logic devised by Frege [Frege79] [Sowa87]. It is based upon the primitives of universal quantification, negation and implication. Fig. 1.1 illustrates how the simple sentence "there

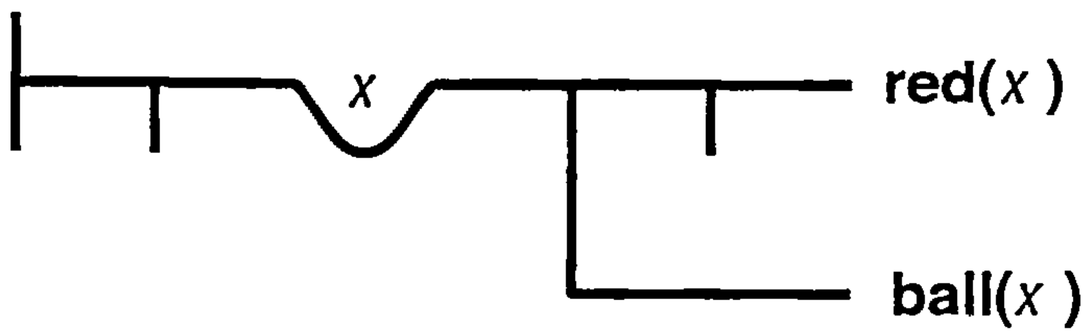


Fig. 1.1 "there exists a red ball"

exists a red ball" is shown. Reading from the left, the initial vertical line is a start symbol. The remaining two short vertical lines represent negation. The hollow represents universal quantification of the embedded variable and the hook represents implication. With these symbols the diagram encodes the explicit sentence "it is false that for all x, if x is a ball then x is not red". Frege took pride in the fact that, because of the use of the particular primitives, the reading of his graphs did not correspond to typical natural language constructs since he felt that natural language was too crude and imprecise. From the computational point of view, the result of this difference between the expression of sentences in natural language and in the Begriffsschrift is that the data structures in a computer that represent simple sentences in natural language will be more complex in an implementation of the Begriffsschrift than they need be.

However, for all its cumbersomeness, Frege's Begriffsschrift admits of a high degree of logical structure in that the scope of quantifiers is explicit and the nesting of the negations is natural. Unfortunately, the semantics of the labels **red** and **ball** are different in the way shown earlier for the predicates *elephant* and *old* and so, for computation, it would be necessary to provide additional machinery to distinguish between them.

Peirce's Relational And Entiative Graphs

Although he worked on the linear notation that subsequently became modern predicate logic, Peirce also produced his system of graphical relational graphs. With these he was able to simplify his notation by the removal of the need for duplication of variables, which he replaced with a single branching 'line of identity'. Relations of any arity between different lines could be expressed by joining each line to the relational node. At first, Peirce adopted the primitives of universal quantification, disjunction and negation to produce his *entiative graphs*

[Sowa87]. Again, from the point of view of linguistic naturalness and hence complexity of computer representation, these suffered from the same difficulties as the Begriffsschrift.

In 1896 Peirce realised that the use of universal quantification and disjunction as logical primitives lead to overcomplexity in his graph logic and moved to his *existential graphs* in which the primitives are existential quantification, conjunction and negation. With these primitives, he was more easily able to map natural language to logic in a way that produced elegant and compact graphs and which allowed a set of very simple rules of inference.

By the use of his existential graphs, Peirce was able to represent the sentence "there exists a red ball" with the following simple graph:

Fig 1.2 existential graph for "there exists a red ball"

This graph is considerable simpler than that in Fig 1.1. As a result its semantics are clearer to the human reader and its straightforward correspondence with the English sentence suggest that the encoding and processing of that sentence in this formalism could be simple and efficient.

1.2.2 Later Knowledge Representations

The foregoing section showed that early formalisms were capable of generating the paradoxes and fallacies observed by Aristotle. It is all too easy with hindsight to criticise the early attempts at knowledge representation; newer formalisms can reject bad aspects and build upon the sound parts of the old. To be of any value, new formalisms must possess two qualities:

I They must be more expressive than previous, restrictive formalisms, with well defined semantics.

II They must be computationally tractable.

The first of these points is one that is universally recognised. Most of the formalisms that currently exist contain, or are based upon, a central core of either predicate logic or some network formalism which is then extended with esoteric symbols, brackets and boxes added for special purposes. And yet many of these notational additions are not actually necessary; the expressive power of first order predicate logic is already complete. What is lacking is a proper representation for predicates of differing fundamental nature (types, relations, attributes) and notions such as contexts and general modalities along with uncertainty.

Point II is the downfall of complex notations. Even a simple logic is potentially intractable when the normal means of proof is by attempting to construct a formula from several axioms and rules of inference. By adding extra notation the complexity becomes even greater and the number of possible interactions between differently constructed sentences which mean the same thing raises problems of computational completeness. Indeed, the trend has been to add expressiveness to a formalism apparently at the expense of any real consideration of how any logic is meant to operate on it.

We now offer three examples which illustrate how greater detail and expressiveness are being incorporated into representations and we consider the semantics and problems of each.

Partitioned Nets

Fig. 1.3 is an example of a *partitioned net* [Hendrix79] [Sowa87], which is an example of the more general graphical approach to knowledge representation, the *semantic net*. An increasing appreciation of the significance of contexts is exemplified in this particular formalism (although this is not the case with all semantic nets). Unfortunately, since the contexts overlap, there is no syntactic information contained within them about how the contents of one relate logically to the contents of the other, nor, therefore, the direction in which proofs must proceed. This information is contained within the (implication) node. and the arcs coming from it in this case and therefore a special purpose algorithm to detect this pattern is required in order to make the above structure computable. One further difficulty for implementors in this particular formalism lies in finding an efficient binary representation for the overlapping but non-

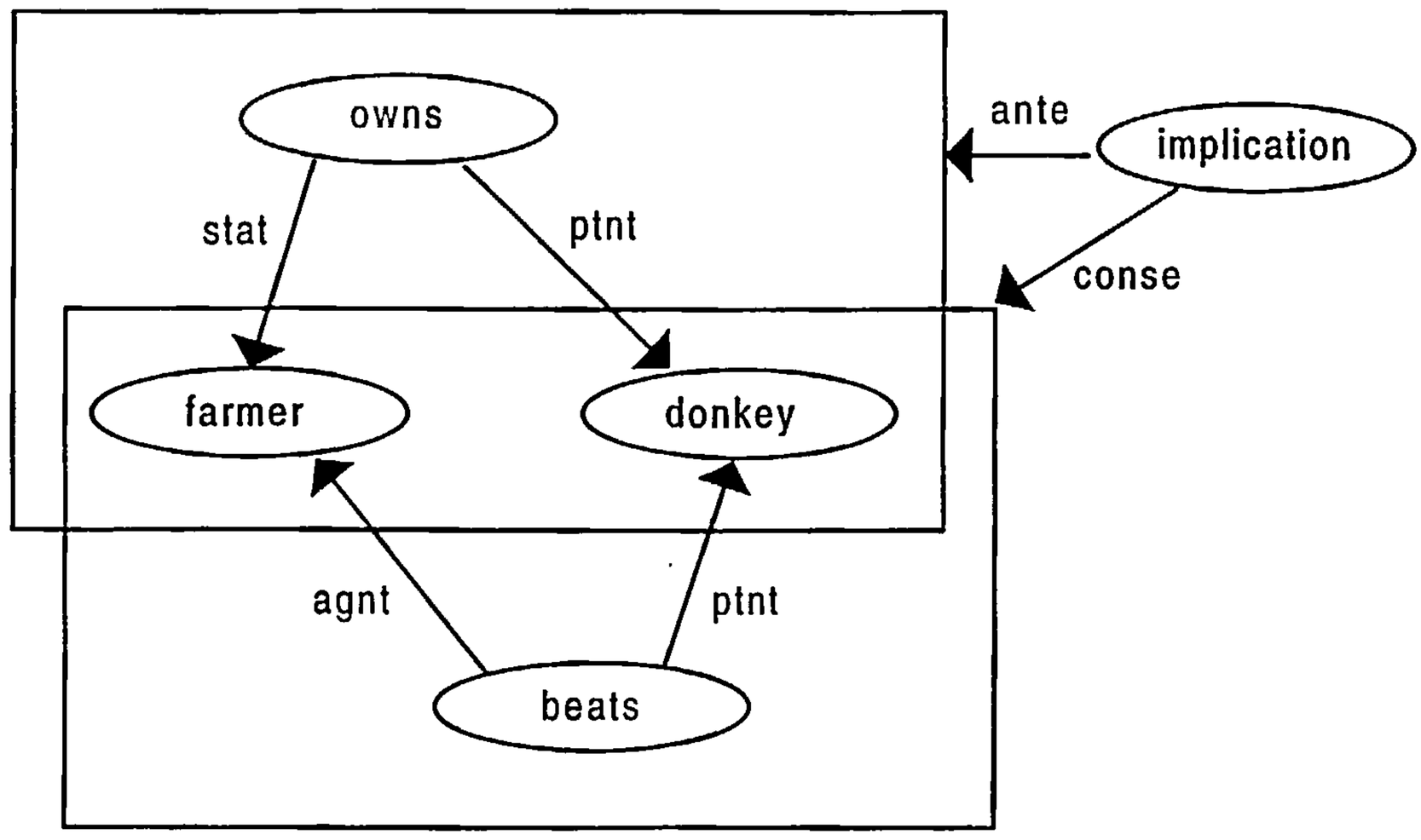


Fig 1.3. "if a farmer owns a donkey then he beats it"

nested contexts. Another semantic difficulty is the observation that each node is identical apart from the label that happens to occur in them, and thus Aristotle's fallacies apply.

It is significant that, as in many cases where a formalism has been mechanised, partitioned nets have been used to implement specific systems [Hendrix79] [Duda79]. They do not in general appear to be used to implement general purpose reasoning systems. We suggest that one of the reasons for this may be the remaining lack of semantic precision which is exhibited by the formalism.

SNePS

SNePS [Shapiro71] [Shapiro79] [Sowa87] is a typical semantic network formalism. It attempts to represent a propositional network and adopts a 'flat' approach to this. Thus the sentence "John believes that a dog is eating a bone" is shown by Fig. 1.4.

The nodes P1, P2, P3 and P4 represent complete propositions (by representing the roots of parse trees for sentences) and the nodes I1 and I2 represent individual identifiers. From our point of view we make the following observations:

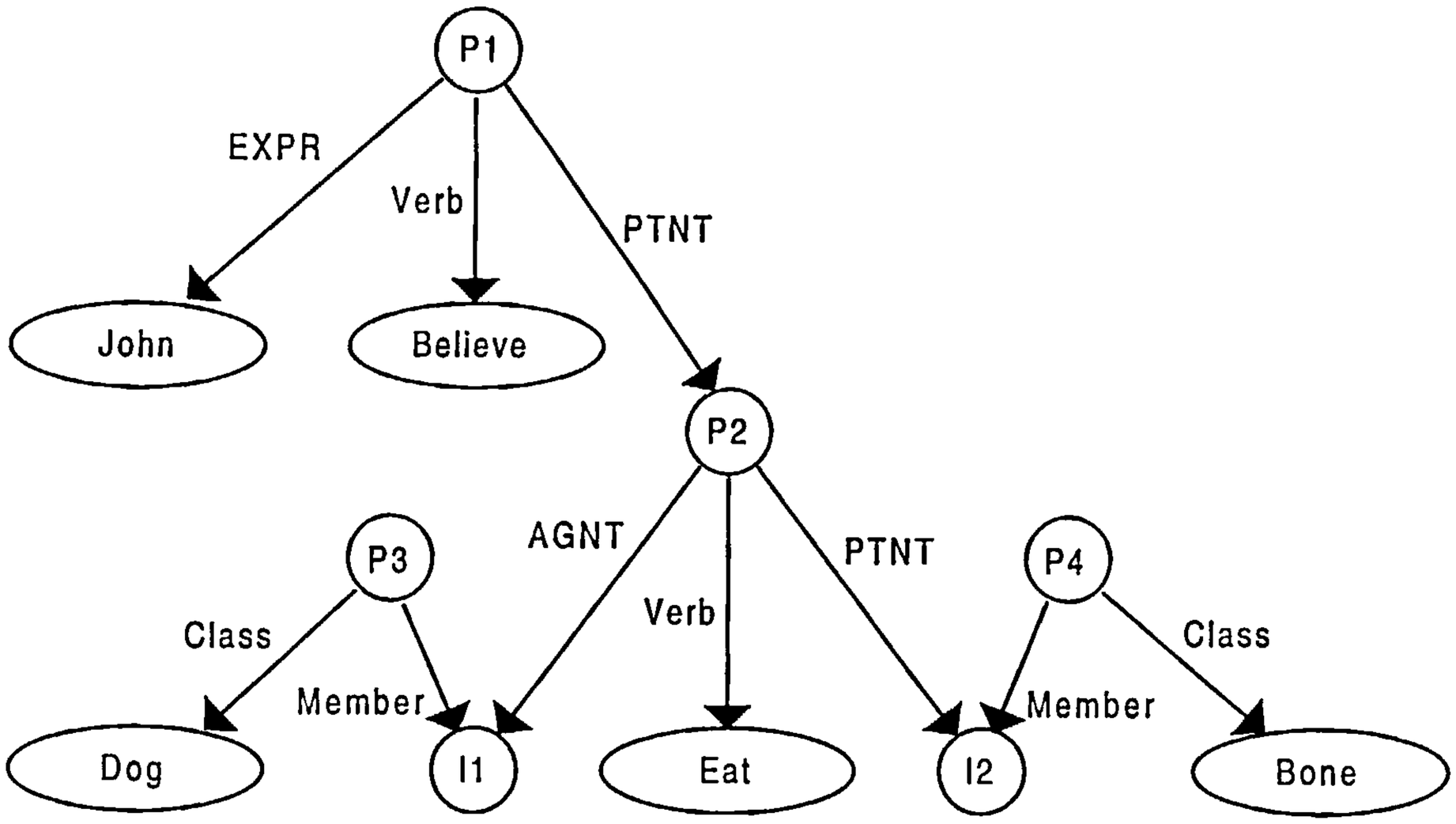


Fig. 1.4. "John believes that a dog is eating a bone"

* Once again we see identically formed nodes with completely different semantics (nodes P*x* and I*x*). Nodes of type P are propositions and nodes of type I are individual markers.

* The person John is shown by an oval node with the name 'John' placed inside it whereas other individuals are given another kind of node and marker.

* The (John) node may be taken as an *individual* marker for John since for John to believe anything he must exist. There is no other kind of node in this graph to denote John's existence. Also, the (believe) node may be interpreted as an individual marker for a belief since if John believes something then there exists an instance of believing. This suggests that the oval nodes represent individuals. Nevertheless, the individual marker semantics for the (Dog) and (Bone) nodes cannot be valid since these nodes are intended to represent *classes* of entities.

* For John to believe a proposition, that proposition must exist and thus the lower section of the graph, from P2 downwards,

can be drawn on its own. When this is done the nodes (I1) and (I2) are 'promoted' to the outer context, which implies the actual existence of some entities. But, John's belief in the existence of the *proposition* P2 does not logically imply the existence of *any* of the entities mentioned in P2. Thus the 'flat' representation of propositions as objects of beliefs leads to paradoxes.

We see once again how relatively modern graphical formalisms do not always possess clear semantics even in the light of past experience. As a result, the onus is on the implementor of a system to provide the implementation with the correct semantics. Since the nodes of a SNePS graph are not adequately defined an implementation will inevitably involve additional searching and checking for special labels in order to find out how to process the nodes which contain them.

Sorted Logics

Later formalisms of logic have adopted a strong *typing* principle. In such logics, each individual marker is assigned to some *sort label.* This approach has advantages in reducing search spaces and making the logic more human-like whilst also making the semantics of the sorted nodes much clearer. Thus we might see the following *sorted logic* constructs:

Clyde : elephant	(sort_label : individual_marker)
old(Clyde)	(predicate(individual_marker)
∃(x:elephant) old(x)	

The use of such logical representation allows the semantics of any formula to be completely unambiguous, mapping more closely to the syntax, and thus to allow proof procedures to be mechanised with the confidence that the correct results will be produced.

The degree to which the notion of typing and strong semantics has been introduced into graphical notations will be considered in Section 1.3 and will form the greater part of the discussions in Chapter 2.

1.2.3 The Future

The extent to which criteria I and II have been met by modern knowledge representation formalisms is summarised by Webster [Webster88] which demonstrates that as formalisms have developed in expressiveness they have also increased linearly in computational complexity. This is perhaps to be expected, but the more recent formalisms are now so much more complex than earlier ones that the increase in expressiveness is offset by the increase in complexity. This complexity implies many simplifications that must be made in computer implementations for efficiency and other pragmatic reasons. We have seen that modern graphical formalisms generally retain the lack of clear semantics of older formalisms and we therefore argue that, for all their expressiveness, these complex formalisms are a hindrance to the development of AI. Indeed, workers in advanced formalisms have so far failed to produce a commercially viable system that approaches the potential of the formalism and this would appear to justify Webster's claim empirically.

We have examined the reasons why theories of knowledge representation and manipulation are becoming more and more complex and expressive and yet are not producing working systems that manage to reproduce the possibilities of which these formalisms are potentially capable. The following sections discuss one further knowledge representation and show how it overcomes many of the difficulties so far described. It is also argued that even this formalism has its problems and thus provides the background to the present work.

1.3 CONCEPTUAL GRAPHS

At the extreme end of Webster's spectrum of expressiveness is Sowa's theory of Conceptual Graphs [Sowa84]. At the extreme end of Webster's spectrum of computational complexity is the same formalism. Nevertheless, we intend to show that the Conceptual Graphs formalism offers a rich representation with sound, natural semantics (is strongly typed and employs existential quantification, conjunction and negation as primitives) and that the computational complexity of this formalism can be tamed by the adoption of novel proof strategies. In doing so we will show that computational complexity

within Conceptual Graphs theory arises not from any intrinsic property of knowledge but from Sowa's attempts to make Conceptual Graphs theory as representationally and logically comprehensive as possible at the expense of parsimony. Evidence for these points comes from two sources: Sowa's theory itself and the results of attempts to implement systems based upon its operations. We briefly review a selection of these in Section 1.4.

Part two of the thesis deals with aspects of redundancy and remaining semantic problems within Conceptual Graphs theory. It discusses each part of the theory, representation and operations, in detail and distils from the original formalism a *rational reconstruction* which maintains the full expressive and computational power of the original theory. In addition, consideration is given to the logical basis of Conceptual Graphs theory, the logic of Charles S. Peirce, and much of the discussion is couched in the terms of his system of logic.

1.4 IMPLEMENTATIONS

1.4.1 General Observations

Many workers have attempted to use conceptual graphs to model problems statically and others have produced systems that are capable of performing some, or even all, of the operations that are defined by Sowa. Indeed we have produced conceptual graphs processors that are capable of performing, at the specific request of their user, a wide range of operations on conceptual graphs. These systems work by accepting graphs and building some kind of model (although the theoretical basis of these systems is sometimes weak or *ad hoc*). The user of the system makes queries by entering a query graph and then selecting specific graphs from within the model with which to attempt to satisfy the query. Operations on the query and the model are selected by the user. In this way the user needs to know how to derive the answer to his query in order to generate it with the help of the system. Whilst these systems have performed their stated tasks and perhaps shown that operations on conceptual graphs can individually be implemented efficiently, few of them have demonstrated that a fully automated system based upon Conceptual Graphs theory is capable of anything more than relatively simple database-like functions and none have taken the interactions between operations into full account.

Some systems are more advanced than the simple Conceptual Graphs based systems in that they are capable of performing certain kinds of inference or deduction, but none of this group has ventured beyond attempting to emulate pre-existing systems such as Prolog or rule-based systems in general. Indeed, it is not normal for such systems even to use the logical formalism of Conceptual Graphs, Peirce logic, as their logical basis; most use some special notation and inference procedures either to circumvent problems caused by Sowa's original theory or to provide a suitable basis for modelling some other formalism. Workers in Conceptual Graphs theory see the computational complexity which arises from the wealth of objects and operations defined within the theory and do not (indeed cannot) attempt to implement a 'complete' conceptual graphs based processor. Instead they simply use a subset of Conceptual Graphs theory to reimplement systems that already exist within other formalisms or to produce dedicated systems. Whilst this approach to system design is undoubtedly valuable in testing aspects of the theory against others on equal terms there comes a time when it is necessary to go beyond this and exploit the theory to the full.

Indeed, what is required is the vision to see that it is not enough to produce a 'conceptual graphs processor' but that it is necessary to build a whole reasoning system for which conceptual graphs are merely the vehicle. When this is done it may be argued that it does not matter which formalism is used eventually since they are all equivalent, logically, to predicate logic. However, we will show that clones of predicate logic are capable of full first order logic but that Conceptual Graphs theory is capable, with some adaptation, of providing a more natural vehicle for other forms of reasoning.

1.4.2 A Brief Review Of Some Existing Systems

In order to demonstrate some of the points made we must now turn our attention to some existing systems. These will fall into the two categories previously identified:

* Systems designed to use Conceptual Graphs operations on conceptual graphs.

* Systems designed to reproduce other formalisms.

The purpose of this very brief review is to show how the experience gained by the study of these has lead to many of the ideas behind the current project. Where deficiencies within systems are identified it is not the aim to criticise these systems in any way but simply to learn the appropriate lessons.

1.4.2.1 Systems Designed To Use Operations On Conceptual Graphs

This section will look at systems produced at Loughborough University in the years preceding the development of the present system. Two of the systems to be considered, by Ho [Ho91] and by ourselves [Heaton89], are built upon largely the same design requirements. In view of this the detailed review will concern Heaton89. This is especially relevant since it was during the development of this system that many of the ideas of the current work came into being.

We will also briefly consider the system of Tsui *et al* [Tsui88] [Chan88]. It appears from this system that implementors of conceptual graphs based systems often start by adopting the same general approach and so it is appropriate to consider Tsui88 along with Heaton89. In this way we will discover commonly encountered difficulties with Conceptual Graphs theory.

The final system to be reviewed in this section is one which was produced by Smith [Smith91] and represents a case in which Conceptual Graphs theory has been used, with some success, in the development of a dedicated system.

Heaton89

Heaton89 was started by Smith. It originally provided a small number of conceptual graph operations (join, maximal join, type expansion, type contraction, projection) which were available as commands and which operated upon named individual graphs. Since the range of available operations was limited the purpose of our project was to provide the missing operations as defined by Sowa with the intention of testing the utility of Conceptual Graphs theory in practice, albeit manually.

Additionally, we added the ability to use contexts of the form that Sowa describes. In order to do this it was necessary to build into the system the privileged type label PROPOSITION, the monadic relation NEG and to add set

referents. We also implemented the ~[ ] shorthand notation for contexts. This allowed the expression of arbitrary contexts in the form depicted in Sowa. Once this was complete the way was set to provide a set of commands to implement the rules of Peirce logic. It soon became apparent that these were not as simple as the seven basic rules that Sowa gives since, as will be shown in Chapter 3, they can be broken down into many, more primitive rules. However, it was possible, for the first time at Loughborough, to build up a set of graphs (although the model theoretic basis of this system was weak, in particular in its use of individually named graphs) and to perform logic using this system.

Nevertheless, and taking into account the fact that Heaton89 was not intended to be a grand synthesis of Conceptual Graphs theory we realised as a direct consequence of 'hands on' experience, that the basic approach of this kind of system was fundamentally flawed in the following ways:

* The use of named graphs, and the necessity within the philosophy of the system to retain their individuality (common to many systems), prevented the use of a soundly developed model theoretic approach to the maintenance of the knowledge base. Indeed, there was no attempt made to verify the consistency of the graphs. It also implied that a user needed to know which graphs contained the answer to a simple query. Thus the operation of projection was rendered somewhat unnecessary.

* The use of Sowa's notation and semantics for negation prevented the operation of projection from working properly on arbitrary graphs and also meant that the generation of inferences by use of Peirce logic did not always result in quite what was expected. In particular, the use of the special type and relational labels showed that, although these occupied certain syntactic positions within a graph they did not behave in the same way as other labels in the same positions. For example, the arbitrary monadic relation:

(REL)->[TYPE]

can be changed by erasure (to be shown in Chapter 3) to:

[TYPE]

whereas the relation:

(NEG)->[PROPOSITION]

must not be changed to:

[PROPOSITION]

since this alters the truth assignment of the graph.

* The use of individual commands to invoke each operation did not bring to light ways in which graph operations interact when chained together. Such interactions subsequently turned out to be a major area of research in the present work.

* The use of separate user-initiated procedures to perform Peirce logic operations meant that the user needed to understand Peirce logic in order to derive the required result. To overcome this it would be necessary to *completely* automate the procedure. Whilst the need to automate was obvious it was not clear how this was to be achieved since there was not any pre-existing methodology for theorem proving in Peirce logic.

* Sowa's use of concepts of type PROPOSITION is not backed up by any formal means of placing non-nested graphs within the referents of these concepts or of extracting them later. This operation had to be fabricated within Heaton89. Thus Sowa allows graphs of the form:

[PROPOSITION : { *graph* }]

graph

in which each graph implies the other (*sic*) and yet there is no formal means of deriving one from the other. Indeed, we will show later that the equivalence is not true.

* The complexity of Sowa's representation of negation, and thus the design and programming effort required to 'make it work', started to seem incongruous and unnecessarily great. This is particularly the case when we recall that Peirce simply enclosed his negated contexts within an oval.

It must be noted that the development of Heaton89 was undertaken at a time when we were relatively unfamiliar with Conceptual Graphs theory. Having said that it is also significant that the lack of prejudices implied by this lack of preconceptions may have allowed us to take a more objective view of the problems with Heaton89 and thus have lead to a clearer view of some of the ideas that are incorporated into the present work.

Some of the deficiencies described above can be dismissed as nothing more than obvious flaws of a relatively unprincipled and incomplete system. Some cannot, in particular the use of the Sowa form of negation and its implications for standard operations such as projection, and the lack of a proper model theoretic approach to storing knowledge. Conceptual graphs systems from other sources which possess any form of negation at all also continue to use the original Sowa form of negation. The experience with Heaton89 showed that it was not possible to use the Sowa form of negation in a way that did not require the development of special purpose algorithms and a range of exceptions and special cases. Also, the use of named graphs is widespread.

Tsui88

This system was originally developed by Tsui as part of his doctoral work and the development of the system has subsequently been continued by others. In many ways it is more powerful than Heaton89 since it exists in a more developed and useable form. Nevertheless, the fundamental approach of Tsui88 is similar to that of Heaton89, in that the knowledge base consists of a set of individual named conceptual graphs and the system is driven by commands which summon conceptual graphs operations.

From our point of view there are two relevant aspects of this system. The first is that the graph representation is complex and each concept possesses, in addition to its referent, an additional identifier which is known to the user (*i.e.* is

not simply some internal label). Indeed, two copies of some concept (where both copies are logically the same concept) within the same context possess different identifiers. The inclusion of additional information in this way necessarily complicates the operations on conceptual graphs since the additional information must be accounted for in a principled manner.

The second relevant aspect of Tsui88 is the inference engine. It employs a modal unification (*sic*) technique to implement *modus ponens* in which a clausal form with the following syntax is used:

[IF : { *antecedent* }]->(IMP)->[THEN : { *consequent* }]

The modal unification technique can generate consequents by instantiating the antecedent (by projecting it onto some graph) and then asserting the consequent. Whilst this approach will certainly generate consequents from antecedents we must also note the following properties:

* The generation of the consequent as a free standing graph requires an axiom of the form:

  [IF : { *antecedent* }]->(IMP)->[THEN : { *consequent* }]

  $\wedge$ { *antecedent* } $\Rightarrow$ { *consequent* }

  In order to make this work the system possesses a dedicated algorithm. In order to provide other logical operations a suitable algorithm must be added to the inference engine.

* The antecedent cannot be shown to be false when the consequent is false. To do so would be to require an additional algorithm and a means of expressing falsity, which is absent in this system. Thus the system could become inconsistent.

Much more can be said about the paradoxes which arise from the expression of logical connectives by the use of special type labels and relational labels but we have made the point. These points demonstrate that the semantics of such graphs are quite different to those of 'normal' graphs and that the semantic differences are reflected in the need to prevent certain derivations from being allowed. We argue that *all* conceptual graphs *must* be given the same

semantics since they all look the same to a computer (and indeed to humans). In addition, the system does not support negation of any kind, except for that hidden in the IMP relation. This point is regretted by Lukose who worked on the system in the years up to 1992 [Lukose92].

The approach adopted in this and other similar systems of using a relation such as IMP is suggested in Sowa's presentation of his theory. Unfortunately it must result in false systems. Nevertheless we argue that some form of deduction or theorem proving is necessary to enable a system to maintain its consistency but we will advocate a return to Peirce logic for the purpose.

Smith91

The work of Smith [Smith91] resulted in a working system (CAMES) that was used by the business corporation Reuters and was designed to provide a system capable of detecting conflicts in specifications of computer based services and associated equipment provided by Reuters. Customers' records were used in conjunction with new orders for additional services. A model (or several) of a customer's new configuration which would result from supplying an order was generated. The model was then tested against a knowledge base which contained a small set of 'schematic rules'. These rules, which were syntactically similar to Sowa's schemata and were not rules in the normal sense, represented disallowed configurations, or configurations which would result in the customer being provided with the same service more than once. If a projection of a schematic rule onto the model was found then the configuration was not allowed. By reporting which rules had been used to show the configuration to be disallowed, CAMES could explain its reasoning.

Smith compared the CAMES system with an equivalent system built with a well known expert system shell. He showed that the use of schematic rules in this way greatly reduced the number of rules required as compared with the expert system (from around one hundred in the expert system to around ten in CAMES) without reducing the power of the system. Indeed, Smith was able to increase the number of service and equipment configurations catered for from about 100 in the expert system to 1000 in CAMES. In trials, he found that users would have greater confidence in CAMES since the approach of building models and testing them in this way fitted their view of the problem domain.

The CAMES system demonstrated that a system based upon a relatively simple subset of Conceptual Graphs theory (a model consisting of simple graphs, a catalogue of graphs which represent exceptions and the operations of maximal join and projection) could be as powerful (in this particular domain) as a much more complex but less intuitive traditional expert system.

Nevertheless, the system did not make any use of general deductive or theorem proving techniques, nor of semantic constraint checking based upon a canonical basis. In particular, the checking of the logical consistency of the schematic rules was incumbent upon the knowledge engineer. Whilst these limitations may be satisfactory for small systems with around ten rules it may not be for much larger systems. Also, there may in general be a need for a more complete logical system for the checking of models since it is unlikely that any model is guaranteed to be deductively closed and all knowledge explicit. Further, it is anticipated that future systems would interact with users to gain further knowledge. For this to be sound there must be a full set of consistency checking procedures that the system automatically applies to all new information.

1.4.2.2 Other Approaches

The previous section showed that the use of special purpose routines, acting on special cases, to produce deductive systems was likely to result in a logically flawed system and that restricted systems were satisfactory for their specific purpose but were not sufficiently general and were thus inflexible, making them too restrictive for general users. The alternative approach is to use a more general theorem proving approach. Within this approach there are two possibilities:

I Reimplement an existing formalism in conceptual graphs, retaining as much of the character of that formalism as possible.

II Implement a radically new approach based on a more pure form of Conceptual Graphs theory.

Option I has some appeal. Either an existing theorem prover may be adapted by simply adding a Conceptual Graphs 'front end' or a new system may

be designed to take advantage of the features of Conceptual Graphs theory which do not exist within all logics, such as semantic constraint checking and the use of the type and relational hierarchies. This was the approach of Fargues *et al* [Fargues86] [Fargues88] who implemented a Prolog-like linear resolution system in conceptual graphs and of Tjan *et al* [Tjan89] who implemented a form of resolution. We will not describe these systems any further here (since we will draw upon these techniques later) but we will take note of the observation that in each these case it was considered necessary to extend the notation in some way. Such extensions were similar to the special type and relational labels encountered earlier but further extensions to the referent were also incorporated.

Option II is the approach adopted in the present work and its use has been prompted by two main driving forces:

* The approaches described in 1.4.2 have been shown to be inherently flawed and it is time to abandon them.

* The approach of option I does not add anything new and simply repeats that which has already been done in other formalisms. Not only that but other formalisms may have been designed with a particular purpose in mind and are likely therefore to be better than Conceptual Graphs theory.

1.4.3 Further Considerations

An additional aspect of the production of logical systems is the nature of logic itself. In this context we note that whilst it is relatively easy for humans to generate proofs it is often the case that much experience is brought to bear when deciding how a proof should proceed. This kind of reasoning is usually known as 'case-based reasoning', but the underlying mechanisms must surely remain constant between domains. It is widely recognised that many logics are intractable and not amenable to computer implementation. For this reason, workers in AI expend much time and effort in devising heuristic approaches to problem solving and many argue that uniform methods of proof can never succeed. Each domain must possess its own heuristics and methods which must be specially encoded rather than stated declaratively. This seems counterintuitive and contradicts the underlying constancy of the case-based

reasoning mechanism that we postulate, and we will show that it is possible, by adopting a particular approach to proof, to use a proof strategy with a built-in domain-independent heuristic which guides any proof.

The previous section demonstrated that most attempts to produce systems of reasoning with Conceptual Graphs theory have either lacked a strong theoretical basis for their design or have replicated preexisting systems. Whilst any resemblance to the workings of the human mind shown by Prolog-like systems and expert systems is tenuous we believe that psychological considerations must nevertheless be made, otherwise future systems based on whatever formalism will be nothing more than special purpose tools and will not be properly classified as intelligent.

We must therefore consider how to represent and compute with the wider variety of sentences that the human mind can handle. In this respect we see that Conceptual Graphs theory is incapable of expressing uncertainty and the handling of general modal statements is incomplete. Not only that but humans are able to build several alternative models and test them. They can also recall past states of knowledge (either their own or someone else's) since old knowledge is not forgotten but simply demoted to a position of 'once believed'. All these notions must be representable.

1.5 MOTIVATION AND AIMS OF THE THESIS

The present work was motivated by study of formalisms as described in Section 1.2. This study showed common problems with many logics and representation formalisms. We also noted in Section 1.3 that Conceptual Graphs theory potentially overcomes these problems.

Study of implementations of processors based on Conceptual Graphs theory of the types described in Section 1.4 and their theoretical bases showed additional problems with each and sowed the seeds of possible solutions. Indeed, one of the major sources of inspiration was our own previous work which fell squarely into the set of implementations that suffered from a lack of appreciation of the need to reduce redundancy. What the previous work has shown is that the time has come to stand back and view the scene from a distance and rebuild Conceptual Graphs theory in the light of gained experience.

The main parts of the thesis attempt to quantify the degree to which Conceptual Graphs theory itself is directly responsible for some of the perceived limitations and they propose solutions. In doing so a computational model for a conceptual graphs based reasoning system is developed. This is a comprehensive, mixed strategy system that can accept arbitrary statements and build rigorously founded models of which arbitrary queries can be made and answered completely automatically, along with full explanations of the reasoning. Not only is the system capable of general logical proofs but it also makes full use of the semantic constraint processing component of Conceptual Graphs theory. In this way, all the above mentioned systems are subsumed.

In short, the thesis seeks to demonstrate the following ideas:

* That it is possible to identify a central core of Conceptual Graphs theory as a logically sound and advanced knowledge representation formalism and to produce from it a non-redundant, static, natural knowledge representation theory which retains all the expressiveness of the original theory.

* That a non-redundant set of operations with their logical basis founded in the logic of Charles Peirce can be identified.

* That a computational model of a deductive mechanism and theorem prover with advantageous characteristics based on these operations is both possible and its implementation reasonably efficient.

* That this computational model will principally describe a methodology which both incorporates domain-independent heuristics and shows how to integrate several strategies in an efficient and seamless manner.

* That a system based upon a suitable computational model will exhibit performance superior in both semantic unambiguousness and functional completeness to systems based upon other formalisms.

1.6 PLAN OF THE THESIS

The thesis is constructed in five main parts:

Part 1 This part is the introductory section and consists of Chapter 1.

Part 2 This part consists of Chapter 2 and Chapter 3 and discusses the theory as presented by Sowa in 1984 and as developed by him and others since. Resulting from this analysis, the complete theory of static knowledge representation that is required in later parts is presented.

Part 3 This part consists of Chapter 4 and Chapter 5. It discusses the development of a computational model for a reasoning system based upon the static representation formalism and operations developed in part 2. In addition, a working prototype written in Prolog and based upon the computational model is introduced. Evaluation of this prototype is carried out and examples presented.

Part 4 This part consists of Chapter 6 and discusses the general conclusions of the thesis. In this part attention is given to discussions about how Conceptual Graphs theory compares with other formalisms and these discussions result in the identification of remaining weaknesses within Conceptual Graphs theory and within the computational model developed in part 3.

Part 5 This part consists of references and appendices.

CHAPTER 2

CONCEPTUAL GRAPHS AND KNOWLEDGE REPRESENTATION

2.1 INTRODUCTION

In the introduction to knowledge representation formalisms in Chapter 1 we suggested that the general deficiencies found in many other formalisms were corrected in Conceptual Graphs theory. The theory of Conceptual Graphs contains descriptions of a wide variety of primitive objects and includes two complete sets of notation. Without going too deeply into the philosophical and psychological backgrounds to these objects, this chapter will review them and show where they are well founded logical primitives and where they are less appropriate. In doing so it will provide a rational reconstruction of the theory from the point of view of representation and semantics within a representationally rich, but lean, and, above all, computable formalism.

Throughout the discussion it is assumed that the reader is familiar with Conceptual Graphs theory and therefore the standard terms are not defined as they are introduced except where their definition requires clarification. Also, examples of graphs will be given before the formal definitions for some of their components have been given. Where this occurs it may be assumed that the syntax and semantics reflect Sowa's original conception.

2.2 ESSENTIAL CONCEPTS

The theory of Conceptual Graphs is derived from earlier theories of Entity-Relation theory, frames, semantic networks, order sorted logics, predicate logic and theories of human perception. As such it attempts to be a distillation of the best features of each. Unfortunately, it also attempts to be everything to everybody and as such it retains some of the less good features and traditions of some of the theories from which it is derived.

The fundamental principles of knowledge that Sowa's Conceptual Graphs notation attempts to capture are:

I Concepts - entities, events, states and attributes are discrete and can be referred to individually. Concepts consist of a type label and a referent. The type label specifies the type, or sort, to which the individual, whose identity is specified by the referent, belongs. The type label of the concept states implicit properties of the individual that it represents.

II Relations - concepts are linked by the relationships that exist between the individuals represented by them. Relations are also of different sorts.

III Definition - new types and relations are defined in terms of more general types and relations.

IV Inheritance - the type labels and relational labels are partially ordered. Those lower down the partial order inherit all the properties of the more general ones in terms of which they are defined.

V Selectional constraints - a rigorous approach to the checking for selectional constraint violations is made possible by the use of catalogues of canonical models and a set of canonical formation rules.

VI Logic - there is a small but linguistically and psychologically intuitive set of logical primitives (existential quantification, conjunction and negation) that allows arbitrary logical relationships between concepts and relations to be expressed, and a set of inference rules with which to operate on these logical relationships. In addition, the logical status of all components of the system is well defined.

VII Modalities - a general means of representing arbitrary modal statements by use of contexts nested within concepts. Contexts also provide a vehicle for the generation of theories

and models for explanation and prediction.

VIII Model theoretic semantics - a sound, open world model theoretic basis for Conceptual Graphs provides an accepted semantics for reasoning.

We will demonstrate that the theory of Conceptual Graphs goes a long way to achieving these aims. Nevertheless, it also contains compromises and polymorphism [Esch89] in its notation (and hence its semantics) that impinge upon the success with which some of the goals are achieved.

2.3 CONCEPTUAL GRAPHS

Conceptual graphs fall into two kinds which we will call *simple graphs* and *compound graphs*. This section defines simple graphs. Compound graphs cannot be dealt with until the subject of negation has been fully covered. The definitions of the terms *simple graph* and *compound graph* that we adopt will differ slightly from Sowa's.

Sowa's definition of a conceptual graph is that it is a connected, bipartite, directed graph. The nodes are *concepts* and *relations*. Each relation is connected to concepts by one or more arcs, which are labelled if there is any ambiguity. Concept nodes represent *entities*, *events*, *states* and *attributes* and relational nodes represent relationships between concepts. We also add that *modal statements* are represented by being nested within concepts. This is necessary to allow modal statements to possess modal attributes and to allow them to be referred to in the standard manner. Sowa suggested this [Sowa84] when he allowed graphs to be placed within the referent field but he failed to formally add modal statements to his list of entity types that concepts represent.

Notation

Within this, two methods of drawing conceptual graphs are defined. The ideal form is the *display form,* which is a purely graphical notation. An alternative form, which is provided for use with text based computer terminals, is the *linear form,* which is intended to be a character based equivalent with a 1:1 mapping to display form. As this chapter progresses the linear form will generally be used, but the display form and the mapping between linear form

and display form will be noted. This is more than an academic exercise since there is one area of the mapping that can lead to logical inconsistencies when the linear form is used. The discussion of this requires some logical equipment which is covered in the next chapter and thus it will be given in Section 3.6.6.

According to Sowa, for any conceptual graph one of the following is true:

* The graph consists of a single concept node.

* The graph consists of a mixture of concept nodes and relation nodes in which each concept node is attached only to relation nodes and each relation node is attached only to concept nodes.

Whilst both of these assertions are undoubtedly true, there will be further discussions in Section 2.16, in which it will be shown that Sowa's definition of a conceptual graph is somewhat restrictive and that his definition has repercussions on the operations on conceptual graphs that he defines. We now provide our definition of a *simple conceptual graph*:

Definition 2.1: Simple Conceptual Graph

A *simple conceptual graph* or *simple graph* is a finite, connected, directed, bipartite graph. The two kinds of node are:

* *Concepts*, which represent instances of *entities*, *events*, *states*, *attributes* and *modal contexts*.

* *Relations*, which represent relationships between such instances.

Any simple graph *g* may consist of a single concept node or of one or more concept nodes, each connected to all other concepts within the graph *via* arcs to at least one relation node in *g*. Each arc of each relation must be connected to a concept and no concept can be connected to another concept.

2.4 THE CONCEPT NODE

A concept is made up of two fields, the *type label* and the *referent*. These are placed within a box in display form and between square brackets in linear form. In each case they occur in the order TYPE : REFERENT and are separated as shown by a colon.

We start our discussion of the concept node by surveying the wealth of objects than can be found within it and which are described by Sowa and others.

2.4.1 The Referent

The referent field is one area in which conceptual graphs theory is rather more relaxed than we would prefer. Sowa gives a whole range of referents and in doing so makes operations on this field extremely complex. At the same time he compromises the semantics of the concept as a whole since different referent types require different interpretations. Other workers continue this trend by adding more and more information to the referent field [Fargues89] [Tsui88] [Tjan89]. In order to provide a single semantics for the referent we define the referent field as containing a simple index (which may be a variable) to identify a particular individual. Any further information about an individual is placed within the definition of its type.

The following sections show and discuss a selection (since the real list is of indeterminate length and composition) of the referents that Sowa suggests. These discussions are intended to give the general flavour of our reasons for wishing to rationalise the referent.

2.4.1.1 Individual Marker - *#n*

Since a concept represents an individual entity then the *individual marker* represents an index to a particular entity within the model. It therefore provides a unique reference number for each individual. The existence of such reference numbers greatly simplifies the access to the attributes of any individuals. It remains an open question as to whether the individual referent possesses any psychological validity but we retain it as a computational and implementational

device since we will need to make copies of concepts from time to time and must retain the identity of the entity represented by each copy.

2.4.1.2 Definite Article -

The # referent without a reference number is intended by Sowa to represent the word 'the'. As such it refers to some context dependent, previously mentioned entity whose individual marker is known (possibly). However, the word 'the' is a linguistic device used to refer to previously mentioned individuals in some discourse, which is linear, without the necessity of repeating the identifying description of that individual. Since we may assume that the discourse is being understood as it develops then that understanding will include the instantiation of the word 'the'. Thus the conceptual graphs which represent the discourse will contain the appropriate individual marker. We will shortly apply Sowa's definite article semantics to the generic referent, which renders the # referent redundant.

2.4.1.3 Name - '*String*'

The name referent is intended to be a shorthand, produced by the operation of *name contraction* from a more explicit graph which links one concept *a* to another *b* which expresses the name of the entity represented by *a*. We do not employ the name referent in this form since this would permit polymorphism with respect to the concept of naming. Instead we state that a name referent is a convenient human-readable but contentless surrogate for an individual marker. There is therefore a 1:1 correspondence between any name referent and some individual marker. This means that two different individuals which happen to possess the same name must be given different name referents.

2.4.1.4 Generic Referent - *

The generic referent is taken as being a variable. It nevertheless stands for a particular individual, or at least for exactly one individual at any one time, whose individual marker is (currently) unknown. However, for reasons discussed in Appendix I, we require that the generic referent, if it occurs within an evenly enclosed graph in a knowledge base, to mean that there exists some individual of unknown identity and we wish to attempt to show that this

individual is actually one whose identity is known somewhere in the knowledge base. In other words, the generic referent 'yearns for instantiation'. With this semantics the generic referent represents words such as 'something' or 'somebody' in declarative sentences and represents the 'wh-' words (who, what etc) in interrogative sentences.

2.4.1.5 Coreference Marker - *x

A coreference marker indicates that two different concepts which both contain generic referents with the same coreference marker refer to the same individual. This device is necessary when an individual of unspecified identity must be referred to across a context boundary (duplicate individual markers imply a line of identity by default). Two generic referents thus linked are said to form a *line of identity.*

A line of identity is a connected graph whose nodes are concepts and whose arcs are coreference links [Sowa84], shown by adding a label to the generic marker. However, any two concepts that both contain the same referent of any type are on a line of identity. Sowa appears to make a distinction between lines of identity and coreference links at some points in his theory but at others he does not. We adopt the view that there is no distinction and state that any two concepts can be said to be on a line of identity if they both contain the same coreference marker, the same individual marker or the same context. In this context our definition of a line of identity is identical to Sowa's. As a final point, we note that the generic referent * is also on a line of identity but that the line of identity does not extend beyond the bounds of the concept.

Section 3.6.6 gives a detailed argument about the nature of coreference links with respect to operations on them and makes comparisons between the display form and linear form representations.

2.4.1.6 Measure - @ 3 Miles

The measure referent represents a single quantity of arbitrary complexity. It is generated by a process of *measure contraction* from a more explicit graph which expresses all the relationships involved. Since it is a contraction of another more explicit form we argue that the measure referent is an oversimplification that results in too much information being placed into the

referent field, and which leads to polymorphism and computational complexity. For this reason we do not include it in our rationalised set of referents. Nevertheless, for pragmatic reasons, we will allow natural numbers. This concession will permit the use of numbers in computations without needing to teach a system about numbers in an axiomatic way.

2.4.1.7 Set (various Types) - { ... }

Sowa introduced four kinds of set referent in 1984. Since then he has replaced these with a slightly different, though fundamentally similar formalism [introduced informally in communications with the Conceptual Graphs Mailing List]. In each, the set referent is intended to show that a number of individuals of a particular type are participating in some relation. The manner in which the individuals within the set are related to each other is described by which kind of set of which they are members. Unfortunately, the grouping together of several individuals within a single concept presents semantic problems which are sufficiently intractable to suggest that we do not include the set referent in any form in our rational reconstruction.

Kocura [Kocura90b] presents an alternative treatment of sets. Since this treatment does not require the introduction of any new notational devices it is not necessary to present and define the treatment here.

2.4.1.8 Universal Quantifier - ∀

Sowa permits the inclusion of the universal quantifier in the referent field. Thus he allows a single concept to refer to all instances of a given type. As a result, Sowa and others are guilty of allowing the quantification to vary not only from graph to graph but also from concept to concept within the same graph. This might have advantages in terms of the compactness of the notation but the computational overheads involved in attempting to process such graphs and understand the results is enormous. This last point is one which is not always fully appreciated, even by Sowa. Generic concept nodes are existentially quantified. This fact is important to the understanding of the semantics of Conceptual Graphs theory as a whole as well as to the meaning of the results of the logical operations on them.

To see some of the difficulties that arise from mixing the quantifiers in an

inappropriate way, we consider the concept:

[PERSON : *]

which states that there exists some person. By comparison, the concept:

[PERSON : ∀]

is a rather meaningless construction which states that there exist all people. There is one more problem with this use of the universal quantifier. What does the following graph mean?

[PERSON : ∀]->(REL)->[TYPE]

This graph is stating, possibly unintentionally, that everyone that exists is in the same relationship with the same individual represented by the concept on the right. Whilst this might be what was intended it is more likely not to be the case. For instance, to state in this notation that all married men have a wife the following graph would typically be used:

[MARRIED_MAN : ∀]->(SPOUSE)->[WOMAN]

Sowa himself has used such examples [Sowa84]. Nevertheless, this is ambiguous because the [WOMAN] is stated here as being some particular woman and there is nothing to state that it is a different woman for each man. In general it is not the case that at any moment in time there is one woman who is married to all married men. The confusion arises because the [WOMAN] concept has an existentially quantified generic referent and this referent is misinterpreted as standing for any individual and not, as is the case, for some particular individual. The difficulty, in terms of predicate logic notation, is that the graph does not and cannot specify the ordering and scope of the quantifiers.

Since we are attempting to remove such irregularities we require that the only quantifier allowed be the existential quantifier, in accordance with the existential quantification of Peirce logic. This both simplifies the operations on the referent field, with a resulting saving of computational effort, and clarifies the semantics of the concept node. Additionally, the universal quantifier can be constructed from the existential quantifier and negation and so is not necessary within concepts.

2.4.1.9 Graph - { *Graph* }

The final type of referent that we consider deserves particular mention. Sowa permits sets of graphs to be inserted into the referent field of a concept. In doing so he is providing the representational mechanism for general modal statements. We call such concepts *modal concepts*. This area is one in which the semantics of conceptual graphs is the weakest and one of the consequences of this weakness is a problem with negation that will be discussed in Section 2.9. It is necessary to introduce the general idea of nested graphs before the problem can be dealt with.

Modal Statements

The representation of general modal statements requires that the modal statement itself be regarded as an entity to which one can refer and to which one can attach attributes. In this respect the modal statement behaves in the same manner as any other entity. However, there is a difference since we must not only be able to refer to any given proposition but also to have access to its semantic content. For this latter reason we require that whole graphs be included within a modal referent and we must provide a semantics for such graphs that allows access to the semantic content. Indeed, we wish to place within the referent a set of graphs which is descriptive of some state, event or situation and is self-identifying. Thus a modal concept may represent a single proposition or a complete knowledge base. We may also state that the graphs within a modal concept need not be true; they represent themselves and not some state or event in the real world. In this way, modal concepts can represent theories and completely fictional stories.

We also require a means of reference to contexts to allow us to talk about graphs as first class objects. Sowa provides a means of including graphs within the referent field of a concept, and which we have accepted, but does not state how they may be processed either to get them in, get them out or otherwise reason with them. For instance we would prefer to be able to say something to the effect of 'If there exists a conceptual graph then that conceptual graph is a proposition'. In the standard syntax there is not any means of expressing this. To overcome this omission we propose a reuse of the { * } notation to represent a *generic context*. In addition we propose that there be a generic context such

as { *x } so that modal concepts may be linked by lines or identity in the normal manner. A final requirement is that the { * } construct can stand by itself outside any concept and that it can become instantiated with sets of free conceptual graphs. With this notation and semantics and some further notation to be introduced in Section 2.8.2 it is now possible to encode the sentence 'If there exists a statement then that statement is a proposition'. Since we have not yet covered negation we delay the provision of examples until Section 2.8.2.

2.4.1.10 Summary Of Referent Semantics

We have argued that the referent identifies a particular individual. The foregoing discussions have shown that there are two kinds of referent: individual and generic. We also argued that there are two kinds of individual that may be considered: entities and modal statements. We consider that the remaining referent types either break this semantics or introduce unnecessary complexity and that these referents are therefore unsatisfactory.

Quantification

Each generic referent is governed, by definition, by an implicit existential quantifier and we may thus say that the presence of a generic concept on a sheet of assertion implies the existence of some particular individual. Since this is the case then there are two further points. The first is that where the generic marker is accompanied by a coreference marker, that coreference marker identifies the particular individual by its links to other concepts. In this case, since the individual is identified, the generic marker plus coreference marker is acting exactly like an individual marker. The second point is that, since any generic concept represents some particular individual and any individual has only one existence there can only be a single concept for that individual. This is the basis of model theory [Sowa84]. As a result, each concept on a sheet of assertion is self-identifying.

Equality

Coreference links show lines of identity between individuals. They therefore show equality (identity) between different concepts in different contexts. Where there is more than one line of identity entering a concept this expresses identity between possibly different markers. This is not to say that

any concept with more than one referent is intended to represent a set but simply that should any concept acquire more than one referent during the course of some proof and the concept is shown to be implied by the knowledge base then all the markers within the concept must refer to the same individual. Thus we may be able to use rules of inference over lines of identity (Chapter 3) to show that two different generic markers represent the same individual or that a generic marker can become 'instantiated' with an individual marker. Similarly, a computational system may also be able to show by some inferential means that two different individual markers also refer to the same entity. Thus we do not wish to restrict the semantics of the individual referent to one which forbids two different individual markers from being unified.

Syntactically, identity is shown in the linear notation by the insertion of the = symbol between the markers within a referent. In this context the = symbol is acting only as a separator. Thus we might write the following:

[PERSON : *x] [PERSON : *y]

[PERSON : *x = *y]

This states that there exists a person *x and a person *y and that person *x and *y are the same person.

In the display notation the = symbol is not required since the same notion is shown by allowing several lines of identity to attach to the same concept.

2.4.2 The Type Label

The type label is typically a token that indicates the type to which the individual represented by the concept node conforms. It is usually shown as either a text string, which acts as a simple label, or as a lambda abstraction, which defines the type rather than specifies it by name. Section 2.10.2 gives a discussion on the nature of lambda abstractions in the type field, and more will be said about lambda abstractions in Section 3.4 where operations on these objects will be considered.

It is not possible, in Sowa's theory, to place any kind of variable within the type field. However, by not including any form of variable within the type field Sowa limits the representational power of conceptual graphs with respect to the

ability to refer to unknown type labels or to encode statements that contain phrases such as '...of the same type as...'. Neither is it possible to place a type label within a referent field. As a result it is not possible to refer to types in any general way as first order entities conforming to second order types and a true metalanguage function for conceptual graphs is impossible. The next section discusses these points in terms of the general concept of coreference.

2.4.3 Lines Of Identity And Higher Order Logic

Sowa defines a *line of identity* as a connected, undirected graph *g* whose nodes are concepts and whose arcs are pairs of concepts, called *coreference links*. This definition of a line of identity makes reference to whole concepts rather than to the referent field only. Nevertheless, we would like to be able to reason with type labels as if they were individual markers. We justify this by showing that we wish to connect fields of any type with lines of identity. Such a case would occur when second order entities must be referred to in a first order manner.

We take as an example a case where we might wish to ask some system a question of the form 'what kind of thing is Clyde?'. This would be a very simple question for a human to answer. In the linear notation this is straightforward; we simply allow the use of generic markers within the type field:

[* : Clyde]

A query such as this is satisfied by unifying the generic marker with the type label to which Clyde conforms. It could be argued that the use of the universal type T in place of the generic marker would achieve the same effect. In this particular case it would. But, it might be necessary to enquire what species Clyde is a member of. In this case we require something such as:

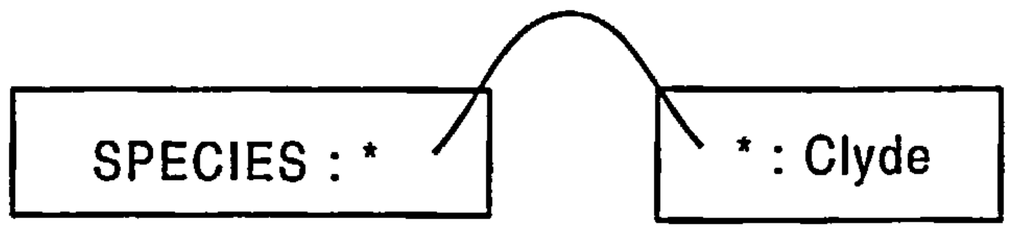
[SPECIES : *x] [*x : Clyde]

Without the specific link between the referent field of the second order concept [SPECIES : *x] and the first order concept [*x : Clyde], this query is impossible. In order to make the link as conceptually simple and computationally efficient as possible we adopt the same notation for the generic marker in a type field as in a referent field. This identity of notation reflects the notion that the object for which the generic marker stands is the same in each

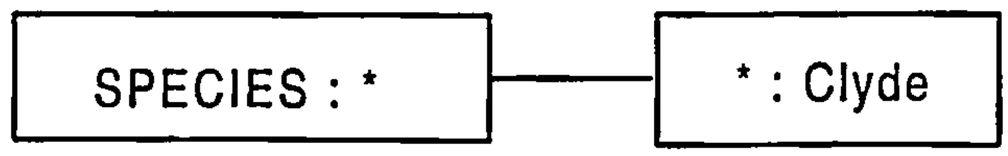
case. The factor that makes the semantic differences between each occurrence of the marker is not the syntax of the actual marker but its position within the concept box.

We have now obtained the result that coreference links must be shown to join not whole concepts but the appropriate fields within concepts. We have used linear notation in this part of the discussion. In the linear notation, coreference is shown by adding a label to the coreference link. Section 3.6.4 discusses a logical pitfall concerned with the use of named generic markers to show coreference in linear form notation which does not occur with the display form. This pitfall represents a departure from the 1:1 correspondence between the two notations.

In order to represent a coreference link in the display form, the display notation requires that we place each end of the line of identity within the correct field. Thus we could draw:



but the following would be incorrect:



We see that there may be a paradox with this. This arises with the concept:

[*x : *x]

or the pair of concepts:

[*x : *y] [*y : *x]

where some individual is a member of its type or of a type which is a member of its own type! In order to prevent these meaningless cases we must postulate that such concepts are not well formed. In order to achieve this we postulate

that there exists an *order hierarchy* such that for any referent *r* of any order its type label must be a label occurring in a table of objects whose order is one greater than *r*. Each table of objects is a complete type hierarchy. The notion of an order hierarchy is not taken any further in this work.

Sowa's Current View Of Higher Order Logics

The standard theory does not make proper provision for logics of a higher order than first order, since Sowa did not originally allow each field to contain the same kinds of object, as we do. However, recent writings by Sowa [Sowa93] have introduced a new operator. This is the τ operator. The τ operator is placed before a variable name (without the *) within the type field and shows that the following label is the name of a generic referent in some other concept. With this device Sowa is able to formulate a graph such as:

[SPECIES : *x] [τx : Clyde]

We see that this idea results in the type label τx of the right hand concept being 'instantiated' when the *x in the left hand concept is instantiated. But we also see that τx must also be instantiated when [τx : Clyde] is projected onto:

[ELEPHANT : Clyde]

When this requirement is met we see that the τ operator behaves exactly like the * operator in the referent field and that the two are actually identical in semantics. We would claim that our representation (which we first published in 1991 [Heaton91] [Heaton92]) is more general and intuitive than the Sowa method.

2.4.4 Formal Definitions Relating To The Concept Node

We complete the discussion of the concept node with formal definitions which will provide the definitions of the concept and its contents that we will adopt for the remainder of the work:

Definition 2.2: Concept

A *concept c* is an object which represents an *individual.* The function *referent(c)* maps *c* to an element of a set *I* of *individual markers* or to the *generic marker* *. An individual marker may be either a *token* or a set of conceptual graphs. Where the referent of *c* is not a set of conceptual graphs the concept *c* is called a *simple concept* and the referent is called a *simple referent.* Where the referent of *c* is a set of conceptual graphs the concept *c* is called a *modal concept* and the referent is called a *modal referent.* The function *type(c)* maps *c* to an element of a set *T* of *type labels* or to the *generic marker* *. Concepts are written as boxes with two fields:

[*type_label* : *referent*]

where the : is a *conformity relation* between the type label and the referent.

Definition 2.3: Line Of Identity

A *line of identity* is a connected, undirected graph *g* whose nodes are generic markers and whose arcs are pairs of labelled generic markers called *coreference links.* A line of identity may be drawn between any two fields in any conceptual graph subject to the requirements of the order hierarchy.

This last definition is sufficiently general to apply also to the results of the discussion of the relation node, which follows.

2.5 THE RELATION NODE

The relation node specifies the nature of the relational arc between two or more concept nodes and also of monadic relations.

2.5.1 Relation Label

As in the case of the type label, the traditional relation node can contain as label a simple text string or a lambda abstraction. Section 2.10.2 gives a discussion on the nature of lambda abstractions in the relational node.

Higher Order Logics And The Relational Field

As with the type field, Sowa did not originally specify any means of placing a variable within the relational field. However, in his most recent discourses [Sowa93] he has defined the ρ operator which is placed before a variable name within the relational field and shows that the following label is also the name of a generic referent in some concept. This is exactly analogous to the τ operator used within the type field and again we have previously introduced the notion in [Heaton91] [Heaton92].

With reference to our definition of line of identity we are now able, by the use of coreference links, to represent second order, or meta, statements such as 'if x has a <reflexive_relation> which is y then y has a <reflexive_relation> which is x'. This kind of statement is impossible in Sowa's original Conceptual Graphs notation. The use of coreference links allows higher order logic to be represented with relational labels and is equivalent to the use of coreference links in the type field.

We are now able to encode the above rule:

```
IF      [ REFLEXIVE_RELATION : *rel ]  &
        [ *type_x : *ref_x ]->( *rel )->[ *type_y : *ref_y ]
THEN
        [ *type_y : *ref_y ]->( *rel )->[ *type_x : *ref_x ]
```

The rule is encoded in this IF ... THEN ... form since we have not yet developed our representation for arbitrary formulae. With this kind of rule not only is it possible to talk about relational labels in general but it is also possible to apply the same rule to any reflexive relation whatever. The only additional information that is required about each reflexive relation is a fact such as:

```
[ REFLEXIVE_RELATION : REL ]
```

where REL is the relational label. This kind of compact generalisation about relational labels (and type labels) is impossible in Sowa's original conception.

2.5.2 Direction Of Relations

All relations are directed, a fact which often contradicts the assumptions of some other theories of relations such as Entity-Relation theory [Chen76]. However, it is important to distinguish which of the entities taking part in the relation are playing the role described by the relation and which are not. In all cases the arrows point to the entity which is playing the role indicated by the relation label. Whilst this is in many ways a minor point we are simply clarifying that in all our examples the arrows that point from a relational node to a concept can be taken as pointing towards a role playing entity.

2.5.3 Formal Definition Of Relation

The following formal definition of a relation is that which we adopt for the remainder of the work:

Definition 2.4: Relation

A *relation R* is an object which consists of a relational node, containing a relational label, and two sets of concepts attached to the relational node by directed arcs, one set with arcs directed towards the relational node and the other set with arcs directed away from the relational node. Either set of concepts may be empty. The function *relation(R)* maps *R* to an element of a set of *relational labels* or to the *generic marker* *. Relations are written as:

{ [ ]->... }(*relational_label*) {->[ ]...}

2.5.4 Canonical Graphs

Syntactically, each relation within a conceptual graph looks the same. It is nevertheless the case that the meaningfulness of a relational label depends upon its being placed between concepts of types appropriate to it. Since any relational label may label a relational node between concepts at different levels of generality there may be examples of relational labels between apparently unrelated concepts. However, we assume the existence of one graph *g*

consisting of a single relation *R* with label *r* which is sufficiently general to be a generalisation of all other graphs in which *R* occurs and which is not sufficiently general that it is also a generalisation of some other graph which is semantically unacceptable with respect to *r*. We call the graph *g* the *canonical graph of r*. Sowa provides the *canonical formation rules* by which graphs may be derived from canonical graphs such that all derived graphs are also canonical. We note that any such derived graph will be canonical but it may not be true. In addition we may say that all uncanonical graphs are false since they certainly are not true.

Assumption 2.5: Canonical Graph

A *canonical graph* of a relational label *r* is a graph *g* which consists of a single relation with label *r* and which is a generalisation of all semantically valid cases of *r* but which does not subsume any semantically invalid cases of *r*.

2.5.5 Reasoning With Selectional Constraints

For reasoning with selectional constraints we must postulate that there exists within our system a catalogue of simple graphs. Section 2.5.4 introduced the idea of the existence of canonical graphs but did not state how they might be handled. The processing of these graphs will be dealt with in detail in Chapter 4. For a completely general system the following points arise:

* There exists a catalogue of canonical graphs called the *canonical basis* or the *canonical model.*

* These graphs are the most general cases in which the relations within them are semantically valid under all specialisations.

* These graphs consist of a single relation.

Assumption 2.6: Canonical Basis

A *canonical basis* is a catalogue of canonical graphs, one for each relational label *r* which is known.

2.5.6 The Nature Of Canonical Graphs

The canonical basis provides a general model for the construction and validation with respect to semantic validity of new graphs. As such we require that each canonical graph can be specialised in whatever way we choose and the result always to be valid. The following examples will reveal an apparent subtlety.

Example 1 - The Canonical Graph For The CHRC Relation

For the purposes of this example we assume that any entity can be characterised by its CHRC (characteristic) relation with some attribute:

[ENTITY]->(CHRC)->[ATTRIBUTE]

From this we see that the following graphs are canonical:

[BLOCK]->(CHRC)->[COLOUR : red]

[WAR]->(CHRC)->[VIOLENT]

Example 2 - The Type Label COLOUR

Colour (in the sense of light emanating from a physical surface or material, rather than orchestral colour or coloration of opinion *etc*) can be a characteristic of physical objects. As such it only makes sense for any concept of colour to be connected by the CHRC relation to a concept of type PHYS_OBJ or a subtype of this. Without this restriction we would be able to say from example 1 that the graph:

[WAR]->(CHRC)->[COLOUR : red]

was canonical even though it is not sensible to think of wars as having colour. We must therefore restrict the concept types to which one end of the CHRC

relation can be attached, depending upon to what the other end is attached. Thus we apparently require within the canonical basis the additional graph:

[PHYS_OBJ]->(CHRC)->[COLOUR]

This graph now specifies that colours can only be characteristics of physical objects. Unfortunately it also states that the CHRC relation is only valid between colours and physical objects. There is a problem in that we must choose which canonical graph is the correct one for our purposes. This seems rather arbitrary.

One way round this problem is to have two catalogues: one for relational labels and one for conceptual labels. The conceptual catalogue would need to contain, for each known type label, a complete set of graphs depicting the contexts in which they can be related to other types. Such a catalogue would grow to an enormous size for any realistic system with many type and relational labels.

The other way round the problem is to attempt to identify if there is any difference in what it means to be a characteristic colour or any other kind of characteristic. This leads us to suggest that whilst colour is certainly a characteristic of physical objects it is a particular *kind* of characteristic and it is important to be as precise as possible. In other words, we need to find a subrelation of the CHRC relation if we wish to link a PHYS_OBJ to a COLOUR. Thus the graph:

[BLOCK]->(CHRC)->[COLOUR : red]

should be expressed as something of the form:

[BLOCK]->(PHYS_CHRC)->[COLOUR : red]

2.6 TYPE AND RELATIONAL HIERARCHIES

There are three ways in which type lattice relations and relational lattice relations can be represented. The first way is the Sowa way that uses relations of the form:

$t1 < t2$

to state that type *t1* is a subtype of type *t2*. Although Sowa is very brief in his

treatment of the relational lattice we assume that one exists since we can trivially find examples to demonstrate the point (*e.g.* FATHER < PARENT).

The second way derives from an intuition that the *isa* link a normal relation between entity types and is represented in a manner similar to the following:

[TYPE : *t1*]<-(SUBTYPE)<-[TYPE : *t2*]

[REL : *r1*]<-(SUBREL)<-[REL : *r2*]

which are metalevel, second order representations of the Sowa form. There is a problem associated with this form since for a graph such as this to actually *make t1* into a subtype of *t2* the SUBTYPE relation must be recognised as a predefined relation within the theory. If it is not then the information carried by it will not be used for the purpose within the theory for which the lattices exist, that of allowing a reference to a type or relation to also be a reference to all its supertypes or superrelations. Thus we must postulate that whilst the graph form is a valid form and expresses the relationship between types and relations we also require a more computable form for efficient access. We will assume that the Sowa form is that which is directly available for the operations to be defined in the next chapter where computational efficiency is enhanced but that the second form can be regarded as implying the first.

The third way in which lattice relations may be expressed is by subsumption within definitions. Any type or relational definition must relate one label to its superlabels. This point is taken up in Section 2.10.

We now provide formal definitions for the type and relational hierarchies:

Definition 2.7: Type Hierarchy

A *type hierarchy* is a partial ordering of type labels. The symbol $\leq$ designates the ordering. For type labels *s*, *t*, *u* and *v* the following definitions hold:

* If $s \leq t$ then *s* is a *subtype* of *t* and *t* is a *supertype* of *s*. If $s \neq t$ then *s* is a *proper subtype* of *t* and *t* is a *proper supertype* of *s*.

* If $s < t$ and $s \neq t$ and there is no type u such that $s < u < t$ then s is an *immediate subtype* of t, written $s <_{immed} t$, and t is an *immediate supertype* of s, written $t >_{immed} s$.

* If $s \leq t$ and $s \leq u$ then s is a *common subtype* of t and u. If there is no type v such that $s < v$ and v is also a common subtype of t and u then s is the *maximal common subtype* of t and u.

* If $s \leq u$ and $t \leq u$ [1]then u is a *common supertype* of s and t. If there is no type v such that $v < u$ and v is also a common supertype of t and u then u is the *minimal common supertype* of s and t.

Definition 2.8: Relational Hierarchy

A *relational hierarchy* is a partial ordering of relational labels. The symbol $\leq$ designates the ordering. For relational labels s, t and u the following definitions hold:

* If $s \leq t$ then s is a *subrelation* of t and t is a *superrelation* of s. If $s \neq t$ then s is a *proper subrelation* of t and t is a *proper superrelation* of s.

* If $s < t$ and $s \neq t$ and there is no relation u such that $s < u < t$ then s is an *immediate subrelation* of t, written $s <_{immed} t$, and t is an *immediate superrelation* of s, written $t >_{immed} s$.

* If $s \leq t$ and $s \leq u$ then s is a *common subrelation* of t and u. If there is no relation v such that $s < v$ and v is also a common subrelation of t and u then s is the *maximal common subrelation* of t and u.

* If $s \leq u$ and $t \leq u$ [1]then u is a *common superrelation* of s and t. If there is no relation v such that $v < u$ and v is also a common superrelation of t and u then u is the *minimal common superrelation* of s and t.

Whilst the definition of the type hierarchy is adapted directly from Sowa, the definition of the relational hierarchy and related terms does not occur in Sowa. It is included here for completeness.

In some of the later discussions we will need to refer to linear chains of types and relations with a hierarchy such as *t1* < *t2* < *t3* ... *tn* where there is a complete ordering of the labels. For convenience we will introduce the term *line of inheritance.*

Definition 2.9: Line Of Inheritance

A *line of inheritance* is portion of a type lattice or a relational lattice in which the labels are completely ordered. For any label *l* in a line of inheritance all other labels in the same line of inheritance are related to *l* as either subtype or supertype.

2.7 CONFORMITY RELATIONS

The standard theory contains *conformity relations*, which are statements that some individual marker *conforms* to a particular type label. Thus it is possible to state the relation:

ELEPHANT :: Clyde

which states that the marker Clyde conforms to the type label ELEPHANT. What is also clear is that for Clyde to be an ELEPHANT, Clyde must be known to exist. If this is the case then the concept [ELEPHANT : Clyde] will be present. Although Sowa argues that there is a subtle difference between individual concepts and conformity relations, we consider that this information is duplication of that contained in concepts. Any concept which is part of a true graph is also a statement that some individual exists (by definition) and conforms to a certain type, given by the type label.

In spite of the duplication of information we will adopt a conformity relations table for computational efficiency reasons where it is quicker to find

this information from a table than from the main knowledge base. In particular, we ensure that the entry for any individual marker within the conformity relations table links that marker with the most specialised type label to which it conforms. When this is done the specialisation of concepts which contain this individual marker to their most specific type can be carried out efficiently.

So far we have considered the conformity of individual markers to type labels but we previously argued that generic markers possessed identifying labels in the form of coreference links, even where the line of identity did not extend outside the concept. Hence we must also include conformity relations for generic markers. Thus the concept:

[ELEPHANT]

which is actually a shorthand for the concept:

[ELEPHANT : *x]

states that some elephant exists and hence that the label *x conforms to the type label ELEPHANT. As such there ought to be an appropriate entry in the conformity relations table for the line of identity indicated by *x. We also note that, should *x be reused at a system interface in some other concept in some other context such as:

[CAT : *x]

where this context is completely independent of any other context then this is a *different* *x to any *x that may have been used previously. This difference results from the fact that labels such as *x are intended for human use and that, unless specifically intended, their reuse does not imply a reference to their previous use.

At a later stage we will need to be able to refer to the most specific type label to which a referent conforms. In order to do this we will introduce at this point the notion of an individual's *proper type label.*

Definition 2.10: Proper Type Label

The *proper type label t* of a marker *i* is the type label of the single, most specific type, or *proper type*, to which *i* conforms. If *i* conforms to types $t_1...t_n$ and there is not a known common subtype of all these types then the proper type of *i* is the common subtype t_s of $t_1...t_n$ such that $t_s \equiv t_1 \wedge ... \wedge t_n$, $t_s \neq$ ABSURD.

The function $type_{proper}$ maps a concept *c* to the proper type of *referent(c)*.

2.8 LOGICAL PRIMITIVES OF CONCEPTUAL GRAPHS

The logical primitives of conceptual graphs are only incompletely described in a formal way by Sowa and indeed are sometimes compromised by him and others. This section will clarify what the logical primitives are and will relate these to the concept and relational nodes. The four logical primitives of Existential Graphs, as required by Peirce, and hence conceptual graphs,are:

* conjunction
* negation
* existential quantification
* lines of identity

We have already dealt with existential quantification and so this will not be repeated. The remaining three primitives will be discussed.

2.8.1 Conjunction

There is *always* an implicit logical AND between any two conceptual graphs drawn side by side. Indeed, within any graph with more than one relation there is an implicit AND between each relation, and in any relation with arity greater than 2, all the concepts which are attached to the relational node by arcs of the same kind also form a conjunction.

Sowa and others have occasionally used constructions such as:

[TYPE1 : *x]->(OR)->[TYPE2 : *y]

to specify that (subject to the OR being interpreted inclusively or exclusively) that either the entity *x OR the entity *y exists. However, since all generic concepts are existentially quantified the graph above states that both entities *x *and* *y exist unconditionally. We will also show in Chapter 3 that the existence of both *x and *y is implied in the above graph by Peirce's rule of erasure. It is logically unsound to attempt to show logical connectives other than conjunction by the use of a conjunction without any negation.

Peirce realised that the last point has another source of validity when he changed from entiative graphs, with disjunction as a primitive, to existential graphs, with conjunction as a primitive. He found that most simple English sentences mapped easily to simple graphs which could be read in the same way as the English equivalent when the primitive of conjunction was used instead of disjunction.

2.8.2 Negation

In order to provide the whole of logic, negation is required in addition to conjunction but we have so far avoided negation altogether since we will now argue that there are semantic and logical problems associated with the standard notation. We must therefore start by considering how negation is represented by Sowa and then present arguments which fundamentally undermine Sowa's semantics of negation in terms of graphs nested within concepts as a logical primitive of conceptual graphs.

2.8.2.1 Negation And Logical Context

The referent field can contain sets of graphs. Traditionally, Sowa's set of graphs is negated by appending the monadic NEG relation to the concept which contains them, the justification being that NEG is regarded as a modality. This section shows that Sowa's representation of negation is not what is required, for various reasons, and discusses this under the following headings:

* What is a proposition?

* Logical considerations.

* A solution.

2.8.2.2 What Is A Proposition?

This question will be answered with examples and will at the same time highlight the problem with the standard notation, and therefore semantics, of negation.

Problem 1

The following concept is stated by Sowa [Sowa84] to represent the fact that the nested context { *p* } is a proposition:

[PROPOSITION : { *p* }] (2.8.2.2.1)

This is a satisfactory way of stating that { *p* } is a proposition at the meta level and we have previously argued for its validity under this interpretation. We now consider another example:

[ELEPHANT : Clyde] (2.8.2.2.2)

By direct analogy with graph 2.8.2.2.1 this graph states that Clyde is an elephant. So far the semantics of concepts 2.8.2.2.1 and 2.8.2.2.2 are the same as each other since we can read them both in the same way.

The next step is to use Sowa's notation for negation and negate the concept 2.8.2.2.2 to state that Clyde is not an elephant at all:

(NEG)->[ELEPHANT : Clyde] (2.8.2.2.3)

which states that 'it is false that Clyde is an elephant'. Let us accept the semantics of the NEG relation in this case. Now we return to the case of the PROPOSITION and consider what the following means:

(NEG)->[PROPOSITION : { *p* }] (2.8.2.2.4)

By analogy with the graph 2.8.2.2.3 it ought to be read as 'it is false that { *p* } is

a proposition'. However, the reading that is usually adopted is 'the proposition { *p* } is false'. The accepted reading is fundamentally different to the reading which is analogous to the reading of the case of graph 2.8.2.2.3. The situation worsens when Sowa's abbreviation for:

(NEG)->[PROPOSITION : { *p* }]

is taken into account. In the abbreviated notation this graph becomes:

¬[*p*] (2.8.2.2.5)

This uses *syntactic sugar* to hide the semantic problems that we have just highlighted.

Problem 2

The above discussion showed that Sowa's notation does not actually negate that which it is intended to do. It also hides the problem that for a graph { *p* } to be a proposition then it must be included as the referent of a concept of type PROPOSITION. If this is the case then how is the graph:

[PROPOSITION : { *p* }]

to be treated. Is it a proposition? If not then what is it? Certainly the semantics of conceptual graphs require that this concept is a proposition. If a graph needs to be included within a concept of type PROPOSITION before it can be treated logically as a proposition then the above graph should be included within a concept of type PROPOSITION:

[PROPOSITION : { [PROPOSITION : { *p* }] }]

We see that this is an impossible situation. Every graph must necessarily be the referent of a concept of type PROPOSITION! The real difference between:

{ *p* }

and

[PROPOSITION : { *p* }]

is therefore that { *p* } is a proposition and that [PROPOSITION : { *p* }] is a proposition that states that { *p* } is a proposition. These two statements are

fundamentally different and can coexist without causing any difficulty.

2.8.2.3 Logical Considerations

There are sound logical reasons why the above representation and semantics for negation are incorrect. These relate to the rules of inference of Peirce logic. Without going into these rules in depth at this stage it is worth stating that the rule of erasure allows evenly enclosed contexts to be generalised. By erasure it is possible to remove concepts, relations and whole graphs without altering the truth of a proposition. Therefore the following evenly enclosed graph:

[TYPE_1 : #53246]->(RELATION)->[TYPE_2 : #65234]

can be generalised by removal of portions of the graph to:

[TYPE_1 : #53246]

or:

[TYPE_2 : #65234]

or:

[TYPE_1 : #53246] [TYPE_2 : #65234]

and to any one of several other graphs in which type labels and referents have been generalised. The important principle from the point of view of the arguments being presented here is that a relation can be removed to leave concepts standing on their own.

Because it is logically sound to remove relations from evenly enclosed graphs the following derivation is possible:

Starting from: (NEG)->[PROPOSITION : { *p* }]

Erasure: [PROPOSITION : { *p* }]

This has demonstrated the absurd possibility that the false proposition { *p* } implies the true proposition { *p* }, which is a clear inconsistency. The conclusion is that the NEG relation is not only ambiguous and prone to paradoxes of infinite nesting as shown previously but is also logically inconsistent. Arguments that

attempt to demonstrate the contrary by using the ¬[*p*] kind of negation syntax are not taking into consideration the fact that this notation, whilst it is taken to mean that which is intended, is only an abbreviation for something which does not. Any computer implementation of a conceptual graphs processor which does not address this problem is doomed.

The conclusions of this section of the discussion are that Sowa's notation for negation and the underlying semantics is neither semantically consistent with the 'normal' semantics of the concept nor is logically consistent with the entire logical basis of Conceptual Graphs theory.

2.8.2.4 A Solution

To define the solution to the paradox of infinite nesting of graphs within concepts of type PROPOSITION we must simply accept that the graph:

p

is a proposition and that the graph:

[PROPOSITION : { *p* }]

is also a proposition. The difference is that one graph makes a proposition and that the other makes the proposition that there exists some proposition. The two statements are both legitimate but are quite distinct.

To derive a solution for the correct representation of negation we must go back to Peirce's original notation. In this notation, Peirce performed a simple (although inspired) act: he negated one (or more) of his graphs by enclosing it in an oval (which he called a *cut*). In doing so he was defining some *zone of negation* in which some graphs were placed. However, this zone of negation was not a relation attached to some individual but rather a logical attribute of the enclosed graph. Negation in Peirce's scheme was a primitive of his logic (as it should be), as Sowa requires for his logic. The difference between Peirce's negation and Sowa's is that Peirce's is truly primitive but Sowa's is a compromise that stems from an over-ambitious attempt to map the basic conceptual graph notation (concepts and relations only) to the whole of natural language and first order logic. The use of a relation to show negation hides the problem of how that relation is defined since the NEG relation is not the top of

the relational lattice. However, since there is not a more primitive form of negation in Sowa's representation there is no means to define the NEG relation.

For conceptual graphs it is therefore necessary to invent an additional piece of notation for the representation of negation. We propose to adopt Peirce's notation so that the display form of this negation is simply an oval or a rounded box around the negated graphs and that the linear form represents a negated context by enclosing the graphs in parentheses. Both of these notational changes imply that an additional syntactic device is added to the theory; that of the negated context. However, the justification for its addition is too strong to ignore and the remainder of this work will adopt the principle and the notation that we have suggested.

With this new notation and its semantics we can give the equivalent forms of the previous examples in our notation. Thus the graph:

(NEG)->[PROPOSITION : { *p* }]

becomes:

([PROPOSITION : { *p* }])

This is unambiguous in its meaning that { *p* } is not a PROPOSITION in the same way as:

(NEG)->[ELEPHANT : Clyde]

becomes:

([ELEPHANT : Clyde])

with an equivalent reading, Clyde is not an elephant. In order to state that the graph *p* itself is false we would write:

(*p*).

2.8.2.5 Further Examples

We conclude by giving two examples of the utility of the use of a Peircean cut. The first shows that the result superficially looks like its Sowa equivalent and the second returns to a previous subject.

Example 1

We argued against the use of a graph such as:

[TYPE1 : *x]->(OR)->[TYPE2 : *y]

Assuming the inclusive interpretation of the OR relation we can now replace this with the logically sound construct:

(([TYPE1 : *x]) ([TYPE2 : *y]))

If we required the exclusive interpretation then we would add an extra constraint:

([TYPE1 : *x] [TYPE2 : *y])

This means of providing negation gives a semantically consistent and, as we shall show, a computationally tractable representation.

Example 2

In Section 2.4.1.9 we showed that it was not possible, in Sowa's notation, to encode metasentences such as 'if there exists a graph then that graph is a proposition' in which there is a reference to some general relative clause or modal statement. We introduced the notational machinery to make references to contexts and now with our notation for negation we are now able to formulate the metasentence properly:

({ *x }) ([PROPOSITION : { *x }]))

With this machinery we are able to produce axioms such as axiom K: 'if we know something then we know that we know it':

([KNOWN : { *x }) ([KNOWN : { [KNOWN : { *x }] }]))

This also requires a statement that 'if something is in the knowledge base then we know it':

({ *x }) ([KNOW : { *x }]))

2.8.3 Extending The Definition Of Graph

Having dealt with the representation of negation we are now able to present our definitions in relation to graphs which are not simple graphs:

Definition 2.11: Negated Context

Any set of graphs (which may be empty) which is enclosed within a cut forms a *negated context.* Any negated context may contain nested within it any number of additional negated contexts. For any context *c* which is enclosed within a number of cuts *n* (nested at *depth n*) one of the following is true:

* If *n* is an odd number then *c* is said to be *oddly enclosed.*
* If *n* is either 0 or an even number then *c* is said to be *evenly enclosed.*

All graphs within any context form a conjunction.

Definition 2.12: Positive Context

Any set of graphs which is not enclosed within a cut is a *positive context.* Such a set of graphs forms a conjunction.

Definition 2.13: Compound Conceptual Graph

A *compound graph* is any graph that either consists of a positive context containing more than one graph of any kind or of a negated context.

By these definitions, the following are examples of compound graphs:

* *p q* where *p* and *q* is a conjunction of simple graphs.
* (*p*) where *p* is any graph whatever.

(*p* (*q*)) where *p* and (*q*) form a conjunction nested within the outer cut.

2.9 CONCEPTUAL GRAPHS AND STANDARD LOGIC

Sowa defines a *formula operator* ϕ which is used to map conceptual graphs to formulae of the first order predicate logic (FOPL). Thus for a conceptual graph *u* the equivalent FOPL formula is ϕ*u*. Unfortunately, at various points in his book, he does not provide unique mappings between conceptual graphs and FOPL and his first set of properties of ϕ is also the first example where the mapping between conceptual graphs and FOPL contains a pitfall. Indeed, Some parts of Sowa's notation are not mapped to FOPL at all. These are covered in this section. Additional aspects will not be discussed but will be included in the set of definitions that we will provide to cover the whole of our notation.

2.9.1 Simple Graphs

2.9.1.1 Simple Concepts

Conceptual Graphs are a notational variant of sorted logic. This is so because of the presence of the type label within the concept, the relational label and the existence of hierarchies of both of these labels. Whilst unsorted FOPL can certainly perform any computation that a sorted version can, the application of a sort label to markers reduces search spaces and makes computations with sorted logics more efficient.

Sowa develops a mapping of Conceptual Graphs notation to FOPL. Whilst it is possible to produce such a mapping it is not possible to make that mapping 1:1 since there is an ambiguity in FOPL that is not addressed by Sowa's version of ϕ at this point. This ambiguity can be shown by example. The following concept illustrates the idea:

[TYPE : Ref] (2.9.1.1.1)

When mapped to unsorted FOPL by ϕ as defined by Sowa this gives:

TYPE(Ref) (or ∃Ref TYPE(Ref) if Ref is a variable) (2.9.1.1.2)

However, in a more standard notation for a sorted logic [Frost86] this would be given as:

TYPE : Ref (2.9.1.1.3)

Similarly, according to Sowa we may map the following graph to unsorted FOPL:

graph: (REL)->[TYPE : Ref] (2.9.1.1.4)

FOPL: REL(TYPE(Ref)) (2.9.1.1.5)

Formula 2.9.1.1.5 has the same general structure as formula 2.9.1.1.2. This implies that there is some ambiguity about whether any predicate symbol is equivalent to a type label or a (monadic in this case) relation. In one sense we may argue that the ambiguity can be resolved by observing that the argument of a predicate which represents a concept is always an atom or a variable whereas the argument of a monadic relation is never an atom or a variable. Nevertheless, this seems to be an arbitrary distinction and we would prefer a tighter distinction between the two. In order to achieve this we redefine the ϕ operator with respect to the mapping between concepts and sorted FOPL in such a way that conceptual types become sort labels and conceptual relations become predicate symbols.

2.9.1.2 Modal Concepts

The following two derivations from modal concepts to unsorted FOPL show a further problem:

Graph 1: [Q : R]
Step 1/1: Q(R) by Sowa's definition of ϕ

Graph 2: [PROPOSITION : { [Q : R] }]
Step 2/1: (Q(R)) by Sowa's definition of ϕ
Step 2/2: Q(R)

The disappearance of the braces { } and type label PROPOSITION in step 2/2 needs clarifying. Sowa's mapping of a set of nested graphs to FOPL is:

[PROPOSITION : { *p* }] => (ϕp)

In other words, since both graph 1 and graph 2 both map to the same thing, it is

not satisfactory to map concepts to monadic predicates in unsorted FOPL since information may be lost. Instead they must also be mapped to the sorted logic form given in formula 2.9.1.1.3. This mapping does not affect the cases where an existential quantifier is required since a generic concept of type *t* maps to:

$\exists(t{:}x)$

A modal concept represents a set of graphs which together form an entire context. In this context it would be strictly wrong to say that we were mapping graphs into FOPL since we are generating second order formulae, so we will instead coin the term sorted GML (general modal logic). The mapping of such a set of graphs into sorted GML must follow the general form of that of simple concepts. Thus a modal concept such as:

[PROPOSITION : { *p*, *q*, ... }]

maps to a sorted GML formula of the form:

PROPOSITION : (ϕp, ϕq, ...)

With this mapping the graph:

[PERSON : John]->(BELIEVE)->[PROPOSITION : { [UNICORN] }]

is equivalent to the sorted GML formula:

BELIEVE(PERSON : John, PROPOSITION : ($\exists$(UNICORN : x)))

The braces representing the nested context boundary of the modal concept are mapped to parentheses in the sorted GML. In unsorted GML the mapping gives:

BELIEVE(PERSON(John), PROPOSITION($\exists$(x) UNICORN(x)))

2.9.1.3 Identity

A referent such as:

*x = *y

maps to the following equality in sorted FOPL:

x = y

2.9.2 Negated Contexts

Sowa does not provide an adequate mapping for negation between Conceptual Graphs and FOPL or sorted logic since Sowa's notation for negation is, as we have shown, itself unsatisfactory. However, we are now in a position to map the cut into FOPL.

The cut of Peirce logic is exactly equivalent to the negation symbol $\neg$ of standard logic. Thus the mapping from conceptual graphs, with the cut notation that we have proposed, into FOPL results in exactly the form of FOPL formula that Sowa's mapping of the NEG relation gives. Nevertheless, the mechanism of the mapping of the cut is different to that of the NEG relation.

The following is an example of the mapping:

Graph:	(*p*)	'*p* is false'
FOPL:	$\neg$(*p*)	'*p* is false'

2.9.3 The Formula Operator ϕ

We now define the formula operator ϕ with respect to our rationalised notation. Since we wish to map to a modal logic we will coin the term *sorted modal predicate logic* for this definition.

Definition 2.14: The Formula Operator ϕ

The formula operator ϕ maps conceptual graphs into formulae of sorted modal predicate logic. If *u* is any conceptual graph then ϕu is a formula determined by the following construction:

* If *u* is a negated context then represent the cut with the negation symbol $\neg$ followed by an opening parenthesis and a closing parenthesis. Map each graph *g* nested within *u* and place ϕg within the parentheses, separating each with the $\wedge$ symbol if there is more than one.

* If *u* contains *k* generic markers then assign a distinct variable symbol $x_1,x_2,...,x_k$ to each one, assigning the same variable symbol to each generic marker on a line of identity.

* For each concept *c* in *u*, let *identifier(c)* be the variable assigned to *c* if *c* is generic, or *referent(c)* if *c* is a simple individual. If *referent(c)* is a modal referent then represent *referent(c)* as an opening parenthesis and a closing parenthesis. Map each graph *g* nested within *referent(c)* and place ϕg within the parentheses, separating each with the $\wedge$ symbol if there is more than one.

* Represent each concept *c* as a sorted variable or individual whose sort label is *type(c)* and whose argument is *identifier(c).*

* Represent each *n*-adic relation *r* of *u* as an *n*-adic predicate whose functor is *type(r)* and whose arguments are the identifiers of the concepts attached to *r.*

2.10 ABSTRACTION AND DEFINITION

2.10.1 Type And Relational Definitions

Conceptual Graphs theory allows the introduction of a type hierarchy and a relational hierarchy and as a consequence of this requires that types and relations can be defined in terms of their more general supertypes and superrelations. Sowa provides a method of definition by lambda abstraction. We will describe an equivalent means of definition in terms of logical equivalence. It will be shown in the next chapter that the use of logical equivalence removes some redundancy from the possible operations on conceptual graphs and thereby reduces computational complexity. For the present it is only necessary to discuss the correspondences between the lambda abstraction form and the logical form of definition.

2.10.1.1 Type Definitions

A typical Sowa type definition has the following general form:

type TYPE(x) **is**

graph(*x)

in which TYPE is the type label being defined and *graph*(*x) is a single simple graph giving the *differentiae* of the definition. The (x) in the heading is a coreference link between TYPE and a concept within *graph*(*x). In addition the following equivalent form is often used:

TYPE = λ(x) *graph*(*x)

where *graph*(*x) is a shorthand notation which represents a graph that contains a concept containing the coreference marker x. We now analyse the semantics of this syntax.

A lambda abstraction represents a universally quantified statement that all entities of type TYPE are defined by *graph*(*x) which contains a concept of type *t* such that TYPE $<_{immed}$ *t*. Now, this can be stated as an implication:

([TYPE : *x] (*graph*(*x)))

In this implication the oddly enclosed concept, which maps to the header of a lambda abstraction, implies *graph*(*x). In other words all entities of type TYPE are also entities of type *t* within *graph*(*x) and they have properties additional to those of entities of type *t*. As this implication is intended to express a definition, *i.e.* an equivalence between the single concept and its defining graph, then the graph *graph*(*x) also implies the single concept [TYPE : *x]:

(*graph*(*x) ([TYPE : *x]))

A complete type definition is therefore a logical equivalence: a single concept *c* which implies some arbitrary graph *g* and the graph *g* which implies the concept *c*.

In this form there is not any need to treat type definitions in any way differently to arbitrary domain rules. This simplifies proof techniques without

removing anything from the representational power of the formalism.

Definition 2.15: Type Definition

A *type definition* is a special case of logical equivalence which has the general form:

([TYPE_1 : *x] ([TYPE_2 : *x]...))
([TYPE_2 : *x]... ([TYPE_1 : *x]))

where TYPE_1 $<_{immed}$ TYPE_2 and '...' represents at least one relation.

2.10.1.2 Relational Definitions

A relational definition may be expressed by lambda abstraction in a similar way to a type definition - as an *n*-adic abstraction. Thus we may have:

relation REL(x_1,...,x_n) **is**
graph(*x_1,...,*x_n)

or in the equivalent notation:

REL = $\lambda(x_1,...,x_n)$ *graph*(*x_1,...,*x_n)

Further Evidence

In Section 2.9 we discussed the mapping of conceptual graphs to FOPL. We can now demonstrate that there is a potential problem with Sowa's mapping of type and relational labels to lambda abstractions, and that this impinges upon and resembles the ambiguity discussed in Section 2.9.1.1. Consider the concept:

[TYPE : *]

and the mapping of its type label to its definition:

TYPE = $\lambda(x)$ *graph*(*x)

Next consider the monadic relation:

(REL)->[TYPE : *]

and the mapping of its label to its definition:

REL = λ(x) *graph*(*x)

This has exactly the same form as a type definition and there is not any way to distinguish between them. This begs the question whether concepts perhaps should have type labels at all and should just contain a referent, the type information being held in a monadic relation. A system known as 'knowledge Graphs' takes a similar view [Bakker87] [James92]. We will not take this question any further in the present work.

In order to deal with the ambiguity problems, we note that the situation with relational definitions is similar to that with type definitions; any relational definition can expressed as a logical equivalence. However, in the case of relational definitions there are two forms that such a definition can take. The first form provides a general relational definition and the second form additionally specifies a subrelation relationship between two relations. Each form will be illustrated:

General Case

The general form of a relational definition can be given as:

(*graph1*(*x_1,...,*x_n) (*graph2*(*x_1,...,*x_n)))
(*graph2*(*x_1,...,*x_n) (*graph1*(*x_1,...,*x_n)))

where *graph1*(*x_1,...,*x_n) is a graph consisting of a single relation, that being defined, with one concept for each referent *x_1,...,*x_n, and *graph2*(*x_1,...,*x_n) is a larger graph such that each concept in *graph1*(*x_1,...,*x_n) is coreferent with some concept in *graph2*(*x_1,...,*x_n). This definition follows since for any relational label *r* there is a graph *h* which maps to the header of the defining lambda abstraction for *r* and a graph *g* which defines *r*. The graph *g* makes explicit the set of attributes that define *r* and is thus a physically bigger graph which must contain all the concepts in *h*. The following example is that of a general relational definition:

```
( [ PERSON : *x ]->( GRANDPARENT )->[ PERSON : *y ]
        ( [ PERSON : *x ]->( PARENT )->[ PERSON ]-
                ( PARENT )->[ PERSON : *y ] ) )

( [ PERSON : *x ]->( PARENT )->[ PERSON ]-
                ( PARENT )->[ PERSON : *y ]
        ( [ PERSON : *x ]->( GRANDPARENT )->[ PERSON : *y ] ) )
```

This example defines a grandparent as a parent of a parent.

Special Case

The case which defines a subrelation relationship between two relational labels is a special case of the general case. In this case the defining graph, *graph2*(*x_1,...,*x_n), also contains a single relation in which each concept is coreferent with a concept in *graph1*(*x_1,...,*x_n). Now, at least one of the concepts attached to *graph2*(*x_1,...,*x_n) must either be more specialised than the corresponding concept in *graph1*(*x_1,...,*x_n) or it must have additional relations attached to it. To exemplify the special case the following will be offered:

```
( [ PERSON : *x ]->( FATHER )->[ PERSON : *y ]
        ( [ PERSON : *x ]->( PARENT )->[ MAN : *y ] ) )

( [ PERSON : *x ]->( PARENT )->[ MAN : *y ]
        ( [ PERSON : *x ]->( FATHER )->[ PERSON : *y ] ) )
```

This example defines a father as a male parent. Because the FATHER relation exists between concepts which correspond to those in the PARENT relation and those in the PARENT relation are more specific, then the FATHER relation is a subrelation of the PARENT relation. The above example may seem somewhat contrived since the [PERSON : *y] in each implication perhaps ought to be given as [MAN : *y]. If this were the case then the definition would become:

```
( [ PERSON : *x ]->( FATHER )->[ MAN : *y ]
        ( [ PERSON : *x ]->( PARENT )->[ MAN : *y ] ) )

( [ PERSON : *x ]->( PARENT )->[ MAN : *y ]
        ( [ PERSON : *x ]->( FATHER )->[ MAN : *y ] ) )
```

But in this case information is lost since there is no means of determining whether FATHER < PARENT or PARENT < FATHER. Indeed, this definition shows that FATHER = PARENT.

Definition 2.16: Relational Definition

A *relational definition* is a special case of logical equivalence which has the general form:

([TYPE_1 : *x]->(REL)->[TYPE_2 : *y]
([TYPE_1 : *x]->{ R_1 ... R_n }->[TYPE_2 : *y])
([TYPE_1 : *x]->{ R_1 ... R_n }->[TYPE_2 : *y]
([TYPE_1 : *x]->(REL)->[TYPE_2 : *y])

where REL is defined by the set of relations signified by { R_1 ... R_n }. In the case where { R_1 ... R_n } is a single relation *r* and either: one or more of the concepts *c* attached to this relation is more specialised than the coreferent concept attached to REL or there are additional relations attached to *c* then REL is a subrelation of *r*.

2.10.2 Lambda Abstractions Within The Type And Relation Fields

Sowa's syntax allows the inclusion of lambda abstractions within the type field of a concept and within the relational field of a relation node. The form of each is similar and follows normal syntax for lambda abstractions. However, Sowa does not go on to show how these lambda abstractions are processed and does not provide Conceptual Graphs equivalents of the lambda calculus operations for them.

We will now show that the inclusion of lambda abstractions in either the type or relational fields does not add any expressive power to conceptual graphs. We consider the following trivial type definition in Sowa's notation:

type ELEPHANT(x) **is**

[MAMMAL : *x]->(PART)->[TRUNK]

With this in mind we can write the following graph:

[λ(x) [MAMMAL : *x]->(PART)->[TRUNK] : Clyde]

This tells us that there exists a mammal with a trunk which is referred to as Clyde. It also tells us, by way of the parameter of the lambda abstraction, that Clyde is a mammal rather than a trunk. We next consider the graph:

[MAMMAL : Clyde]->(PART)->[TRUNK]

This graph states that there exists a mammal referred to as Clyde and that this mammal possesses a trunk. The information within each graph is identical, thus we do not regard lambda abstractions within the type or relational fields as adding any expressiveness. Not only that, should a knowledge base contain the nested form but a query of whether Clyde possesses a trunk be made in the 'flat' form then a proof by projection will fail. It will be necessary to implement a rule of inference that creates nested forms from flat forms, or *vice versa*, and then reproves the query. This is easy in this simple case and results in the proof only having to be attempted three times in total (once for the original flat query, once for the query with the lambda abstraction nested in the MAMMAL concept and once for the lambda abstraction nested within the TRUNK concept). For any query which is larger than a single relation the number of possible variants is very large.

2.10.3 Composite Individuals

Sowa argues for the existence of *individual definitions* based upon instantiations of (some of) the generic referents within the type definition of the type to which the individual conforms. In this way it is possible to record and reason with properties that differentiate one individual from another. The format of an individual definition is:

individual TYPE(individual_marker) **is**

graph

where *graph* contains a concept of type TYPE (presumably the individual's proper type) which contains individual_marker as referent.

The information in an individual definition may be greater than that which simply matches the topology of the type definition of TYPE. Thus *graph* may be of any size greater than that of the *differentiae* of the definition of TYPE. From this it follows that the individual definition of any particular individual could encompass the whole of the information connected with that individual in a knowledge base. Such a definition would include information about other individuals. Taken to its extreme, for any knowledge base there would be one individual definition for each individual marker but *graph* in each case could be identical. If *graph* were deliberately kept small (equivalent to an instantiation of the type definition of the type label for the individual), there would not be any guarantee that the definitions of each marker of any one type would distinguish between individuals except by their possession of different markers. In either case the individual definition is not a definition at all.

The status of the individual definition of Sowa's theory is therefore unclear. We argue that all information about individuals is merely a part of the general knowledge base. As such the information within 'individual definitions' is as immediately available as any other information. This approach represents another simplification of the formalism and computational complexity without removing any expressive power.

2.10.4 Schemata And Prototypes

Sowa provided two further forms of abstraction, *schemata* and *prototypes*. The format of each is:

schema for TYPE(x) **is**

graph

and

prototype for TYPE(x) **is**

graph

In each case *graph* contains a concept of type TYPE which contains the referent *x.

The philosophy of definition, by *definiendum* and *definiens*, discussed so far is that of Aristotle [Aristotle]. In this conception, each type is defined in terms of *necessary and sufficient* conditions. Wittgenstein [Wittgenstein21] [Wittgenstein53] proposed a differing view in which types are defined by *family resemblance* in which it is not necessarily the case that one can pinpoint the exact nature of the type. We suggest that Wittgensteinian definitions can be reduced to Aristotelian definitions at some level. To use one of Wittgenstein's own examples, that of the definition of *game*, we argue against Wittgenstein's view that there are completely different forms of game and that some have so little in common with others that it is impossible to define a common concept of *game* by *definiendum* and *definiens*. It may simply be the case that language has evolved to use what is now the same word for completely unrelated concepts. If this is true then the different uses of a word must be taken into account when attempting to define the concepts expressed by them. Thus, when the construction of a definition for a word which applies to a genuine set of similar and related entities is attempted we will find that at the centre of any conceptual definition is a core of the Aristotelian form. For those concepts such as *game*, for which it is difficult to find such a core, we argue that those searching for the core fail to find it because it is not there and the only similarity between different concepts of *game* is in the name that humans have arbitrarily assigned to fundamentally different concepts.

As a consequence of the foregoing argument we conclude that Sowa's schemata and prototypes are of uncertain status within the theory. Schemata have uncertain status since, if they are to be understood as definitions, they appear, by virtue of the fact that the header contains the same type label as the genus, to define types in terms of themselves. This leads to circular definitions. Prototypes have uncertain status since they do not contain any information about exactly how typical is the prototype and Sowa does not provide any logic with which to operate on them. For the purposes of the present work we will not consider schemata and prototypes further.

2.10.5 Actors

General Considerations

Actors and the associated control marks appear to be included in Sowa's theory in order to provide certain features that are present in certain programming languages, that is, escapes to external routines which can be used to efficiently perform certain routine computation. The actor is represented as a diamond shaped node and has some concepts as inputs and some as outputs. In this way it is possible to specify actors for computations such as addition. However, it would appear that our purpose of producing a completely declarative knowledge processing environment is in contradiction with the use of such devices and we therefore do not include them explicitly within our logic.

Nevertheless we also accept that in the future we may extend our theory to allow a form of actor in real systems. The reason for this is that practical applications which are computation intensive will require efficient bulk processing capabilities which can be performed more efficiently with dedicated code. Without such code a system would have to be taught mathematics from first principles. For this reason we may wish to provide a mechanism for accessing such code and which fits our requirement for the maintenance of the declarative nature of our system.

There is an alternative view of actors that may be taken. This view is one in which the actor actually specifies a relationship between its 'inputs' and 'outputs'. Thus we may represent an actor for addition as the relation:

relation SUM (x,y,z) **is**

(SUM)
<-[NUMBER : *x]
<-[NUMBER : *y]
->[NUMBER : *z]

Since we require the relation to generate the correct sum when certain arguments are presented we must implement a suitable algorithm to trap relations of this kind and perform the calculation, filling in the referents. Such a relation must then evaluate to true. Any such relation which already has the

numbers filled in but those numbers do not satisfy the relation must evaluate to false.

Numbers

Numerical calculation is an area which is important in many aspects of problem solving. Many recreational puzzle books contain 'logic' puzzles which ostensibly do not require any specialist knowledge but which almost invariably require numeracy (Appendix I). Numbers appear in most aspects of everyday reasoning. Thus, for the solution of most kinds of problem, a working knowledge of arithmetic is essential. With this in mind we must provide a semantically acceptable representation for numbers.

There are two ways in which to handle numbers. The most theoretically satisfactory way is to axiomatize numbers and arithmetic and teach the axioms to a conceptual graphs processor. The second way is to use some form of active relation as just described. In the first case the processor will possess a full understanding of numbers and arithmetic but will also acquire a set of rules which must be processed each time some arithmetic need be carried out. In the second case the processor will not understand numbers or arithmetic but rapid calculations will be possible.

We propose to make the second approach available by providing the *number* referent and the type label NUMBER. The number referent will derive its syntax from Sowa's *measure referent* since this was intended for similar purposes. Thus the concept:

[NUMBER : @3]

represents the number 3. The @ symbol states that the label 3 stands for the numerical value 3. Subtypes of NUMBER may be defined and these may be used to disambiguate the roles of numbers in situations where this matters, such as division or subtraction. Thus we may then use a relation such as:

(QUOTIENT)
->[NUMBER]
<-[NUMERATOR]
<-[DENOMINATOR]

Chapter 4 describes the relations of this kind which will be built into the system and show how the set can be extended.

2.11 OTHER FORMS OF REASONING

The form of reasoning that Conceptual Graphs theory supports is exact reasoning where the world is black and white, true and false. There are additional refinements in the form of inheritance but we have shown that this is only a special case of the more general logic. However, the trends in modern automated reasoning are towards logics which provide inexact reasoning based upon some form of fuzzy reasoning. These are manifested in fuzzy logic [Zadeh74] and neural nets in particular which are both becoming more common and popular in their application and the claims made of them. Whether or not these claims are fully justified it is evident that such forms of reasoning are very effective for the applications where they are used and that they possess some psychological validity. Neural nets can be trained to recognise patterns with the same degree of precision and objectivity that humans display.

Conceptual Graphs theory does not contain anything that would permit fuzzy reasoning since the logical basis is one of exact reasoning. It is argued by Sowa that fuzzy reasoning can be emulated by defining whole ranges of conceptual types. Whilst this is no doubt true it is arguable that, for practical applications, this approach is unrealistic. Many such ranges would be very large and the number of definitions required would be correspondingly large. Proof procedures involving such large amounts of information would soon run into problems of complexity which were out of proportion to the problem being solved. Neural nets, even simulated neural nets running on ordinary computers, can solve pattern recognition problems in real time even where the problem space is large.

A possible solution to the problem of fuzzy pattern recognition in Conceptual Graphs theory would be the development of fuzzy equivalents of the standard conceptual graphs operations. In this way, a single definition would cover a whole range of similar types and the degree to which a definition matched a pattern would be included in the formalism. Some work has been carried out in this area [Wuwongse93].

2.12 SUMMARY

This section brings together the discussions of the previous sections and lists the primitives of our rational reconstruction.

2.12.1 Summary Of Conceptual Graphs Structures

Our reconstruction consists of the following primitive objects:

Concept Nodes

A concept node has two fields; the type field and the referent field.

The type field can contain:

* labels/markers
* labelled variables

The referent field can contain:

* names/markers
* labelled variables
* numbers
* contexts

Each field may contain more than one object, separated by the = symbol.

Relation Nodes

A relation node can contain:

* labels/markers
* variables

A relation may be of any arity. It may contain more than one object, separated by the = symbol.

Since each of the above fields can contain more than one of the allowed objects, this permits nodes to lie one more than one line of identity and therefore allows equality.

Negation

Negation of a context is represented by enclosing it within a simple enclosure. Enclosures may be nested to any depth.

2.12.2 Model Theoretic Semantics

The model theoretic semantics of our system are those of the open world. We will clarify how specific Conceptual Graphs notions fit into the standard model.

2.12.2.1 Canonical Basis

The canonical basis, or canonical model, is the catalogue of graphs which are deemed canonical and are used by the system to check for selectional constraint violations.

2.12.2.2 The True Set

The true set *T* of conceptual graphs contains all simple graphs. The graphs of the canonical basis are not necessarily true and are therefore not a part of *T*. This is why these form an additional component of our model, the canonical basis *C*.

2.12.2.3 The False Set

The false set *F* of conceptual graphs contains all negated simple graphs and compound negated contexts. In a sense these are also part of *T* because if some conceptual graph *g* is false then it is true that it is false and this falsity is expressed as part of the graph and not by its inclusion in a separate compartment of the knowledge base.

2.12.3 Type Lattice And Relational Lattice

These lattices are additions to the normal components of an open world and exist because the Conceptual Graphs formalism requires them to do.

2.12.4 Conformity Relations Table

The conformity relations table exists within our open world as a structure to provide quick lookup of conformity relations and not for its own sake.

2.13 CONCLUSIONS OF CHAPTER 2

This chapter has surveyed the components of Sowa's theory and has suggested areas where changes are appropriate. In each case the changes have been specified and justified on theoretical grounds. The changes are designed with the forthcoming development of a set of operations and a computational model of an automated reasoning system in mind.

We have shown that the theory of Conceptual Graphs is basically sound and that our changes have improved upon the semantics of several aspects of the notation. Nevertheless, we recognise that there remain areas of expressiveness where further work is required. Such work is mainly related to aspects of modal logic and inexact reasoning.

CHAPTER 3

CONCEPTUAL GRAPHS AND LOGIC

3.1 INTRODUCTION

This chapter provides a review of the operations on conceptual graph structures. Particular attention is paid to identifying any results which can be obtained in more than one way with different parts of the theory, showing such alternative ways. Also, the modifications to the representational aspects of the theory which were made in Chapter 2 will be accounted for in computational terms, since some notational changes require extension and modification of Sowa's operations.

By the end of this chapter, we will have developed a lean set of operations and shown that some operations can be carried out by other more primitive operations and are nothing more than special cases. As a result, a set of operations that retains the computational completeness of the theory in a manner that will allow more general and efficient computation will have been identified. The direction which the discussion will take will be towards the development of the operations required for the computational theory to be developed in Chapter 4.

We begin this chapter by considering the graph combination rules, the canonical formation rules, and then review the operations which follow from these, with particular regard to their logical status. Following this we discuss the logical basis of Conceptual Graphs, Sowa's interpretation of Peirce's Existential Graphs. This provides us with the opportunity to compare the formation rules with the logical rules and to show that there are areas where the two overlap. Where such overlap occurs we rationalise the rules to remove the duplication. Peirce's rules are themselves subjected to scrutiny and rationalisation.

Having set out the rationalised rules we then go on to show that Sowa's linear form notation can lead to logical inconsistencies and we offer a solution.

We then direct our attention to how the operation of projection can be fully defined in terms of Peirce logic to cover cases which were omitted in Sowa's original account.

3.2 CANONICAL FORMATION RULES

In Sowa's terminology, the basic graph combination operations are:

* Copy.
* Join.
* Restrict.
* Simplify.

There is one further rule that Sowa introduced into earlier versions of his theory and which he later dropped. This rule is the rule of *detachment* [Sowa84]. We will include it in the present discussions since we will wish later to link it to Peirce's rule of *erasure*.

The canonical formation rules, which Sowa regards as refutation rules, are intended to allow the construction of new graphs that maintain the basic property of semantic correctness. Thus the rules do not allow the generation of a graph expressing Chomsky's famous sentence 'colourless green ideas sleep furiously' from any set of graphs which does not contain the semantic violations of this sentence. Alternatively, any graph can be refuted if it can be shown that it cannot be constructed by the canonical formation rules from a canonical basis. On the other hand, the canonical formation rules do not guarantee the truth of every graph generated from a true graph. We review each rule and identify where truth is maintained and where it is not.

3.2.1 Copy

When some graph p is copied the result is the conjunction of both copies of p, $p \wedge p$. Since the idempotency of conjunction states that $p \wedge p \Leftrightarrow p$ we can also claim, since we stated in Chapter 2 that an uncanonical graph must be false, that the conjunction $p \wedge p$ must be true if p is true and that therefore the copy of p must be true and therefore be canonical.

The rule of Copy therefore guarantees that if any graph is canonical then any copy of it is also canonical. Since a copy of any graph *g* contains exactly the same information as *g* then the copy operation is also truth maintaining. It will be used later to allow 'working copies' of graphs to be generated.

3.2.2 Join

This operation, known more strictly as the *join on identical concepts* in the form described by Sowa, is the operation of connecting two graphs together if they share identical concepts. Thus the following two graphs may be joined since they share a common concept:

[*p* : #1]->(*r1*)->[*q* : #2]

[*p* : #1]->(*r2*)->[*r* : #3]

The result of the join is the graph:

[*p* : #1]-
(*r1*)->[*q* : #2]
(*r2*)->[*r* : #3]

We should add that the term 'identical concepts' ought to mean that the two concepts share the same referent since the type labels may vary by generalisation. Thus we can join:

[*p1* : #1]->(*r1*)->[*q* : #2]

[*p2* : #1]->(*r2*)->[*r* : #3]

where *p1* < *p2*. The result of the join is the graph:

[*p1* : #1]-
(*r1*)->[*q* : #2]
(*r2*)->[*r* : #3]

Indeed, if neither *p1* < *p2* nor *p2* < *p1* then, if we know that both graphs are true, we may still join these two graphs and create the type *p3* such that *p3* < *p1* and *p3* < *p2* since we know that there must be a type *p3* which is the proper type of the marker #1 in this case.

The join on identical concepts, in Sowa's use of the term, is not truth maintaining. For instance:

Given: [*p*]->(r1)->[*q*]
[*p*]->(r2)->[*s*]
Join: [*p*]-
(r1)->[*q*]
(r2)->[*s*]

and yet the two given graphs do not imply the joined graph since, although the notation makes the two [*p*] concepts look identical, there is no line of identity between them. As such the join on identical concepts in this form does not have any logical validity since there is no reason why the two concepts [*p*] necessarily refer to the same individual. However, we will later wish to use the join on identical concepts operation in a truth maintaining form for ensuring that the knowledge base contains each individual marker only once in the true set. Thus we must define a truth maintaining form of the join on identical concepts.

Definition 3.1: Join On Coreferent Concepts

Two graphs *g1* and *g2* may be joined if *g1* possess a concept with referent *r* of type *t1* which shares an identical individual marker or is connected by a line of identity to a concept with referent *r* of type *t2* in *g2*. If either type *t1* or *t2* is a more specific type than the other then the concept generated in the join is of the more specific type. If neither type *t1* or *t2* is a subtype of the other then a new type *t* is created such that *t* is the common subtype of *t1* and *t2* and the concept generated in the join is of type *t*. The relation *t* :: *r* is thus created.

This definition of a truth maintaining join on identical concepts forbids the previous join on apparently identical generic concepts but allows the following join:

Given: [*p* : **x*]->(r1)->[*q*]
[*p* : **x*]->(r2)->[*s*]

Join: [*p* : **x*]-
(r1)->[*q*]
(r2)->[*s*]

Later on, the maximal join operation will be discussed. This is also not a truth maintaining join but it will be shown that it has its place within a theory generating system. The definition of this form of join will be extended to incorporate the non-truth-maintaining join that we have just excluded from the join on coreferent concepts operation.

Having redefined the join on identical concepts operation we see that it has taken on the appearance of Peirce's coreferent join operation (discussed later). Indeed, the other canonical formation rules will be shown to be special cases of Peirce's rules. This point is demonstrated further in Section 3.6.

3.2.3 Restrict

Restriction is the replacement of a type label *t1* in some concept *c* with a type label *t2* where it is known that *t2* < *t1* (*t2* is a subtype of *t1*) and *t2* :: *referent(c)* (*referent(c)* conforms to *t2*). Since all semantically correct graphs are specialisations of graphs in the canonical basis then the restriction of *t1* to *t2* must be semantically valid when it is known that it is true that the individual referred to by *referent(c)* participates in the relations attached to *c*. Additionally, since the relation *t2* :: *referent(c)* holds, the restriction must also be truth maintaining. Where a restriction is performed in the absence of *t2* :: *referent(c)* then the operation is not truth maintaining.

We will later use the restrict operation in the presence of conformity relations to maintain each individual concept within the knowledge base in its most specialised form. In this way we will encapsulate as much information as possible within individual concepts and thus reduce the search space required by an algorithm for the determination of inherited properties.

3.2.4 Simplify

Simplification, in its simplest form, is the removal of duplicate relations. Removal of duplicated information neither removes semantic validity nor alters the truth of a graph. This follows from the idempotency of conjunction as

discussed in Chapter 2. Thus the simplify operation is truth maintaining.

The rule of simplify will later be used to ensure that duplicate information is removed from the knowledge base.

3.2.5 Detachment

This rule is given by Sowa as:

> If *u* is a canonical graph and *r* is any conceptual relation of *u*, then let the resulting graph *w* be any connected graph that remains when *r* and its arcs are detached from *u*.

This rule allows concepts and relations to be removed from graphs. Any removal of information must leave a previously canonical graph canonical and any true graph will remain true after the removal, although any false (oddly enclosed) graph may not remain false and so detachment in oddly enclosed contexts is not truth maintaining. The non-maintenance of truth does not present a problem for simple graphs on the sheet of assertion however, since all such graphs are true by definition.

There is a second way in which detachment may operate. This is to allow the splitting of a simple graph which contains several relations into several simple graphs, each with one (or more) relation. Thus we see that the following graph may be split by detachment:

```
[ p : #1 ]-
          ( r1 )->[ q ]
          ( r2 )->[ s ]
```

To give the following pair of graphs:

```
[ p : #1 ]->( r1 )->[ q ]
[ p : #1 ]->( r2 )->[ s ]
```

The individual concept has been duplicated to produce two individual concepts with the same referent. We now consider a second case:

```
[ p ]-
      ( r1 )->[ q ]
      ( r2 )->[ s ]
```

The result is the pair of graphs:

[*p* : **x*]->(r1)->[*q*]
[*p* : **x*]->(r2)->[*s*]

We see that we are entitled (indeed compelled) to place a coreference link between the two concepts of type *p* which originated from the single concept in the original graph. Since the two concepts of type *p* refer to the same individual then we require the coreference link to maintain the semantics of the result of the operation.

We can now extend the definition of detachment to include the splitting of graphs:

Definition 3.2: Detachment

If *u* is a canonical graph and *c* is any concept of *u*, then let the resulting graph *w* be any graph that results when *c* is copied as *c'* and some relational arcs that were attached to *c* are removed and attached to *c'*. If *c* is a generic concept then insert a coreference link between *c* and *c'*.

This definition of detachment allows graphs to be split, possibly in a stepwise fashion. It is the direct reverse of the join on coreferent concepts.

As it stands above the definition does not provide for the removal from the context of detached portions of graphs. It will be shown later that such portions can be removed, where logically valid, by the rule of erasure. Thus, the combination of the rule of detachment and Peirce's rule of erasure provides a complete means of generalising graphs in a way which is not possible with erasure alone.

Since detachment is closely allied to Peirce's rule of erasure, we will later use detachment to augment the rule of erasure in the context of Conceptual Graphs theory.

3.3 GENERALISATION AND SPECIALISATION

This section discusses the more advanced transformations of graphs. It generalises some of them to whole contexts and to lines of identity, features which are not present explicitly in Sowa's theory but which are nevertheless important aspects of these operations. The following sections will not derive formal definitions for each operation as these are given in Sowa but will, instead, concentrate on the higher level extensions that are being proposed. These extensions are natural extensions which ought to have been included by Sowa.

3.3.1 Generalisation

Generalisation is a process by which the semantic content of a Conceptual Graph is weakened. Since any true sentence *s'* in any logic can be weakened to give a true sentence *s* (in which we adopt the ' notation to indicate that a ' superscripted graph is a specialisation of the same graph without a ' superscript) we can say that $s' \Rightarrow s$. According to Sowa, generalisation can proceed in any one of the following ways:

I Replacing a type label with a supertype. Thus the concept:

[TYPE_1]

can be generalised to:

[TYPE_2]

where TYPE_1 < TYPE_2 and we can say that [TYPE_1] ⇒ [TYPE_2]

II Replacing a relational label with a superrelation. Thus the relation:

->(REL_1)->

can be generalised to:

->(REL_2)->

where REL_1 < REL_2 and we can say that ->(REL_1)-> ⇒ ->(REL_2)->

III Replacing an individual referent with a generic referent. Thus the concept:

[TYPE : #x]

can be generalised to:

[TYPE : *]

and we can say that [TYPE : #x] ⇒ [TYPE : *]

This is the basic list as described by Sowa. However the following two rules need to be added:

IV Removing arbitrary parts of any graph by detachment (without leaving any relational arcs dangling). Thus the graph:

[TYPE_1]->(REL)->[TYPE_2]

can be generalised to give either (or both) of:

[TYPE_1] and [TYPE_2]

V Replacement of an individual referent which occurs in two separate concepts involves a first step of replacing both referents with generic referents joined by a line of identity and a second step of removal of that line of identity. Thus the two concepts:

[TYPE : #x] [TYPE : #x]

can be generalised to:

[TYPE : *y] [TYPE : *y]

and then to:

[TYPE : *] [TYPE : *]

Rule IV is dealt with more fully in Section 3.6.5.3. Rule V is a more general treatment of generalisation of the referent than that given by Sowa and repeated in Section III.

These rules cannot be applied to any graph whatever. We will later discuss the rules of Peirce logic in which the rules that result in a generalisation can only be applied to evenly enclosed contexts if truth is to be maintained. Indeed, these rules will be shown to completely subsume Sowa's rules of generalisation.

3.3.2 Common Generalisation

The common generalisation of two graphs *a* and *b* is that graph which is produced when a series of generalisations of *a* and a series of generalisations of *b* both result in the same graph. For any two graphs there may not be a meaningful common generalisation (other than the concept [T]), or there may be many common generalisations. The minimal common generalisation is the most specialised of all common generalisations, or, if there is more than one, it is any one of them.

3.3.3 Specialisation

Sowa's rules of specialisation allow any graph to be specialised by restricting the type labels of its concepts (subject to the conformity relations table being satisfied) and relational labels, and by joining new parts to the original graph. The specialisation of evenly enclosed graphs in arbitrary ways does not preserve truth but the checking of the conformity relations table will prevent false graphs from being formed by specialisation of type labels. There will be more to say about the role of specialisation of evenly enclosed graphs in Chapter 4 where we discuss the maintenance of a knowledge base. As previously stated, specialisations produced by the join on coreferent concepts are, however, truth preserving.

Apart from the rule allowing type labels to be specialised subject to non-violation of conformity relations, the rules of Peirce logic allow all the generalisations of Section 3.3.1 to be reversed within oddly enclosed contexts.

3.3.4 Common Specialisation

A common specialisation of two graphs *a* and *b* is that graph which has been produced when a series of specialisations of *a* and a series of specialisations of *b* both result in the same graph. For any two graphs there may not be a meaningful common specialisation (other than [ABSURD]), or there may be many common specialisations. The maximal common specialisation in the most general of all common specialisations, or, if there is more than one, it is any one of them.

3.3.5 Maximal Join

The maximal join operation joins two graphs on regions where they overlap such that the overlapping portions of each graph will project onto each other. The corresponding concepts and relations in the overlapping regions of each graph are specialised so that the result contains the most specific object from each node involved. Such overlapping regions will possess a common generalisation and the result of the join is that the overlapping regions are transformed into the maximal common specialisation of the original two regions. Since the common generalisation can be projected onto one of the regions and the projection then itself be projected onto the other the projections are called *compatible projections.* The overlap may be partial, although the topologies of each overlapping region of each graph must be identical. Since the join is stated to be maximal it must involve the largest compatible projections of the overlapping regions although there may be more than one such join.

As noted in Section 3.2.2 this operation does not preserve truth. It therefore does not have a place as a rule of inference within the set of operations that we are developing for an exact reasoning system. It does, however, have a place in any scheme that builds models by piecing together bits of evidence. We recall that we stated that the semantics of the generic referent allowed us to state that the concept containing it referred to a particular individual but that we did not know which one. Thus the use of the maximal join operation to combine concepts with generic referents is equivalent to making an hypothesis about those individuals. Appendix I will discuss a simple deductive mechanism in which hypotheses are generated and these are then tested to see if the query can be proved (generate and test).

We note from Sowa's definition of the maximal join that it is not necessarily complete since it requires that the join be over the greatest possible compatible projections of the two graphs being joined. Sowa points out that this will be sufficient in *most* cases but we must also recognise that where there are several joins possible we may need to move to those produced by a less than maximal overlap in order to find a solution. We also stated in Section 3.2.2 that we may need to make the join on single concepts whereas Sowa's maximal join will not do this. Thus we require an operation that will return all joins. We call this join the *join on compatible projections*. Whereas the maximal join specifically requires the compatible projections to be the biggest possible between the graphs we generalise this requirement and state that the join can take place on any compatible projection. This relaxation subsumes the restriction made to the definition of the join operation in Section 3.2.2. For efficiency reasons it is worth noting Sowa's claim that the most useful joins are maximal. If this is true then it is probably most efficient for an implementation of this join to generate the maximal joins first.

Definition 3.3: Join On Compatible Projections

Where two overlapping subgraphs *s1* and *s2* of graphs *g1* and *g2* share a common generalisation *g* with compatible projections onto *s1* and *s2* then graphs *g1* and *g2* may be joined on a compatible projection. Where g *is* maximally extended the join is a *maximal join on compatible projections*. For each concept *c* derived from concepts *cg1* in *g1* and *cg2* in *g2* in *g*:

* If *referent(cg1)* and *referent(cg2)* are both generic then *referent(c)* is derived from *referent(cg1)* and *referent(cg2)* separated by the equals symbol =.

* If *referent(cg1)* is generic and *referent(cg2)* is individual then *referent(c)* is derived from *referent(cg2)*. All other occurrences of *referent(cg1)* must also be replaced with *referent(cg2)*.

* *type(c)* is derived from the more specialised of *type(cg1)* and *type(cg2)*.

For each relational node *r* derived from relations *rg1* in *g1* and *rg2* in *g2* in the common specialisation *g*:

* *relation(r)* is derived from the more specialised of *relation(rg1)* and *relation(rg2)*.

3.4 ABSTRACTION AND DEFINITION

Although Sowa includes lambda abstractions as a means of definition of types and relations, and he also allows them to be placed within the type and relational fields he does not define the lambda calculus in terms of Conceptual Graphs theory. This is a serious omission since there is no means of computing anything at all with them. He also introduces Wittgensteinian notions of definition and suggests that his Schemata provide a Wittgensteinian semantics although he does not adequately show this. This section attempts to clarify these points from the point of view of attempting to define operations on conceptual graphs that will allow these definitional mechanisms to work.

3.4.1 Lambda Calculus

We saw in Chapter 2 that the type field and relational field may contain lambda abstractions. Such a lambda abstraction is equivalent to a marker for which the lambda abstraction is the definition, and as such the two are interchangeable. We also concluded that lambda abstractions within the type and relational field and elsewhere added nothing to the expressiveness of Conceptual Graphs. We contrasted this with the use of a logical equivalence and showed that the two were equally expressive. We now wish to show that the operations on lambda abstractions that Sowa described are subsumed by the more general operation, implication.

The discussion will be centred around the following type definition represented first in the lambda abstraction form, and then in the logical equivalence form developed earlier:

Lambda Abstraction:

TYPE(x) = λ(x) *graph*(*x)

where the left hand side is the *definiendum* and the right hand side is the *definiens*, or the *differentiae* in Sowa's terminology.

Logical Equivalence:

([TYPE : *x] (*graph*(*x)))

(*graph*(*x) ([TYPE : *x]))

where, in each part of the equivalence, the single concept is the *definiendum* and *graph*(*x) is the *definiens*.

Expansion And Contraction

There are two Conceptual Graphs operations associated with definition by lambda abstraction. These are *type/relational contraction* and *type/relational expansion*. When a contraction of a concept or relation in a graph *g* is performed, the *definiens* is projected onto *g* and, if a match is found, the appropriate label in *g*, that which is coreferent with the header, is specialised to that of the *definiendum*. All generic concepts and relations which are subsumed by the definition of the *definiendum* are deleted. The process of expansion works in reverse. To expand a concept, the *definiens* of the appropriate lambda abstraction is maximally joined to the concept, whose type label is replaced with that of the corresponding concept in the *definiens*. In this way it is possible to contract and subsequently re-expand any concept or relational node. We can provide an example:

Type definition: ELEPHANT(x) =
λ(x) [MAMMAL : *x]->(PART)->[TRUNK]

Given: [ELEPHANT : Clyde]

Expansion: [MAMMAL : Clyde]->(PART)->[TRUNK]

Contraction: [ELEPHANT : Clyde]

The operation of expansion contains a pitfall. We will use the above type definition to illustrate it::

Given: [MAMMAL : Clyde]->(PART)->[TRUNK : #53246]

Contraction: [ELEPHANT : Clyde]->(PART)->[TRUNK : #53246]

Expansion: [MAMMAL : Clyde]-
(PART)->[TRUNK : #53246]
(PART)->[TRUNK]

In this case the expansion has added an extra TRUNK concept since the contraction did not remove the original. In this case Sowa states that the extra trunk concept, since it is generic, must be maximally joined to the original. When this is done the operation of *maximal type expansion* is derived and this restores the original graph in this case. If the original graph had been:

[MAMMAL : Clyde]-
(PART)->[TRUNK]
(PART)->[TRUNK]

which states that Clyde is a mammal with (unusually) two trunks, contraction would give:

[ELEPHANT : Clyde]->(PART)->[TRUNK]

if we assume that the contraction subsumed only one of the trunks (since the *definiens* only contains one [TRUNK] concept). Now, however, it is not clear whether Clyde has two trunks or not since expansion of this graph could yield either:

[MAMMAL : Clyde]-
(PART)->[TRUNK]
(PART)->[TRUNK]

or

[MAMMAL : Clyde]->(PART)->[TRUNK]

depending upon whether the expansion was maximal or not. In other words,

where the expansion was maximal in the second case, information has been lost.

Implication

Where definitions are given as logical equivalences there cannot be any loss of information since there is no contraction as such. Since inherited properties are computed from implications, original graphs are not altered and conclusions follow with certainty. Instead, the proof of the existence of some inherited property of an entity or of a specialisation of a concept is carried out by a theorem proving approach. We show this by two examples, one of contraction and one of expansion. For each we assume the following equivalence:

([MAMMAL : *x]->(PART)->[TRUNK] ([ELEPHANT : *x])) (a)

([ELEPHANT : *x] ([MAMMAL : *x]->(PART)->[TRUNK])) (b)

Contraction By Implication

Since there is no operation of contraction, and therefore no removal of relations, the scenario is one in which we wish to know if some graph entails the existence of an elephant. Graph (a) can be interpreted as a rule which states that 'if there is a mammal which possesses a trunk then that mammal is an elephant'. Thus, in order to ascertain whether an elephant exists in the following context:

[MAMMAL : Clyde]->(PART)->[TRUNK : #53246]

it is only necessary to 'fire' rule (a) and generate the concept:

[ELEPHANT : Clyde].

Briefly, the computation (to be discussed fully in Chapter 4) involves the lookup of the rule and the projection of the *definiens* onto the context. Had a lambda abstraction been used to specify the definition then the computation would have involved the lookup of the lambda abstraction and the projection of its body onto the context. Computationally the two processes are very similar.

Expansion By Implication

Since there is no operation of expansion, and therefore no addition of relations, the scenario is one in which we wish to know if some graph entails the existence of a mammal which possesses a trunk. Graph (b) can be interpreted as a rule which states that 'if there is an elephant then that elephant is a mammal which possesses a trunk'. Thus, in order to ascertain whether a mammal which possesses a trunk exists (or indeed whether there exists a trunk) in the following context:

[ELEPHANT : Clyde]

it is only necessary to 'fire' rule (b) and generate the graph:

[MAMMAL : Clyde]->(PART)->[TRUNK].

Again, the computation by use of either the rule or the lambda abstraction requires the lookup of the definition and the projection of the *definiens* onto the knowledge base.

In neither of these examples is a mechanism for the firing of rules given, since such a description requires reference to the rules of logic which have yet to be covered. We will show in Chapters 4 and 5 that such a mechanism exists and is sufficiently general to fire any arbitrary rule, not only rules of the forms given here. We will argue that this mechanism, when taken as part of an overall proof procedure, is as efficient as possible mechanism for the same kind of inference using contraction and expansion and we will argue that the use of a single mechanism for all forms of rule firing reduces the search space by reducing the options available to a proof mechanism. Not only that but the use of implications removes the possibility of spurious expansions.

3.4.2 Operations With Schemata And Prototypes

We have already stated that schemata and prototypes will not feature in our system in the form presented by Sowa. Nevertheless, we will mention that Sowa included a *schematic join* operation, which proceeded in a manner similar to type expansion and which added default knowledge to a graph. We will note that the addition of such knowledge is not truth maintaining since, by definition,

a schema only contains information that may possibly but not necessarily (albeit usually) be associated with a given concept.

The position with prototypes is that Sowa did not provide any information at all about how he envisaged them to be processed. Presumably he intended that some form of graph comparison operation between a description of an individual and a prototype for that individual's type be defined, but what the theory behind this might be is not clear. The development of this theory is beyond the scope of the present work.

3.5 AGGREGATION AND INDIVIDUATION

The operation of *aggregate individual expansion* is intended to add background knowledge to some graph in order, presumably, to allow some proof to proceed. We argued in Chapter 2 that there is no real boundary between knowledge that is part of an individual's definition and what is not and so we did not include individual definitions in our scheme. Nevertheless we briefly include the operation of aggregate individual expansion in our review.

Given the aggregate individual:

individual TYPE(individual_marker) **is**

graph

the operation of aggregate individual expansion of a concept *c* [TYPE : individual_marker] proceeds by joining *graph* to *c* and then simplifying. Since this operation may be needed during a proof, perhaps when all else has failed, its use would necessitate the system backtracking to the point where additional information about the individual is needed and then the expansion is carried out. Since there may be many such points where an expansion might lead to an otherwise unobtainable result the search space of the whole proof may grow to be very large. For this reason, it would be computationally more efficient to start each proof with all individuals fully expanded. Where the knowledge base is organised so that aggregate individual definitions are not used then each proof must start in this state by default. Thus we adopt the approach of not using individual definitions or aggregate individual expansion.

3.6 LOGICAL OPERATIONS - PEIRCE'S RULES

This section provides a survey of Peirce's Beta rules of inference as interpreted by Sowa [Sowa84] and applied to conceptual graphs. Following a brief listing of the rules as set out by Sowa we analyse them and identify a set of primitive and independent rules. Following on from this we recombine these primitive rules into 5 derived rules of inference which will provide a sound and complete logical basis for the computational theory to be discussed in Chapter 4. These rules will be the dual of Peirce's original rules.

3.6.1 Survey Of The Complete Set Of Rules As Given By Sowa

Peirce's Beta rules of inference are equivalent to first order predicate calculus. This has been shown by others [Roberts73] and there is not any attempt made in this work to prove this fact. The Beta rules are given by Sowa in the form of the following seven basic rules:

3.6.1.1 **Erasure** - in an evenly enclosed context any graph may be erased, any coreference link from a dominating concept to an evenly enclosed concept may be erased, any referent may be erased and any type label may be replaced with a supertype.

3.6.1.2 **Insertion** - In an oddly enclosed context any graph may be inserted, a coreference link may be drawn between any two identical concepts and restriction may be performed on any concept.

3.6.1.3 **Iteration** - A copy of any graph *u* may be inserted into the same context in which *u* occurs or into any context dominated by *u*. A coreference link may be drawn from any concept of *u* to the corresponding concept in the copy of *u*. If concepts *a* and *b* in some context *c* are both dominated by a concept *d* on some line of identity then a coreference link may be drawn from *a* to *b*.

3.6.1.4 **Deiteration** - Any graph or coreference link whose occurrence could be the result of iteration may be erased. Duplicate conceptual relations may be erased from any graph.

3.6.1.5 **Double negation** - A double negation may be drawn around or removed from any graph in any context.

3.6.1.6 **Coreferent join** - Two identical coreferent concepts in the same context may be joined and the coreference link between them may then be erased.

3.6.1.7 **Individuals** - If an individual concept *a* dominates a generic concept *b* where *a* and *b* are coreferent then *referent*(*a*) may be copied to *b* and the coreference link may be erased.

Axiom - The empty set of graphs { } is the only logical axiom. Any graph that is provable from { } by these rules is called a *theorem.*

This set of rules is to Sowa's adaptation of Peirce's *permissions* to conceptual graphs. We must clarify some aspects of them and show how they must be augmented to deal with the changes made in Chapter 2 to the notation. In particular, we argued in Chapter 2 that there ought to be extensions to Sowa's notation to allow the handling of higher order logics and to allow the use of a relational lattice. These extensions were in the kinds of objects allowed in each field. Thus we wish to allow generic objects and coreference links in the type and relational fields and the generalisation and specialisation of relational labels. We extend Sowa's rules to permit these operations. We also discuss Sowa's statement that the only logical axiom is { }.

Erasure

The wording of the definition of the rule of erasure given above suggests that the following derivation is allowed:

$$\{[\,P : *x\,]\,([\,Q : *x\,])\} \Rightarrow ([\,Q : *x\,])$$

However we must consider this situation carefully. Since the concept [P : *x] is intended to represent some entity that actually exists it follows that the graph ([Q : *x]) states that the particular entity *x is not of type Q. The erasure of [P : *x] from the above context leaves a graph which states that there are not *any* Qs. This fallacy results from the erroneous disappearance of the line of identity between the two concepts, the deeper nested of which is oddly enclosed. Lines of identity to oddly enclosed graphs can only be removed by

deiteration. Thus we see that evenly enclosed graphs can therefore only be erased when they are not connected to a more deeply nested oddly enclosed graph by any line of identity. The disappearance of lines of identity to more deeply nested evenly enclosed graphs can be accounted for by erasure and is therefore sound.

To view this another way, we must consider that the line of identity has some 'real' existence in the same way as any other part of a conceptual graph. By definition, a line of identity must connect two distinct fields. When an evenly enclosed concept such as [P : *x] above is erased then either the line of identity must be considered to 'evaporate' or it is left 'dangling'. Since the result is either the spurious removal of information or a syntactically malformed graph, neither of these situations is satisfactory and so no graph can be erased until all attached lines of identity have been removed by legitimate means.

Iteration

The rule of iteration allows two concepts *a* and *b* in some context *c* which are both dominated by a concept *d* on some line of identity to have a coreference link drawn between them. Thus the following derivation is possible:

{ [P : *x=*y] ([P : *x] [Q : *y]) }

⇒

{ [P : *x=*y] ([P : *x=*z] [Q : *y=*z]) }

In other words, a new line of identity has been added between the two oddly enclosed concepts [P : *x] and [Q : *y]. It is also possible to interpret this aspect of the rule of iteration as allowing the direct derivation:

{ [P : *x=*y] ([P : *x] ([Q : *y])) }

⇒

{ [P : *x=*y] ([P : *x=*z] ([Q : *y=*z])) }

where [P : *x] and [Q : *y] are now at different depths. But this is not the case since the next steps are:

Deiteration of line of identity: { [P : *y] ([P : *z] ([Q : *y=*z])) }

Erasure of line of identity: { [P] ([P : *z] ([Q : *z])) }

Erasure of evenly enclosed graph: { ([P : *z] ([Q : *z])) }

In other words we have shown that all Ps are Qs from an original statement that there existed a P such that a particular P implied the existence of a particular Q. Thus the two concepts *a* and *b* between which the line of identity can be iterated must be in the same context and not separated by any cuts. This interpretation is suggested in 3.6.1.3 when it refers to 'concepts *a* and *b* **in some context *c***' but it is not stated as clearly as it might be.

Erasure And The Rule Of Detachment

In Section 3.2 we discussed Sowa's discarded rule of detachment and stated that the use of this rule was only truth-maintaining when applied to evenly enclosed simple graphs. Peirce's rule of erasure allows whole graphs to be erased from evenly enclosed contexts and yet we see that the removal of relations from graphs leaves graphs that are implied by the original graph. Thus we can augment the rule of erasure by incorporating the rule of detachment.

Axioms

Sowa states that the only logical axiom is { }. Peirce, however [Roberts73], stated the additional *permission* for the Beta system:

> *Permission No.9. It is permissible to scribe an unattached line of identity on the sheet of assertion,...*

This is equivalent to the model theoretic assumption that there exists a set *I* of individual markers. For conceptual graphs this means that for markers #1..#$_n$ the concepts [T : #1] ... [T : #$_n$] are derivable from { }.

Completion Of Adaptation Of Peirce's Beta Rules To Conceptual Graphs

We now state the following set of rules of inference:

3.6.1.8 **Erasure** - in an evenly enclosed context any graph, or part of a graph, which is not linked by any line of identity to a more deeply nested field may be removed, any coreference link from a dominating field to an evenly enclosed field may be removed, any marker may be removed and any type label or relational label may be replaced

with a supertype or superrelation.

3.6.1.9 **Insertion** - In an oddly enclosed context any graph may be inserted, a coreference link may be drawn between any two identical fields and restriction may be performed on any type label or relational label.

3.6.1.10 **Iteration** - A copy of any graph *u* may be inserted into the same context in which *u* occurs or into any context dominated by *u*. A coreference link may be drawn from any field of *u* to the corresponding field in the copy of *u*. If fields *a* and *b* in some context *c* and not separated by any cuts are both dominated by a field, or a pair of fields in the same context linked by some line of identity, then a coreference link may be drawn from *a* to *b*.

3.6.1.11 **Deiteration** - Any graph or coreference link whose occurrence could be the result of iteration may be removed. Duplicate conceptual relations may be removed from any graph.

3.6.1.12 **Double negation** - A double negation may be drawn around or removed from any graph or set of graphs in any context.

3.6.1.13 **Coreferent join** - Two coreferent concepts in the same context may be joined and the coreference link between them may then be removed.

3.6.1.14 **Individuals** - If a marker *a* dominates a generic object *b* where *a* and *b* are coreferent then *a* may be copied to *b* and the coreference link may be removed. Any type label *t1* in *b* which is a supertype of the type label *t2* to which *a* conforms may be replaced with *t2*.

Axioms

There are two logical axioms:

The empty set { }.

For any individual marker *I* the concept [T : *I*] may be added to the sheet of assertion.

3.6.2 Analysis Of Peirce's Rules

Whilst the rules presented in Section 3.6.1 contain all that is required, most of them are composed of several independent rules which are placed together because they are semantically related. Each will now be dissected.

3.6.2.1 Erasure

The rule of erasure allows any evenly enclosed graph to be generalised in some way. With reference to 3.6.1.1 there are four ways in which objects within an evenly enclosed context can be erased:

* Erasure of graphs - which includes erasure of subgraphs and relations subject to the graph not being connected by lines of identity to a more deeply nested graph.

* Erasure of lines of identity - any generic object can have its coreference marker removed if the coreferent field is dominant.

* Erasure of type labels - which is the replacement of a type label with one of its supertypes.

* Erasure of markers - which is the replacement of a marker with a generic object.

In addition to these there is an additional rule which is not given by Sowa but which is nevertheless implied by him. This additional rule of erasure is:

* Erasure of relational labels - which is the replacement of a relational label with one of its superrelations.

3.6.2.2 Insertion

The rule of insertion allows any oddly enclosed graph to be specialised in some way. With reference to 3.6.1.2 there are four ways in which objects within an oddly enclosed context can be inserted:

* Insertion of graphs - which includes insertion of conceptual relations between concepts.

* Insertion of lines of identity - any generic field can have added to it a coreference link to a field which occurs in a dominating context.

* Insertion of type labels - which is the replacement of a type label with one of its subtypes.

* Insertion of markers - which is the replacement of the generic object with a marker.

In addition to these there is an additional rule which is not given by Sowa but which is nevertheless implied by him. This additional rule of insertion is:

* Insertion of relational labels - which is the replacement of a relational label with one of its subrelations.

3.6.2.3 Iteration

The rule of iteration allows tautologies to be created by copying graphs or lines of identity to the same context or to more deeply nested contexts within the context of the existing graph or line of identity. There are two ways in which iteration can be performed:

* Iteration of graphs - which includes the iteration of all types of graph, not only simple graphs.

* Iteration of lines of identity - which is the addition of a line of identity between any field and a deeper nested copy.

3.6.2.4 Deiteration

The rule of deiteration is the exact reverse of that of iteration. By deiteration any graph which is a copy of a less deeply nested graph can be removed and any field which is identical to a corresponding, less deeply nested one can be replaced with a generic referent. The two forms are:

* Removal of deeper nested copies of graphs.

* Removal of lines of identity between fields and deeper nested copies.

3.6.2.5 Double Negation

The rule of double negation allows a double negation to be added to or removed from any graph or conjunction of graphs whatever. It therefore exists in two forms:

* Addition of a double negation.

* Removal of a double negation.

3.6.2.6 Coreferent Join

The rule of coreferent join allows the connection of graphs in some context which contain concepts sharing the same referent, as stated in Section 3.2.2.

3.6.2.7 Individuals

There are two forms of this rule:

* The copying of markers along lines of identity.

* The restriction of type labels within individual concepts.

We note that when any concept [T1] which acquires an individual referent #x to become [T1 : #x] such that:

T2 < T1 &
T2 :: #x

that concept may be specialised to [T2 : #x]. We prove this by noting that the relation T2 :: #x could only be added to the conformity relations table in the case where it was certain that the concept [T2 : #x] was part of a true graph.

Replacement of a type label with a subtype label in this way is termed *restriction*, and is identical to the canonical formation rule of that name when the conformity relations table is taken into account.

3.6.3 Derived Rules Of Inference And Duplication Of Operations

The rules given in Section 3.6 have been shown to be complete [Roberts73]. In addition to these rules, Sowa also gives us several other rules

and operations that can be performed on conceptual graphs (as previously described). This section shows that each of these is either a special case of a sequence of Peirce's rules or can easily be performed by them, with one exception. The purpose of the demonstration of the equivalences is to show that many of the conceptual graphs operations given by Sowa are not strictly necessary within a system that retains the full power of the formalism.

3.6.3.1 Canonical Formation Rules

* Copy - this rule is identical to the rule of iteration where the iterated graph is placed into the same context as the original and is therefore subsumed by the rule of iteration.

* Join - the join on identical concepts is identical to the rule of coreferent join where identical concepts are joined and is therefore subsumed by the rule of coreferent join.

* Restriction - the rule of restriction allows any concept whose referent is known to conform to a more specialised type label than that given to be restricted. It also allows the addition of a referent to a generic concept of an appropriate type. In an oddly enclosed context these two operations can be performed at will whilst in an evenly enclosed context they may be performed by iteration. The rule of restriction is a canonical formation rule and these rules preserve falseness. Therefore for any logic based truth maintaining system the rule must only be used in cases where there is a known conformity relation. This means that restrictions would always be performed by iteration or insertion.

* Simplify - this rule is equivalent to deiteration of duplicate relations from one context and is therefore unnecessary.

3.6.3.2 Generalisation

The rules of generalisation allow information to be removed from any simple graph. We previously showed that the graphs that are generalised can only retain truth if they are evenly enclosed. Therefore the rules of generalisation correspond to the operations provided by the rule of erasure.

3.6.3.3 Common Generalisation

The rules of generalisation also allow for common generalisations between two graphs to be made. As generalisation of oddly enclosed graphs does not preserve truth it does not make sense to attempt it with these. Generalisation of evenly enclosed graphs follows directly from the rule of erasure and therefore so too does the generation of common generalisations.

3.6.3.4 Specialisation

Specialisations are made by restricting type labels and referents or by joining graphs on identical concepts. Therefore the comments made above concerning restriction and join are appropriate here.

3.6.3.5 Common Specialisation

A common specialisation is simply a graph that is produced by specialisations of two different graphs that result in the same graph. Once again these specialisations are covered by the comments about restriction and join.

3.6.3.6 Maximal Join

The maximal join operation does not preserve truth and is therefore impossible within the rules of Peirce logic. However, it is interesting to attempt a maximal join by Peirce logic and note the point at which further progress is impossible. This point is always that at which non-coreferent concepts, at least one referent of which is generic, are joined. Unless a line of identity can be established, Peirce logic cannot join such concepts.

The future purpose of the maximal join operation will be in the generation of theories and therefore it should be retained as an operation of conceptual graphs. In this role it subsumes the operation of specialisation.

3.6.4 Iteration And Linear Notation

This section provides a detailed survey of Sowa's definitions and assumptions on the nature of the generic referent and the line of identity. Its motivation is the discovery that Sowa's adaptation of Peirce's Beta rules to Conceptual Graphs drawn in linear notation can generate inconsistent graphs

and is thus unsound. It also shows that Sowa's semantics of the generic referent are extremely vague. Evidence in the form of statements from Sowa's book is presented to show that there is a subtle difference between the display and linear forms of Conceptual Graphs notation. Also it is shown that some of Sowa's assumptions and definitions are contradicted by his own examples and that the inconsistency arises from a notational defect with the linear form and not from a fundamental logical problem.

Where possible, the text will contain direct quotations from Sowa's book, but where these are too long a reference will be given. It is essential that such references be read at the appropriate point in the text.

Evidence

The following paragraphs cite 10 pieces of evidence from Sowa about the nature of and operations on generic referents and coreference links.

1) Page 78 introduces the first inkling of the notion of variables in conceptual graphs. It is brought in at this point in the book to show how to write a linear form graph which contains cycles. The notation for a variable is '*x' and this shows cross references between different concepts, as written in linear form, which are actually the same concept. Nothing more is said about the behaviour of these objects at this point.

2) Page 84 gives what he terms (in the index) a 'formal definition' of the generic referent. This formal definition states that:

> '...: they [generic concepts] are like variables that represent an unspecified individual of a given type.'

Page 85 then goes on to give *assumption 3.3.1* which states that:

> *referent(c)* is either an individual marker in *I* or the *generic marker* *.
>
> ...
>
> When *referent(c)* is *, then *c* is said to be a *generic concept*.

Some examples of generic and individual concepts are then given. These reiterate the idea given in the 'formal definition' of the generic concept.

3) Page 86 states in the first paragraph that:

'In fact, variables like *x or *y in the linear notation are simply the generic marker *, followed by an identifier for indicating cross references.'

Two equivalent graphs are then given; one contains the referents *x, *y and *z and the other contains only * referents (although the notational concession that allows this referent to be omitted is used).

4) Page 90 gives further information:

'In the linear form, the * for generic concepts may be followed by a variable name, such as *x or *abc.'

5) Page 104, Definition 3.6.1 uses *x and *y as formal parameters to a lambda abstraction. This example is given in linear form but page 106 gives another example, this time of a type definition, in display form. Further examples are also given in the same section.

6) Page 141 introduces the line of identity for the first time. Fig 4.7 gives an example of its use and its display form representation. This line '... shows that two nodes represent exactly the same individual.'. Page 142 then goes on to give an assumption (Assumption 4.2.5) about lines of identity:

4.2.5 Assumption. A *line of identity* is a connected, undirected graph *g* whose nodes are concepts and whose arcs are pairs of concepts, called *coreference links.*

No concept may belong to more than one line of identity.

A concept *a* in *g* is said to *dominate* another concept *b* if there is a path $<a_1,a_2,...,a_n>$ in *g* where $a=a_1$, $b=b_n$, and for each *i*, either a_i or a_{i+1} both occur in the same context or the context of a_i dominates the context of a_{i+1}.

Two concepts *a* and *b* are *coreferent* if either *a* dominates *b* or *b* dominates *a*.

A concept *a* is *dominant* if *a* dominates every concept that dominates *a*.

The whole of Section 4.2.5 is worth careful reading, especially the part that

introduces the = symbol. This must be borne in mind when studying what Sowa has to say about the rule of iteration. Section 4.2.6 should also be read.

7) Page 147 gives an example of a graph which contains two lines of identity, given here in pseudo display form and using Sowa's notation for negation:

which says that there exist at least two people. A linear form equivalent of this graph is also given:

[PERSON : *x] [PERSON : *y] ¬[[PERSON : *x=*y]]

This graph is important because it establishes an equivalence of the display form of the line of identity and the linear form variable, along with the role of the = symbol as a separator.

8) Sowa's adaptation of Peirce's rules require careful interpretation. The rules of insertion and iteration, as described in Assumption 4.3.5 are the ones relevant to the present discussion and some ambiguity in iteration has already been shown. These are the only two rules that we have that allow coreference links to be added to any graph.

9) Page 155 shows how the rule of *existential generalisation* follows from the rule of erasure. In other words, it shows the possibility of the derivation:

[TYPE : #n] => [TYPE : *]

which is equivalent to a derivation of the form:

pred(individual_marker) => ∃(x)pred(x)

10) Page 170 describes an example of an evaluation game. It contains the following two statements:

> 'In the linear notation, the coreference links are indicated by variable symbols *x and *y.'
>
> 'The act of erasing coreference links corresponds to erasing the variable symbols:'

Example Derivation In Linear Notation

The following example shows a very straightforward derivation. It is given in linear notation:

Given: [PET_CAT : felix][CAT : tiddles]
([PET_CAT : tiddles]) (1)
PET_CAT < CAT
PET_CAT < PET

Graph (1) has the interpretation that there is a pet cat 'called' felix and there is another cat 'called' tiddles but tiddles is not a pet cat. The type lattice is surely beyond doubt.

Copy: [PET_CAT : felix] (2)

This graph can be considered as having been copied (canonical formation rule) from (1) to some workspace. The rule of erasure could also have been used to generate (2) from (1) by removing everything else.

Erasure: [PET_CAT : *] (3)

Iteration: [PET_CAT : *] [PET_CAT : *] (4)

The rule of iteration is used here, rather than copy, to draw attention to another property of this rule to be used next.

Iteration: [PET_CAT : *x] [PET_CAT : *x] (5)

The rule of iteration allows any copy of a concept that could owe its existence to the use of the rule of iteration to be joined to the 'parent' by a line of identity, shown as *x in linear form.

Erasure: [CAT : *x] [PET_CAT : *x] (6)

In an evenly enclosed context (including depth 0) any type label can be replaced with any supertype.

Double negation: [CAT : *x] [PET_CAT : *x] (( )) (7)

A double negation can be placed anywhere at will and we choose to place it

here.

Iteration: [CAT : *x] [PET_CAT : *x] ([CAT : *x] ( )) (8)

A copy of any graph may be copied to any context dominated by the parent graph.

Iteration: [CAT : *x] [PET_CAT : *x]
([CAT : *x] ([PET_CAT : *x])) (9)

Justified as step (8).

Deiteration: [CAT : *] [PET_CAT : *x]
([CAT : *x] ([PET_CAT : *x])) (10)

Since the evenly enclosed concept [CAT : *x] is joined to the oddly enclosed copy by a line of identity labelled *x this line could have arisen by iteration and can be deiterated. We do not wish at this stage to remove any other line so we simply remove the x from the evenly enclosed concept. This breaks the link without affecting anything else.

Erasure: [CAT : *] [PET_CAT : *]
([CAT : *x] ([PET_CAT : *x])) (11)

Any line of identity which extends to an evenly enclosed graph can be erased. Again we wish to retain the remaining line of identity and so the deeply nested PET_CAT concept retains the x. The line of identity could also have been removed by deiteration, as in step (10), in this case.

Erasure (twice): ([CAT : *x] ([PET_CAT : *x])) (12)

Any evenly enclosed graph can be erased if the erasure does not leave any 'dangling' lines of identity. We have erased the two graphs at depth 0 in this one step.

Graph (12) has the interpretation that all cats are pet cats. However this conclusion is not the case in this example because there is an exception (tiddles is not a pet cat) stated in the original formula (formula (1)) and in any case it should not be possible to make such sweeping generalisations from only one instance.

There is a problem here. The rules of inference have apparently allowed us to derive an inconsistent rule from our original set of premises. Nevertheless, Peirce's rules were shown to be sound and complete by Roberts [Roberts73]. This can only suggest that either Sowa's adaptation of Peirce's rules to conceptual graphs or conceptual graphs themselves are flawed.

Example Derivation Using Display Notation

We now attempt the same derivation using display notation (although this will be simulated in linear notation for compactness, with the added liberty of allowing lines of identity to connect whole concepts with the understanding that it is referents that are linked). The sequence of steps will be the same as before and so the justifications will be the same.

Given: [PET_CAT : felix][CAT : tiddles]
([PET_CAT : tiddles]) (13)
PET_CAT < CAT
PET_CAT < PET

Copy: [PET_CAT : felix] (14)

Erasure: [PET_CAT : *] (15)

Iteration: [PET_CAT : *] [PET_CAT : *] (16)

Iteration: [PET_CAT : *]----------[PET_CAT : *] (17)

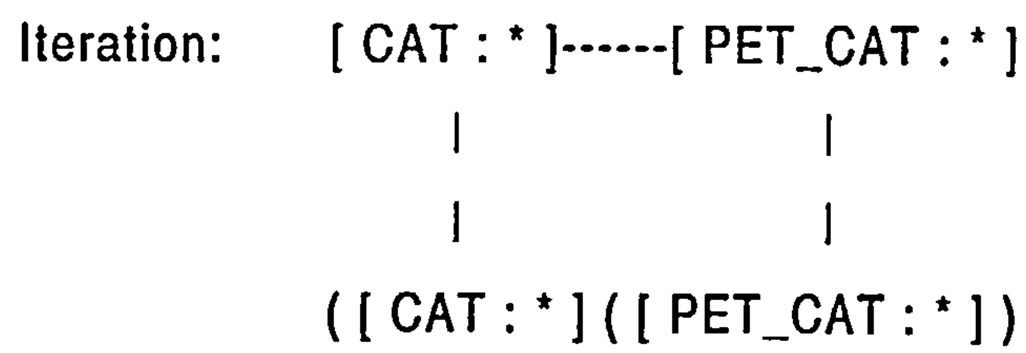
Erasure: [CAT : *]----------[PET_CAT : *] (18)

Double negation: [CAT : *]----------[PET_CAT : *] (( )) (19)

Iteration: [CAT : *]----------[PET_CAT : *] ([CAT : *] ( )) (20a)

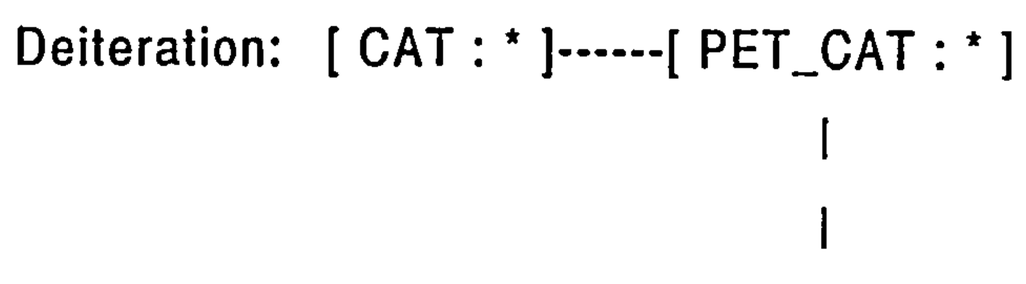
Iteration: [CAT : *]----------[PET_CAT : *]
|
|
([CAT : *] ( )) (20b)

The rule of iteration has been used twice here to achieve the same effect as the corresponding single iteration in the linear notation example.

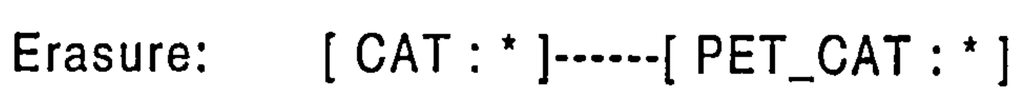


(21)

This iteration step took place in two similar stages to step (20).



(22)



([CAT : *] ([PET_CAT : *])) (23)

Erasure: ([CAT : *] ([PET_CAT : *])) (24)

This step could be seen to have taken place in two stages: the erasure of the line of identity and then the erasure of the two concepts.

Notice that the result of this derivation is a graph with the interpretation 'it is false that there exists a cat and not a pet cat' which is radically different to the interpretation of the result of the first derivation and is also quite sound.

Argument

The example presented in linear notation showed that it was possible to derive inconsistent graphs from a consistent set of premises. However, the same example presented in display notation showed that it was not. The only difference between the two cases is that one used the linear notation whilst the other used the display notation. It is therefore necessary to look at these two notations more closely. In what follows, a reference such as E*n* corresponds to one of the ten references from Sowa given earlier.

Variables

Assuming that the evidence given here from Sowa is complete, it can be seen that the reader is expected to form his own opinions about certain aspects of what the generic referent is made up of. In particular, E1 introduces the idea

of a variable as being something with the form *x. This variable is used to indicate cross references when a graph with a cycle is written in linear notation. E2 then goes on to state that generic concepts are like variables and contain the generic marker *. E3 then states that variables *in linear notation* are simply the generic marker * followed by an identifier. Indeed, Sowa then tells us that, in some cases, the identifier can be dropped. E4 tells us that, *in the linear notation*, the * may be followed by a variable name.

From this evidence we conclude that both * and *x are semantically equivalent; both are variables. The only difference between them is that one form is named and can be used to show cross references between concepts which would have drawn as a single concept in display notation. Both E3 and E4 suggest that the named version (*x) is a linear notation device. However, E5 also gives cases where the display notation can contain the same notation.

Lines Of Identity

The line of identity is introduced in E6. It is defined as a series of lines that join concepts which represent the same individual. Each segment of the line, any segment that joins two concepts, is called a coreference link. Thus, a line of identity can be composed of several coreference links. E6 also includes Assumption 4.2.5 which states that no concept may belong to more than one line of identity. E7 then shows us an example of a graph with a line of identity. This graph (*q.v.*) is shown in a pseudo display notation, but the correct (Sowa's correct) display notation for the coreference link is used - that of the broken line which joins two concepts but *does not enter* the concept box. This notation is used (but not exclusively) throughout Section 4.2 to show coreference links. What is important about this particular example is that this display notation graph is then mapped to an equivalent linear notation graph in which the coreference link is shown in the form of a named variable *inside* the concept box and *part of* the referent. E10 gives another example of this. Thus the coreference link is nothing more than a named variable used to show cross references.

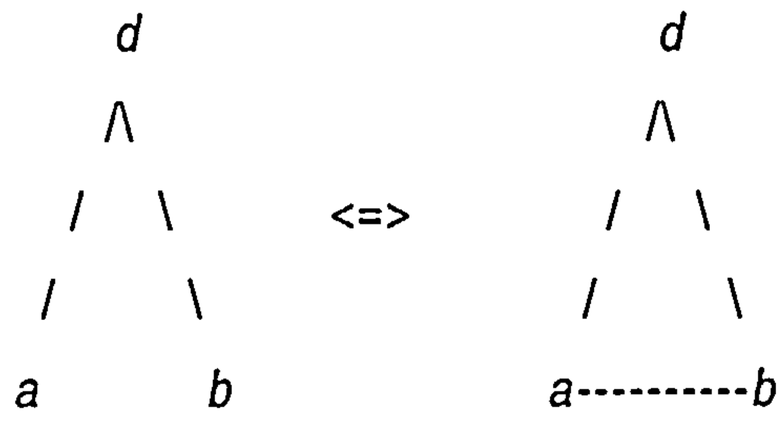
The evidence about lines of identity therefore shows them to be composed of exactly the same objects that the cross references shown by variables are composed. Therefore there is not any semantic difference between coreference links and lines if identity at all.

Peirce's Rules

The derivations given in the linear and display notations can be seen to have produces exactly the same results up to stages (8) and (20). At stages (9) and (21) the picture starts to change. Graphs (9) and (21) do not say the same thing. The rule of inference used at this point is the rule of *iteration*. We will ignore all other rules because they either were not used at all in the derivation or they produced identical results in each (but the reader is directed to E8 and E9 for further information).

The rule of iteration allows any graph to be copied either to the same context in which it occurs or to any context which is nested within the context in which it occurs. Following the copy a coreference link may (if desired) be added between any concept in the original graph and the corresponding concept in the copy of the graph. The wording of this section, which is reproduced in E8 from Section 4.3.5 of Sowa, is critical to the following arguments. It is important to notice that the coreference link *may* be added *after* the iteration of a graph, *if desired*, but is *not* an integral part of the iteration.

There is an additional part to the rule of iteration which deals with the following case as described previously (section 3.6.1.7). The general pattern for this rule is:



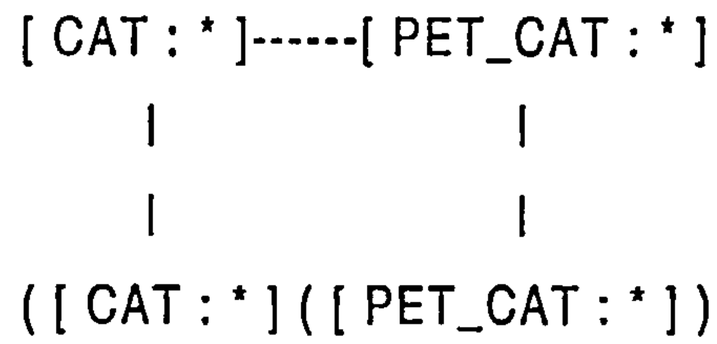
(25)

Note that, following the previous discussion, *a* and *b* must not only both be dominated by a single concept *d* (a situation which can be generated in both example derivations) but must also not be separated by any context boundaries whatsoever.

Result

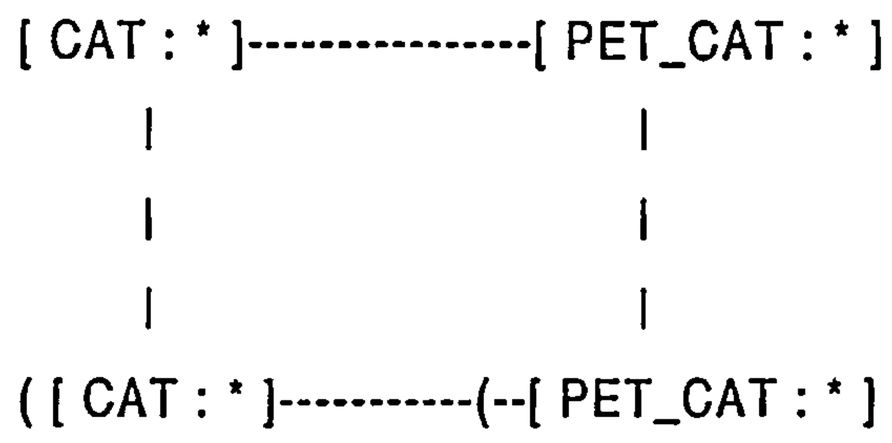
The difference between the results of the derivations in each notation is the presence of the line of identity between the concept at depth 1 and the

concept at depth 2 in (10) and its absence in (24). Indeed it is not possible to generate this line of identity at any stage when the display notation is used for the derivation. To see why we need to study graph (21) a bit further:



(21)

In order to allow the spurious derivation we need to be able to convert (21) to:



(26)

At first sight this may seem intuitively possible because there is a line of identity joining all concepts anyway. However, the rule of *insertion* (which allows coreference links to be added arbitrarily to oddly enclosed concepts) does not allow lines of identity to be added to evenly enclosed concepts at will, so this rule will not allow the extra line of identity to be added between the concepts [CAT : *] at depth 1 and [PET_CAT : *] at depth 2 of (21). The only other rule that adds lines of identity is iteration. Within this rule there are the two possibilities previously described. The first is that the [PET_CAT : *] concept at depth two in (21) arrived there by an iteration from the [CAT : *] concept at depth 1 in (20). This cannot, however, be the case here because its type label (PET_CAT) is more specific than CAT. In order to produce [PET_CAT : *] at depth 2 from [CAT : *] we would have to specialise the concept, but this cannot be done with an evenly enclosed concept. The second possibility is the one which is depicted in (25) above. This derivation does not apply, however, because the two concepts are not in the same context. If the derivation in (26) were allowed between concepts *a* and *b* which were separated by context boundaries then the false derivation would follow immediately from (21).

The question is 'why can the spurious coreference link be generated when a derivation is performed in linear notation and not when it is performed in display notation?'. This is especially intriguing when we have shown the

equivalence of the two notations from the evidence in Sowa. In fact, we have not shown any such equivalence at all and indeed have shown that the two notations are subtly different.

The difference between the linear notation and the display notation is largely in the way that coreference links are shown. The display notation uses broken lines between concepts and the linear notation uses named variables. However, there is one point about the display form that is very subtle; that is that the broken line is *not* part of the referent of either of the concepts that it joins. Whilst this is not too explicit in Sowa the evidence for it can be found. In particular, Sections 4.2 onwards show several examples of display notation (*q.v.*) and there is not a single case where the broken line is shown as part of the referent.

Further evidence for the distinction between a coreference link and its attached concepts can be found in E8. This suggests that the coreference link is separate from the concept because the rule of iteration allows coreference links to be added after the iteration of a graph and then only if this is desired. In other words, the iteration of a graph does not imply the automatic and inevitable existence of such links. Unfortunately, Sowa does not give any example iterations using display notation, but the wording of the definition of the rule of iteration does state that the iteration of a coreference link is not compulsory.

When we examine the situation with the linear notation we find that the coreference link as represented by named variables *is part of the referents* of the concepts to which it is attached. This is a different situation to that for the display notation. Now, the iteration of a concept with a named variable in its referent field results in that named variable being copied automatically. The semantics of the coreference link and iteration operations are therefore quite different when linear notation is used. In particular, the fallacy of the linear notation example rested upon the derivation of the coreference link between the two concepts at depth 1 and 2 of (10). The display notation example, on the other hand, showed that this link could not be formed. The mapping between linear notation and display notation is flawed.

If we want linear notation to map 1:1 to display notation then we must make the following observations:

* The variable name in linear notation is equivalent to the broken line in display notation.

* If the broken line is not part of a concept, but is simply attached to it, then the variable name is similarly not part of the concept.

* The above observation implies that the iteration of any generic concept can only result in a copy that does not contain any variable names whatever. In other words, the linear notation concept [T : *x] will produce upon iteration the concept [T : *]. This is equivalent to the display notation concept [T : *]--- producing upon iteration the concept [T : *].

* The subsequent addition of a coreference link by iteration must generate a link with a unique name. This is reasonable because in the display form, such a link would be a separate and distinct broken line.

Significance

The significance of this result is not merely of academic or theoretical interest. It also influences the way in which concepts are stored in a computer and the manner of implementation of certain operations, in particular, graph copying operations. Any data structure that contains the names of coreference links within the referent fields of concepts must take into account the logical fallacies that might arise by indiscriminate copying of such names. Thus, if any iteration operations are carried out by simply making a copy of the whole of such a data structure then an inconsistency has been introduced into the system.

3.7 PROJECTION

In the discussions which follow we will adopt the following terminology:

* *projector* - if a graph *u* is projected onto a graph *v* then *u* is the projector.

* *projectee* - if a graph *u* is projected onto a graph *v* then *v* is the projectee.

3.7.1 Projection Of Simple Graphs

We illustrate the projection of a simple graph *u* onto a simple graph *v* with examples. The following type hierarchy will be assumed:

MAN < PERSON
WOMAN < PERSON
BOY < MAN
GIRL < WOMAN

The following relational hierarchy will be assumed:

FATHER < PARENT
MOTHER < PARENT
SON < CHILD
DAUGHTER < CHILD

In each case the graph *v* is:

```
[ MAN : John ]-
        ( SPOUSE )->[ WOMAN : Elizabeth ]-
                                ( SON )->[ BOY : Anthony ]
                                ( DAUGHTER )->[ GIRL : Pamela ],
        ( SON )->[ BOY : Anthony ]
        ( DAUGHTER )->[ GIRL : Pamela ].
```

Case 3.7.1.1: Let *u* be the graph:

```
[ MAN : John ]->( SON )->[ BOY : Anthony ]
```

This is the most trivial case of Projection in which the graph *u* is an exact subgraph of the graph *v*. $\Pi(u,v)$ is the set of graphs:

```
{ [ MAN : John ]->( SON )->[ BOY : Anthony ] }
```

Case 3.7.1.2: Let *u* be the graph:

```
[ MAN : John ]->( CHILD )->[ PERSON : * ]
```

In this case the graph *u* is more general than it was in Case 3.7.1.1. As a result there are two subgraphs of *v* that match *u*. Thus $\Pi(u,v)$ is:

{ [MAN : John]->(SON)->[BOY : Anthony]

[MAN : John]->(DAUGHTER)->[GIRL : Pamela] }

These two graphs match graph *u* since in each case the relational label is a subrelation of CHILD, the type label in the right hand concept of each graph is a subtype of PERSON and the referent in the right hand concept of each graph is an individual marker which matches the generic marker of graph *u*.

Case 3.7.1.3: Let *u* be the graph:

[PERSON : *]->(CHILD)->[PERSON : *]

In this case $\Pi(u,v)$ is the set containing the following four subgraphs of *v*:

{ [MAN : John]->(SON)->[BOY : Anthony]

[MAN : John]->(DAUGHTER)->[GIRL : Pamela]

[WOMAN : Elizabeth]->(SON)->[BOY : Anthony]

[WOMAN : Elizabeth]->(DAUGHTER)->[GIRL : Pamela] }

The projection operation is similar to a database querying operation. The general principle is that of pattern matching between two different graphs. If the pattern of one graph *u* is repeated in some other graph *v* then there is said to be a projection of *u* onto *v*. The projection operation is more flexible than the simple case where *v* contains an exact copy of *u*. The following definition is adapted from Sowa's definition for projection of simple graphs, gives the conditions under which a projection will exist:

Definition 3.4: Projection Of Simple Graphs

For any simple conceptual graphs *u* and *v*, a concept *cu* in *u* will project onto a concept *cv* in *v* if the following conditions are met:

* EITHER the type field of *cv* is a type label which is a subtype of the type label of *cu* OR the type field of *cu* contains the generic marker.

* EITHER the referent of *cu* is identical to the referent of *cv* OR the referent of *cu* is generic.

A relation *ru* will project onto a relation *rv* if:

* EITHER the relational field of *rv* is a relational label which is a subrelation of the relational label of *ru* OR the relational field of *ru* contains the generic marker.

* Each concept attached to *ru* projects onto the corresponding concept in *rv*.

$\pi(u,v)$ is any projection of *u* onto *v* and $\Pi(u,v)$ is the set of all projections of *u* onto *v* called the *projective extent*. The topology of $\pi(u,v)$ must be identical to *u*.

From this definition we see than in every case where there exists a projection of *u* onto *v*, *u* is (by definition) a *generalisation* of *v* and could be generated from *v* by erasure. This follows from: (a) the partial ordering implied by the type and relational hierarchies and (b) from the fact that *u* may be a smaller graph than *v* (*u* may contain fewer relations than *v*) but is never bigger.

The above definition only defines the case where both graphs *u* and *v* are simple graphs which do not contain any modal concepts. We will show that this definition is inadequate when compound graphs or graphs containing modal concepts are considered. We will therefore extend the definition of projection to include all of the additional cases and do so in a way which does not depend upon any accidental characteristics that a particular graph may possess.

3.7.2 Projection Of Compound Graphs

The projection of compound graphs has not been previously defined except by ourselves in [Heaton91]. We will show that the idea of a partial ordering of simple graphs also extends to compound graphs.

3.7.2.1 Projection And The Rules Of Peirce Logic

We saw in Section 3.7.1 that for any simple graph *u* that projects onto a graph *v* we can say that *u* is a generalisation of *v*. In other words, $v \Rightarrow u$. This is not a coincidence and we can take this principle further when formulating the projection of a compound graph *u* onto a compound graph *v*.

The rules of Peirce logic provide the logical basis of an algorithm for projecting a compound graph *u* onto another compound graph *v*. Since we require, by definition, that $v \Rightarrow u$ we must produce an algorithm that tests if *u* could be produced from *v* by these rules. Having said that we are not attempting to produce an algorithm to test whether *u* could be produced from *v* no matter what the topologies of *u* and *v* are. Instead, as in the case of projection of simple graphs, the projection operation compares two graphs to see if they share the same topology. To do more would be to produce a full theorem prover for Pierce logic, which is a completely different matter.

We start by noting that application of the rules of Peirce logic to some graph *g* nested within a compound graph depends only upon whether *g* is evenly enclosed or oddly enclosed. Thus the rules allow us to add information to any oddly enclosed graph or context and to remove information from any evenly enclosed graph or context. We illustrate with the following cases in which we adopt a notational convention that any graph p' is a specialisation of a graph *p* and therefore that $p' \Rightarrow p$:

Case 3.7.2.1.1: Let *u* and *v* be the following compound graphs:

u: (p')
v: (p)

Because we wish to compute $\pi(u,v)$ and we know that for this to succeed *u* must be derivable from *v* we immediately see that the projection succeeds in this case since p' nested in *u* is derivable from *p* nested in *v* by the rule of insertion.

Case 3.7.2.1.2: Let *u* and *v* be the following compound graphs:

u: ($p'\ q$)
v: (p)

As with the previous case we see that *u* can be derived from *v* by insertion to generate *p'* from *p* and by insertion to obtain *q*.

Case 3.7.2.1.3: Let *u* and *v* be the following compound graphs:

u: (*p'* (*q*))
v: (*p* (*q'*))

This case is more complex because of the presence of the nested negated context. Nevertheless, we see that *u* can be derived from *v* by an insertion to generate *p'* from *p* and an erasure to generate *q* from *q'*.

3.7.2.2 Rules For Projection Of Compound Graphs

The examples above illustrate the objectives of this section. We must now state them more formally:

For a compound graph *u* which projects onto a compound graph *v* the following are true:

3.7.2.2.1 Evenly Enclosed Simple Graphs

Since any evenly enclosed graph *g* in *u* is derivable from a corresponding evenly enclosed graph *g'* in *v* by erasure it follows that *g* must project onto *g'*. For any evenly enclosed context *cu* in *u* all nested simple graphs within *cu* must have similarly been derived from corresponding graphs in a corresponding evenly enclosed context *cv* in *v* by erasure. Thus, $\pi(cu,cv)$ is the set of projections, one for each simple graph in *cu*, onto corresponding graphs in *cv*.

Corollary: for any evenly enclosed simple graph in *v* there need not be a corresponding evenly enclosed simple graph in *u* since the rule of erasure may erase entire graphs when deriving *u* from *v*.

3.7.2.2.2 Oddly Enclosed Simple Graphs

Since any oddly enclosed graph *g'* in *u* is derivable from a corresponding oddly enclosed graph *g* in *v* by insertion it follows that *g* must project onto *g'*. For any oddly enclosed context *cu* in *u* all nested simple graphs within *cu* must have similarly been derived from corresponding graphs in a corresponding oddly enclosed context *cv* in *v* by insertion. Thus, $\pi(cu,cv)$ is the set of

projections, one for each simple graph in *cv*, onto corresponding graphs in *cu*.

Corollary: for any oddly enclosed simple graph in *u* there need not be a corresponding oddly enclosed simple graph in *v* since the rule of insertion may insert entire graphs when deriving *u* from *v*.

3.7.2.2.3 Evenly Enclosed Negated Contexts

For any evenly enclosed negated context *c'* in *u*, there must be a corresponding evenly enclosed negated context *c* in *v*. Since the simple graphs in *c'* and *c* are oddly enclosed the simple graphs in *c* must project onto those in *c'*.

Corollary: for any evenly enclosed negated context in *v* there need not be a corresponding evenly enclosed negated context in *u* since the rule of erasure may erase entire negated contexts when deriving *u* from *v*.

3.7.2.2.4 Oddly Enclosed Negated Contexts

For any oddly enclosed negated context *c* in *u*, there must be a corresponding oddly enclosed negated context *c'* in *v*. Since the simple graphs in *c* and *c'* are evenly enclosed the simple graphs in *c* must be projected onto those in *c'*.

Corollary: for any oddly enclosed negated context in *u* there need not be a corresponding oddly enclosed negated context in *v* since the rule of insertion may insert entire negated contexts when deriving *u* from *v*.

3.7.2.2.5 Lines Of Identity

For each line of identity in *u* which terminates in an oddly enclosed context there must be a corresponding line of identity in *v* (since such lines cannot be removed by erasure when deriving *u* from *v*). For each such line of identity in *v* there need not be a corresponding line of identity in *u* but $\pi(u,v)$ will contain such a line of identity.

For each line of identity in *v* which terminates in an evenly enclosed context there must be a corresponding line of identity in *u* (since such lines cannot be added by insertion when deriving *u* from *v*). For each such line of identity in *u* there need not be a corresponding line of identity in *v* but $\pi(u,v)$ will

contain such a line of identity.

We see that the projection operation as described by Sowa and applied to simple graphs is simply a particular case of the above more general rules; that in which the 'compound graph' happens to be a context which contains a single simple graph. Thus not only have we placed the operation of projection of compound graphs onto a firm logical foundation (something not previously done) but we have unified it with the projection of simple graphs. This extension to projection allows a partial ordering of rules to be defined.

3.7.3 Projection Of Modal Concepts

In terms of the basic operation of projection as described up to this point the projection of one modal concept onto another modal concept presents a problem. It is best seen by looking at an example. Consider the following possible representation in some knowledge base of the trivial statement 'John believes that unicorns do not exist':

[PERSON : John]-
(BELIEF)->[PROPOSITION : { ([UNICORN]) }] (9)

This graph states that 'there is a proposition that it is false that there exists a unicorn and that John believes this proposition'. Now it is straightforward to imagine that a query to the knowledge base such as 'is it true that John believes that there are not any unicorns?' would be encoded in exactly the same way and that the query graph would therefore project onto the statement graph. What happens if the query is 'is it true that John believes that there are any unicorns?'? In this case the query graph looks like:

[PERSON : John]-
(BELIEF)->[PROPOSITION : { [UNICORN] }] (10)

and this does not project onto the statement graph since the referent of the modal concept in (10) does not project onto that in (9). The nested graph in (10) does however *evaluate* to FALSE if the nested graph in (10) is *evaluated* against the nested graph in (9) and it is assumed that nested graphs represent nested knowledge bases with the full properties of the global knowledge base. In this case we want the system to reply with something like 'your query is false because I know that John believes that there are not any unicorns'.

It would appear from the arguments above that the projection operation ought to do more than simply signal that a straight match has been found. Indeed it must also return a truth value. For referents other than nested contexts the projection operation already effectively returns either TRUE or NO MATCH (UNKNOWN). We require it to also return FALSE for the case where an evaluation has been made.

Nevertheless, we state that the definition of projection should be more restricted than this since we note that, for non-modal graphs, there is no projection of *u* onto *v* in situations of the following kind:

u: *p*
v: (*p*)

Since the situation described by graphs (9) and (10) is of this form with respect to the referents of the modal concepts we may restrict the projection to an attempt to project the modal referent of (10) onto that of (9). In this case the projection fails. Thus we may define two forms for the projection of modal concepts: strong projection and weak projection. The following definitions state this more formally:

Definition 3.5: Strong Projection Of Modal Concepts

For any modal concept *cu* which is to be projected onto a modal concept *cv*, the projection $\pi(cu,cv)$ succeeds with TRUE if *type(cv)* ≤ *type(cu)* and *referent(cu)* evaluates to TRUE with respect to the modal *referent(cv))* and the projection $\pi(cu,cv)$ succeeds with FALSE if *type(cv)* ≤ *type(cu)* and *referent(cu)* evaluates to FALSE with respect to the modal *referent(cv))*.

Definition 3.6: Weak Projection Of Modal Concepts

For any modal concept *cu* which is to be projected onto a modal concept *cv*, the projection $\pi(cu,cv)$ succeeds if *type(cv)* ≤ *type(cu)* and π(*referent(cu)*,*referent(cv))* succeeds.

We see that the adoption of the weak projection operation brings the definition of the projection of modal concepts into line with the previous definitions and simplifies the definition of projection. This is so since we have made the projection of the referents of modal concepts to be of exactly the same form as for the cases of simple and compound graphs. We see that for a modal referent *ru* to project onto a modal referent *rv* then *ru* must be derivable from *rv*. This is exactly analogous to the situation with simple graphs and compound graphs. Nevertheless, the strong projection operation makes possible the full drawing of conclusions from theories and other nested models. This will be important in the future for proving more complex systems than is possible otherwise.

The rule of projection allows one graph *u* to be overlaid onto another topologically equivalent graph *v*. This overlaying proceeds if the objects in *v* are not less specific than the corresponding objects in *u*. Thus the graph *v* is a model of the graph *u*. This being the case the projection of *u* onto *v* proceeds by a series of attempts to specialise each type label, referent and relation in *u* to be the same as the corresponding object in *v* and at the same time preserve the topology. Since the operations involved are all present in Peirce logic we may state that projection is therefore a derived rule of inference.

Although projection is a derived rule of inference its use within any working system is likely to be extensive and there are therefore sound computational reasons for retaining it as a specially programmed operation.

3.8 CONCLUSIONS OF CHAPTER 3

Each of the major operations on conceptual graphs has been considered in detail and certain important deficiencies and errors have been corrected. The rules of Peirce logic with respect to conceptual graphs have been clarified and extended to cover the all aspects of the representational theory that we developed in Chapter 1.

The review of CG operations has shown that several operations can be performed in more than one way and that where this is the case it is suggested that only one of these ways is necessary for an implementation except for where efficiency is a prime consideration.

As a result of reducing the components of the system it has been possible to reduce the number and complexity of operations to a level that allows a tractable computational model to be built without losing any of the power of Conceptual Graphs.

CHAPTER 4

COMPUTATION WITH PEIRCE LOGIC

4.1 INTRODUCTION

In this chapter we describe how the operations that were discussed in Chapter 3 can be assembled into a reasoning system which is capable of attaining the following objectives:

* Testing its inputs for selectional constraint violations.

* Testing its inputs for logical self-consistency and consistency with the current knowledge base.

* Adding new consistent knowledge to its knowledge base and rejecting inconsistent statements.

* Answering arbitrary queries of its knowledge base as fully and automatically as possible.

There are many possibilities that are not covered by this short list. Examples are: how to deal with alternative models of the world, how to deal with uncertainty, how to deal with the correction of incorrect knowledge. The work presented here is related to what can be seen to be the first stage of a much larger project. This initial stage is concerned with building the basic mechanisms by which one particular model is created, maintained and queried. Once this first stage is complete it will be possible in the future to generalise the approach to cover the more advanced forms of reasoning, but all such advanced forms depend ultimately upon the basic logical mechanisms.

4.2 FUNDAMENTAL CONCEPTS

4.2.1 Objectives

In order to be able to meet the objectives listed above the system must be logically complete. If it were not then the user would need to add knowledge that no user should have to be concerned about since such a system would resemble a programming language such as Prolog. Since we wish to produce a system that is fully automated, in other words the system is fully declarative and $\models \Rightarrow \vdash$, we must develop a methodology that does not place restrictions upon what can be computed. This is in contradiction of the normal approach of restricting the logic so that proofs are computationally efficient but incomplete. Thus, Prolog for instance cannot show that the clause:

```
pred :-
        pred.
```

is a tautology, nor can it prove that the clause:

```
pred :-
        not pred.
```

is a contradiction. Many expert system shells suffer similar deficiencies since they are actually very similar to Prolog. Prolog is even more limited since its semantics do not allow the denial of a fact by the assertion of a clause of the form:

```
not pred.
```

The only way to do this is to state that:

```
pred :-
        !,
        fail.
```

This will certainly prevent pred from being true (except that pred can be independently asserted). Nevertheless, this clause will not allow the clause:

```
pred :-
        pred_1.
```

to be used to prove that pred_1 must be false.

Thus we must provide a system that can perform all these functions so that a user may add arbitrary information with confidence. In order to do this we will need to develop a novel approach to computation with conceptual graphs and Peirce logic. The foundations for this have already been laid in the reconstruction which was the subject matter of Chapters 2 and 3.

Ideally, we wish to construct a system which will subsume current database systems, relational databases, deductive databases and expert system shells. At the same time we wish to provide complete mechanisms for the checking of knowledge base consistency. The checking of consistency will be made up of two aspects: checking for selectional constraint violations and checking of logical consistency.

For the checking of selectional constraint violations we need to provide a simple mechanism for comparing new assertions and queries against a *canonical basis*. Since the graphs in the canonical basis form a set of graphs of which all other graphs must be specialisations, it is necessary to test all new graphs for the existence of this property with respect to the canonical basis. A procedure for this must therefore be developed.

For the checking of logical consistency of the knowledge base, and indeed any queries that are made, we need a general purpose theorem prover. This tool will provide us with a very powerful means of verifying the truth of results of many forms of reasoning.

At the more advanced levels we note that many, if not all, forms of reasoning depend upon the existence of some kind of model and model generating mechanism. Thus, when reasoning with alternative models, as when attempting to generate explanations or predictions, we may wish to generate theories and test these, in the first instance, with the theorem prover for logical consistency, since if such a test fails then the model is false. In this way the system will simply be applying itself to some particular submodel. Similarly, reasoning with uncertain knowledge will involve the generation of alternative

models and the verification of these models will be carried out with the same theorem prover. Similar arguments can be offered for the need to use a theorem prover at some stage in reasoning of other forms.

From these informal arguments we conclude that the first stage in the development of a sophisticated reasoning system is the provision of a theorem prover.

4.2.2 Knowledge Base Structure

The knowledge based system that we will develop will consist of two main sections. These are a *knowledge base* and a *canonical basis*.

Knowledge Base

Conceptual graphs theory has its logical roots in the formalism of Peirce. As such the structure and semantics of the knowledge base is largely determined. In particular, all true statements that correspond to factual information are placed on the Peircean *sheet of assertion*. In addition, we will include a type lattice, relational lattice and conformity relations table in the knowledge base. Such information is already implicit within the graphs which make up the knowledge base and these additional structures merely make it more accessible.

Canonical Basis

Additionally, we assume the existence of a canonical basis as defined in Chapter 2. There are two ways in which to view this set of graphs:

* All canonical graphs are common generalisations of known true facts and are in a dynamic state.

* All canonical graphs are placed within the system by the knowledge engineer and are thus static.

Furthermore, the semantics of the generic referent prevent us from simply adding common generalisations, however derived, to the sheet of assertion. In any case, to do so would create vast quantities of additional graphs without adding any information. The canonical model must therefore be seen as a separate entity to the sheet of assertion.

4.2.3 Model Theoretic Considerations

We adopt the *open world* model theoretic semantics of Rescher and Brandom [Rescher79]. These were adopted by Sowa as the model theoretic basis of conceptual graphs. The reasons for their selection by Sowa are:

* Inconsistencies may exist within a knowledge base without necessarily making all things provable,

* Greater flexibility than the models of Prolog and rule based expert systems,

* Provides a three valued logic with truth values of TRUE, FALSE and UNKNOWN.

We adopt the following 'strong' truth tables [Kleene38] [Kleene62]:

		$\overline{Q}$	$Q \vee R$			$Q \wedge R$			$Q \rightarrow R$		
			R	t	f u	R	t	f u	R	t	f u
Q	t	f		t	t t		t	f u		t	f u
	f	t		t	f u		f	f f		t	t t
	u	u		t	u u		u	f u		t	u u

Since we only employ negation and conjunction we only require the truth tables for these. Nevertheless, it is worth pointing out that there are other interpretations of UNKNOWN. That of Łukasiewicz [Łukasiewicz20], the 'weak' truth table, defines $(Q \rightarrow R) \rightarrow t$ where Q and R are both UNKNOWN. However, Peirce's rules cannot generate an arbitrary implication of the form:

$$(\,Q\,(\,R\,)\,)$$

where R is UNKNOWN and $R \neq Q$ since the only means of placing R into the evenly enclosed context is by iteration. Such iteration may be possible as follows:

Double negation:	$(\,(\,)\,)$
Insertion:	$(\,Q\,(\,)\,)$
Insertion:	$(\,Q\,R\,(\,)\,)$
Iteration:	$(\,Q\,R\,(\,R\,)\,)$

but the problem of how to remove the oddly enclosed *R* remains.

UNKNOWN has an *apparent* effect on the truth tables. One case is where $R \neq (Q)$ and the strong truth table gives the truth assignments as shown. The other case is where it happens that $R = (Q)$. We now have $Q \wedge R \rightarrow f$ (at * in the table below) when *Q* and *R* are UNKNOWN since now we actually have $Q \wedge \bar{Q}$. The strong truth tables can be seen to imply this.

		$Q \vee \bar{Q}$			$Q \wedge \bar{Q}$			$Q \rightarrow \bar{Q}$		
		$\bar{Q}$	*t*	*f* *u*	$\bar{Q}$	*t*	*f* *u*	$\bar{Q}$	*t*	*f* *u*
Q	*t*		-	*t* -		-	*f* -		-	*f* -
	f		*t*	- -		*f*	- -		*t*	- -
	u		-	- *u*		-	- *f**		-	- *u*

where - signifies impossible conditions, *e.g. Q* cannot be TRUE and FALSE.

The Rescher and Brandom semantics allows the division of the truth value UNKNOWN into two forms: UNKNOWN because there is not any information at all and UNKNOWN because a proposition has models in both the TRUE and FALSE sets. This division allows us to detect inconsistencies.

Assumption 4.1: Open World Model

An open world model consists of:

T - the set of true graphs.
F - the set of false graphs.
I - the set of individual markers.

Each individual marker *i* in *I* must occur only once within *T* but may occur any number of times within *F*. In an open world, a graph *g* may evaluate to one of three truth values: true, false or unknown.

Corollary

The set T is all the simple graphs enclosed at depth 0 and the set F is all the compound graphs.

Since each individual marker *i* in *I* can only occur once in *T* it follows that all graphs in *T* that contain concepts which contain *i* must be joined on those

concepts. Additionally, it also follows that all concepts in T on a line of identity must be joined.

The second part of this corollary not only follows from the model theoretic assumptions and thus ought to be adopted on purely theoretical grounds but also has important consequences for efficiency of implementations. These will be discussed fully in Section 4.4.3.

4.2.4 Proof Strategies

Having described the representation formalism, operations and semantics that we wish to adopt we must now discuss proof strategies. We will start by reviewing simple proof strategies and the proof strategies that Sowa introduces. We will then go on to consider deductive mechanisms and theorem proving techniques. Finally we will extend the discussion to areas where the conceptual graphs theory is less developed and thereby give indications as to what future work must be undertaken.

The aim of finding appropriate proof strategies is that of providing a computational mechanism for conceptual graphs that subsumes those for other formalisms and methodologies. Ideally it should be demonstrable that conceptual graphs are capable of somewhat more than other formalisms otherwise there is not much point in using them. In the following sections we argue that Peirce logic provides some alternative strategies that have not previously been used in the form to be presented.

The strategy that we finally adopt will be developed at a high level, with scope for future improvement, especially from the points of view of efficiency and heuristics which are beyond the scope of the present work. However, fundamental problems of efficiency with respect to combinatorial explosion are considered since the aim of the work is to show that a usable system is possible.

4.2.5 Types Of Proof

This section discusses the identification, already hinted at, of two fundamental aspects to the proof of any formula. These are:

* Proof of simple graphs.

* Proof of compound graphs.

4.2.5.1 Proof Of Simple Graphs

We recall that a simple graph is any single connected conceptual graph, or the negation of such a graph. From this it follows that simple graphs can be proved either by a process of direct lookup in the knowledge base or by deduction from rules.

In proving a simple graph *g* we are attempting to show that all relations in *g* have a model in the knowledge base. Any such graph may consist of several relations. As a result, some of the relations in *g* may be represented explicitly in the knowledge base and some may be implicit. Thus it is necessary to split *g* into its component relations and prove each separately. When this is done the problem of proving *g* is reduced to one of proving single relations (and single concepts if *g* is a single concept).

4.2.5.2 Proof Of Compound Graphs

A compound graph *g* may be proved by the lookup mechanism but in the majority of cases it must be shown that *g* is implied by the knowledge base. This involves either showing that *g* can be generated from graphs in the knowledge base or that *g* is intrinsically tautological according to the rules of inference of Peirce logic. For this testing we require a theorem prover to show the validity of the logical connectives between the simple graphs that *g* contains. Any individual simple graphs within a compound graph can be proved as necessary by the lookup and deductive mechanisms.

4.2.5.3 Factorisation Of Proof Strategies

By differentiating between the proofs of simple graphs and compound graphs we have factored out two main approaches. The approach to the proof of simple graphs bears a close resemblance to existing strategies employed in expert systems. Nevertheless, the system that we will develop will be shown to be much more flexible than those in current use. The proof of compound graphs requires a radically new approach to Peirce logic which we will describe later.

Although we propose a factoring of proof strategies we should state that such factoring is not necessary for logical reasons since the proof of simple graphs by 'pure' Peirce logic is perfectly possible. Nevertheless, most if not all existing knowledge processing existing of all kinds do not employ a purely logical approach since it is flawed for pragmatic reasons. Instead, a dedicated deduction generating mechanism is employed. Such a mechanism can be optimised to overcome the otherwise fatal problems of combinatorial explosion.

Since we wish our procedure for the proof of compound graphs to be tractable we must also investigate a fundamentally new approach. Not only that but we must also ensure that the two basic proof mechanisms, that for simple graphs and that for compound graphs, together provide a total proof mechanism that is complete.

4.3 SELECTIONAL CONSTRAINTS

Selectional constraints are constraints on graphs which define semantically acceptable forms and do not allow the combination of concepts and relations in meaningless ways. There are two ways in which selectional constraints may be enforced. These are the logical method and the specialisation method.

4.3.1 Logical Enforcement Of Selectional Constraints

This method relies upon the specification of a rule which states the most general types that can be attached to the concepts of a relation. As an example we will make use of the following:

```
( [ T : *x ]->( WIFE )->[ T : *y ]
        ( [ MAN : *x ] [ WOMAN : *y ] ) ).
```

This rule states that for all x and all y, if x has a wife y then x is a man and y is a woman. Thus we can attempt to use this rule to deny that some cat has a wife which is some dog:

```
[ CAT ]->( WIFE )->[ DOG ]
```

We can see from the rule that this graph is false if it is also false that a cat is a

man and a dog is a woman. There are two ways in which this denial may be specified.

Rule Based Denial

This is achieved by the introduction of the two rules:

([MAN : *x] [CAT : *x])

([WOMAN : *y] [DOG : *y])

This suffers from two problems, one of which is very serious. The less serious problem is that it is necessary to find some mechanism to make it work. In reality this is not a problem since we will be describing such mechanisms shortly. The second, more serious problem is that it may be necessary to define any number of rules to deny that any other entity is also a man or a woman. Since there may be a vast number of such rules for each of a vast number of relations this approach is not practical.

Lattice Based Denial

It may be the case that a cat cannot be a man simply because both types are natural types and natural types do not have a common subtype. If this conjecture is true then we can dispense with the rules that are required in the rule based approach. This would simplify the whole procedure. Much work on defeasible inheritance has been done and yet this work seems to complicate the picture in a counterintuitive way since it allows exceptions to 'universally' quantified statements. Nevertheless, the simple denial that two natural types can ever possess a common subtype would certainly allow the rejection of ill-formed graphs provided a suitable mechanism was found.

The logical approach to checking of selectional constraints, even when optimised by the use of defeasible inheritance, still requires that rules be found and fired. This is computationally expensive in comparison to the method to be described next.

4.3.2 Specialisation Enforcement Of Selectional Constraints

A canonical basis is the set of graphs from which all other graphs in a particular knowledge base can be derived by use of the canonical formation

rules. We previously defined a canonical model of a relation *r* as a generalisation of all semantically valid cases of *r*. This tells us that all well formed graphs must be specialisations of graphs within the canonical basis that contain the same relations. As a result we make the following Assumption:

Assumption 4.2: Semantically Valid Relations

If a relation *r* is well formed then there is a graph in the canonical basis with the same relational label as *r* and which projects onto *r*.

This Assumption is valid if defeasible inheritance is denied [Fetzer90]. Since the formation rules will generate canonical graphs by restriction of type labels it follows that the original canonical model will project onto the new graph. Thus for each relation in any formula, that relation is well formed if the result given in the Assumption applies. We make the following additional points:

* All simple graphs at depth 0 must be shown to be well formed since they are meant to represent true atomic facts. Indeed all evenly enclosed graphs at other depths must be shown to be well formed since they may be 'released' at depth 0 as the result of a deduction at some time in the future.

* Simple graphs within oddly enclosed contexts need not be well formed as they can never be made to appear at depth 0 (or any other evenly enclosed depth since to do so would require the removal of a single negation) unless there is a dominating copy of the graph at an evenly enclosed depth anyway. In such circumstances this will have been tested for selectional constraint violations and must therefore be well formed. Thus, all compound graphs are well formed if all their evenly enclosed simple graphs are well formed.

The second point requires further illustration. We take the case of the following formula:

([T1]->(R)->[T2] (*p*))

where *p* is any arbitrary well-formed graph that also occurs at depth 0. The following is a proof that the formula is a theorem:

Proper axiom:	*p*
Double negation:	**((*p*))**
Insertion:	([T1]->(R)->[T2] (*p*))

Thus the formula is a theorem. If we say that oddly enclosed graphs must be well-formed we note that the above proof could not be constructed if R is not well formed. Indeed the following kind of statement could not be made:

([MAN]->(SPOUSE)->[MAN])

The result of these two observations is that any mechanism for testing for selectional constraint violations only need be applied to the evenly enclosed simple graphs in any compound graph.

Testing For Selectional Constraint Violations

The Assumption stated that for each relation *r* in a well formed graph there will be a graph *g* in the canonical basis with the same relational label as *r* such that *g* will project onto *r*. Thus the test for canonicity involves taking each evenly enclosed relation in turn and attempting to project onto it the canonical model for that relation. Should any such projection fail then there is a violation of selectional constraints otherwise there is no such violation.

Since we know exactly which canonical model to use for each relation the search space for the correct model is minimal, especially if the canonical basis is suitably indexed. Also, since we do not fire rules and chains of deduction as in the logical method of testing canonicity we do not incur the time penalties of that approach.

Unknown Type Labels And Relational Labels

It may be the case that a formula contains a type label or a relational label which the system has never previously encountered. In this situation it is

necessary to adopt either of two policies:

* Assume that all unknown type labels and relational labels are errors and reject such formulae.

* Assume that all such labels are simply unknown and attempt to place them in the correct place within the appropriate lattice, adding canonical models to the canonical basis as necessary.

Assertions And Queries

There is a difference between assertions and queries. All assertions that are to be added to a knowledge base must be semantically correct. Queries, however, may be as abstract as required and this means that it must be possible to submit a query to the system even though the concepts attached to a particular relation are strictly more general than those in the canonical model. For instance we may know that:

[MAN : John]->(WIFE)->[WOMAN : Elizabeth]

In this case it would be unreasonable to reject a query such as:

[PERSON]->(WIFE)->[PERSON]

However, it would not be unreasonable to reject this as an assertion since it is ambiguous. Nor would it be unreasonable to reject the following as a query:

[ANIMAL]->(WIFE)->[EVENT]

The difference between these two queries is that the first projects onto the canonical model for WIFE, and could therefore support valid specialisations, whereas the second does not. Thus, a query is canonical if all concepts attached to the evenly enclosed relations are on the same line of inheritance as the corresponding concepts in the canonical model.

4.3.3 Additional Tests

The test for selectional constraint violations as described above are entirely satisfactory when the concept employs only individual or generic referents. We also specified that the referent of a concept of type PROPOSITION was a context and that concepts of type NUMBER held a

numeric referent. It is therefore also necessary to ensure that the correct referent type is present in each evenly enclosed concept. It is a simple matter to check this.

4.4 LOOKUP

The simplest form that the proof of any formula can take is that of *lookup*. If the formula can be found within the knowledge base then it is certainly true with respect to that knowledge base and the query has been answered. In other words, if the knowledge base is a model of the query then the query is satisfied.

The conceptual graphs formalism allows the lookup of abstract graphs *via* the projection operation. With this operation the knowledge base can be exhaustively searched. In general a query may have several solutions since any relatively general graph (one containing type labels which are high in the type hierarchy or which contains generic referents) may potentially match several cases in the knowledge base.

It is important to provide a theoretically sound mechanism for the process of lookup. We now discuss the above points in terms of the model theoretic semantics that Sowa provides for conceptual graphs and also the restriction that lookup only be used for simple graphs and negated simple graphs.

4.4.1 Lookup And The Open World

With an open world the problem of lookup is complicated slightly by the possible presence of negated graphs. Any query formula may be either a simple graph or a negated simple graph. Any graph in the knowledge base may also be either a simple graph or a negated simple graph. As a result, there are four possibilities to consider:

I The query is a simple graph and the solution is a simple graph.

II The query is a simple graph and the solution is a negated simple graph.

III The query is a negated simple graph and the solution is a simple graph.

IV The query is a negated simple graph and the solution is a negated simple graph.

Cases I and IV must give the result TRUE and cases II and III must give the result FALSE (if any query has solutions in more than two cases then the knowledge base is inconsistent). Each case will now be examined in turn with examples. The naming convention that a label such as SUBT or SUBR represents a subtype of T or R will be adopted.

Case I

In this case a solution is found when the query graph is projected onto a graph in the knowledge base. Each result of the projection is a solution. Thus the query:

[T1]->(R)->[T2]

is satisfied by the following graph within the knowledge base:

[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]

The projective extent of a query is the set of all such projections.

Case II

In this case a solution is found by projecting negated simple graphs from the knowledge base onto the query. The projection is accomplished by first extracting the simple graph from its negative context. If any such projection is found then the query is FALSE. Thus the query:

[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]

is satisfied by the following graph within the knowledge base:

([T1]->(R)->[T2])

This simple graph nested within the negative context is projected onto the query. The projective extent of a query is the set of all such projections.

Case III

In this case the solution is found by extracting the simple graph from the negative context in the query and projecting it onto the simple graphs from the knowledge base. If any projection is found the query is FALSE. Thus the query:

([T1]->(R)->[T2])

is satisfied by the following graph within the knowledge base:

[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]

This simple graph nested within the negative context is projected onto the graph in the knowledge base. The projective extent of a query is the set of all such projections.

Case IV

In this case the solution is found by projecting the query onto negated simple graphs within the knowledge base. If any such projection is found the query is TRUE. Thus the query:

([SUBT1 : #x]->(SUBR)->[SUBT2 : #y])

is satisfied by the following graph within the knowledge base:

([T1]->(R)->[T2])

The projective extent of a query is the set of all such projections.

Direction Of Projection

Cases II and III both state that a graph is projected onto the query. The reason for this is the fact that and false graph *g* in a knowledge base is a generalisation of a hierarchy of false graphs, all specialisations of *g*. Thus for *g* to prove a query false the query must be a specialisation of *g* and therefore *g* will project onto the query.

4.4.2 The Evaluation Game

These four cases are subsumed by a single, well founded procedure, the

open world evaluation game. To see how this is the case we will review the evaluation game and then examine each case in that context.

Evaluation Game

As mentioned previously we adopt the semantics of Rescher and Brandom which allow us to reduce the effects of an inconsistent knowledge base by detecting such situations and computing denotations accordingly. We summarise the moves of the game in terms of notation for negation that we developed in Chapter 2:

Definition 4.3: Evaluation Game

An *evaluation game G* on an open world $W = \langle T, I \rangle$ is a two-person, zero-sum, perfect information game between two players called *proposer* and *sceptic*. It is defined by the following rules:

* The positions of G are triples $\langle p,t,s \rangle$ where the player on move p is one of {proposer,sceptic}, t is one of three *move types* in the set {project, select, reduce}, and s is any set of conceptual graphs.

* The starting position of G is a triple $\langle$proposer,project,$s\rangle$ for any set s of conceptual graphs.

* From a position $P = \langle p_1,\text{project},\{u_1,...,u_n\}\rangle$, the successors are all positions of the form $\langle p_2,\text{select},\{v_1,...,v_n\}\rangle$ where p_2 is the opponent of p_1 and each v_i is a version of u_i modified by the following algorithm:

FOR each simple graph u_i **IN** P **LOOP**
IF $\Pi(u_i,T)$ is nonempty **THEN**
choose any projection π in $\Pi(u_i,T)$;
FOR each concept c in u_i **LOOP**
copy *referent*(πc) to the referent field of c and
all concepts dominated by c;
erase any newly added referent in a concept

that is identical to a referent previously present;

insert "=" between the referents of concepts that have more than one referent;

erase all coreference links dominated by *c*;

ENDLOOP;

ENDIF;

ENDLOOP;

* From a position $P = \langle p_1, \text{select}, \{u_1, \ldots, u_n\} \rangle$, the successors are the n positions of the form $\langle p_2, \text{reduce}, \{u_i\} \rangle$ where p_2 is the opponent of p_1 and i is any integer from 1 to n. If n=0, P is an ending position and p_1 loses.

* From a position $P = \langle p_1, \text{reduce}, \{u\} \rangle$ where p_2 is the opponent of p_1,

IF *u* is a non-negated simple graph **THEN**

P is an ending position;

CASE value of $\langle \Pi(u,T), \Pi(F,\tilde{}u) \rangle$ **IS**

WHEN ⟨empty, empty⟩ => game is drawn;

WHEN ⟨nonempty,empty⟩ => p_1 wins;

WHEN ⟨empty,nonempty⟩ => p_2 wins;

WHEN ⟨nonempty,nonempty⟩ => game is drawn;

ENDCASE;

ELSE

u must be a negated context;

s := the set of graphs enclosed in *u*;

the successor to *P* is $\langle p_2$, project,*s*⟩;

ENDIF;

This Definition is similar to that given by Sowa with the following exceptions:

* The algorithm for the reduce move refers to negated contexts and graphs enclosed within them. This small difference reflects our alternative view of the notation and semantics of negation.

* The algorithm for the reduce move is reworded to reflect our definition of the term *simple graph*. In accordance with this it is also necessary to specify that the operation $\Pi(F,\tilde{}u)$ is carried out with u negated since any graph in F which may deny u will be a graph of the form $\tilde{}u'$.

* Both the project move and reduce move calculate the projective extent of some graph. Since this may be large and only one projection is ever required at any one time it is assumed that the projective extent operation finds projections one at a time.

* Sowa's version prevented concepts with more than one referent from generating a projective extent, with the restriction: "Some graphs in any position may have concepts with two or more distinct referents separated by equals signs. The projective extent in W of any such graph must be empty, since no concept in any atomic fact may contain multiple referents.". We remove this restriction since we argued (section 2.4.1.10) that it is reasonable to equate individual markers.

We now show how these rules perform the function of lookup in the four cases identified earlier.

Case I

Knowledge base:

[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]

Query:

[T1]->(R)->[T2]

Game:

P_1 = ⟨proposer,project,{[T1]->(R)->[T2]}⟩

P_2 = ⟨sceptic,select,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_3 = ⟨proposer,reduce,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

Since the graph in P_3 is a simple graph u this is an ending position such that:

$$\Pi(u,T) = \text{nonempty}$$
$$\Pi(F,\tilde{}u) = \text{empty}$$

therefore the current player, proposer, wins the game and the query is TRUE.

Case II

Knowledge base:

```
( [ T1 ]->( R )->[ T2 ] )
```

Query:

```
[ SUBT1 : #x ]->( SUBR )->[ SUBT2 : #y ]
```

Game:

P_1 = ⟨proposer,project,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_2 = ⟨sceptic,select,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_3 = ⟨proposer,reduce,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

Since the graph in P_3 is a simple graph u this is an ending position such that:

$$\Pi(u,T) = \text{empty}$$
$$\Pi(F,\tilde{}u) = \text{nonempty}$$

therefore the current player, proposer, loses the game and the query is FALSE.

Case III

Knowledge base:

```
[ SUBT1 : #x ]->( SUBR )->[ SUBT2 : #y ]
```

Query:

```
( [ T1 ]->( R )->[ T2 ] )
```

Game:

P_1 = ⟨proposer,project,{([T1]->(R)->[T2])}⟩

P_2 = ⟨sceptic,select,{([T1]->(R)->[T2])}⟩

P_3 = ⟨proposer,reduce,{([T1]->(R)->[T2])}⟩

P_4 = ⟨sceptic,project,{[T1]->(R)->[T2]}⟩

P_5 = ⟨proposer,select,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_6 = ⟨sceptic,reduce,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

Since the graph in P_6 is a simple graph u this is an ending position such that:

$\Pi(u,T)$ = nonempty
$\Pi(F,\tilde{}u)$ = empty

therefore the current player, sceptic, wins the game and the query is FALSE.

Case IV

Knowledge base:

([T1]->(R)->[T2])

Query:

([SUBT1 : #x]->(SUBR)->[SUBT2 : #y])

Game:

P_1 = ⟨proposer,project,{([SUBT1 : #x]->(SUBR)->[SUBT2 : #y])}⟩

P_2 = ⟨sceptic,select,{([SUBT1 : #x]->(SUBR)->[SUBT2 : #y])}⟩

P_3 = ⟨proposer,reduce,{([SUBT1 : #x]->(SUBR)->[SUBT2 : #y])}⟩

P_4 = ⟨sceptic,project,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_5 = ⟨proposer,select,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

P_6 = ⟨sceptic,reduce,{[SUBT1 : #x]->(SUBR)->[SUBT2 : #y]}⟩

Since the graph in P_6 is a simple graph u this is an ending position such that:

$\Pi(u,T)$ = empty

$\Pi(F,\tilde{}u)$ = nonempty

therefore the current player, sceptic, loses the game and the query is TRUE.

4.4.3 Lookup And Knowledge Base Structure

We argued in Section 4.2.3 that the open world assumptions suggest that all graphs at depth 0 (the graphs in the set *T)* should be joined. We will see when we consider the deductive mechanism that this also applies to graphs nested at any depth within any one context.

To see why this requirement is more than a theoretical detail we consider the following simple example:

Given a knowledge base with the following graphs:

[T1 : #1]->(R1)->[T2 : #2] [T2 : #2]->(R2)->[T3 : #3]

the graph:

[T1 : #1]->(R1)->[T2 : #2]->(R2)->[T3 : #3]

cannot be proved directly by projection onto the knowledge base. We also observe that the type label in each concept is required to be the proper type label of the individual referent. If this were not the case then the following knowledge base:

[T1 : #1]->(R1)->[T4 : #2]->(R2)->[T3 : #3]

T2 :: #2

T2 < T4

would not easily satisfy formula:

[T1 : #1]->(R1)->[T2 : #2]->(R2)->[T3 : #3]

by projection using the standard operation.

4.4.4 Lookup Heuristics

As described above, the evaluation game does not contain any means of optimisation with respect to a working system. There are two ways in which such optimisation can be achieved.

Indexing Of Type Labels And Relational Labels

A knowledge base would be arranged in such a way that graphs were indexed. There are two ways of achieving this:

* Indexing of relational labels.

* Indexing of type labels.

In each case where labels are indexed the graph retrieval mechanism would be able to locate in the knowledge base all occurrences of any type or relation (or any subtype or subrelation) in a query. All other types and relations would be ignored. Where the lattice was shallow and bushy the time savings would be considerable. There has been a certain amount of work into efficient graph retrieval carried out [Ellis93] [Ellis91]. However, the problems are not completely solved. In particular, many such schemes involve the generation of large hierarchies of all generalisations of graphs in the knowledge base. This generalisation hierarchy is then searched in a top down manner. Whilst the retrieval may itself be efficient the number of possible generalisations, and hence the size of the generalisation hierarchy, is vast even for a very modest knowledge base. This precludes the use of such a hierarchy in a practical system.

Indexing Of Referents

The open world assumption requires the joining of graphs on concepts sharing identical referents. The corollary to this from the present point of view is that any concept in a query which contains a referent which is also present in the knowledge base can only occur in one graph in the knowledge base. Thus, an indexing mechanism that allowed access to concepts within the knowledge base on the basis of a referent indexing scheme would allow direct access to the single graph that contained that referent. Once that graph had been found

an indexing mechanism for relational labels would then be used to select appropriate relations.

4.5 DEDUCTION

The need for a deductive mechanism arises because our system is not *deductively closed* and we will require access to all knowledge that the knowledge base implies. In order to supply that knowledge we require a deductive mechanism to be available on demand. Such deductive mechanisms already exists in expert systems and in Prolog but these are not usually logically complete and require the user to add further reasoning powers by writing programs. We wish to remove all necessity to write programs and so our deductive mechanism must be sufficient to generate all simple graphs that the knowledge base entails.

4.5.1 Typical Deductive Mechanisms

Many deductive systems that are found in expert systems, deductive databases and Prolog are capable of performing simple forms of *modus ponens*. *Modus ponens* states:

$$(p \wedge (p \Rightarrow q)) \Rightarrow q$$

For a closed world this is adequate; there are not any false facts as such, only unknown (and therefore 'false') facts. However, we have adopted the open world assumption and we now find that *modus ponens* as applied to these deductive systems is not adequate. In particular, the adoption of specialised clausal forms such as the restricted Horn clauses found in Prolog and specialised algorithms to operate on these means that derivations such as the following are not possible:

$$(\neg q \wedge (p \Rightarrow q)) \Rightarrow \neg p$$

This derivation is an example of *modus tollens* and ought to be straightforward in a general purpose system. The reasons why this is not possible in Prolog and many expert systems are twofold:

* These systems make a distinction between the head of a clause (the *consequent*) and the body of a clause (the *antecedents*).

This distinction is made for sound implementational and efficiency reasons which are valid in the context of their use in programming languages but it means that any predicate which is to be proved can only ever be unified with the 'head' of the clause. In the example above we might wish to ask the system if p is true. Unfortunately there is not a clause in the knowledge base which contains $\neg p$ as its consequent and therefore a system that works in this way will not be able to show that p is false.

* The closed world assumption adopted by such systems prevents the proposition $\neg q$ from being added to the knowledge base.

4.5.2 Deduction And Peirce Logic

When we examine deduction in Peirce logic we see a different picture. The rule of *modus ponens* is expressed as:

$$(p (p (q)) (q))$$

If the 'antecedent' of this is expressed in the way in which it would be in a knowledge base (as a fact and a rule) we get:

$$p (p (q))$$

To prove the formula q we simply select the rule which contains q, that is $(p (q))$, and deiterate the 'antecedent'. This leaves:

$$((q))$$

which gives us q when we remove the double negation. The rule of *modus tollens* is expressed as:

$$((q) (p (q)) ((p)))$$

Again, if the 'antecedent' of this is expressed in the way in which it would be in a knowledge base (as a fact and a rule) we get:

$$(q) (p (q))$$

In this case, to prove the formula *p* we simply select the rule which contains *p*, that is (*p* (*q*)), and again deiterate the 'antecedent'. This time the antecedent is a different graph; it is (*q*), which can be deiterated because of its occurrence outside the rule. The result in this case is the graph (*p*) which is sufficient to show the formula *p* to be false.

This example has shown that the distinction between 'head' and 'body' of clauses can be removed and, when clauses of the form ~*q* are allowed, it is a simple matter to provide extensions of a deductive mechanism for an open world. It will be shown in the next chapter that these extensions do not result in gross inefficiencies in computational systems. We can therefore generalise the rules of *modus ponens* and *modus tollens* by stating that, given any rule and an unrestricted mechanism for proving its 'antecedents', whatever residue remains was previously unknown. This residue is the result of the deduction.

Since conceptual graphs are a sorted logic the selection of rules to be used at each stage of a deduction is not a simple matter of matching goals with 'consequents' on a 1:1 basis. In general, any goal may be more general than the consequent where the goal is going to be true and may be more specific where it is going to be false. As an example we may have the following trivial knowledge base:

[ELEPHANT : Clyde]

([ELEPHANT : *x] ([MAMMAL : *x]))

MAMMAL < ANIMAL

CAT < MAMMAL

In this case the query:

[ANIMAL : Clyde]

may be proved deductively by unifying the goal with the 'consequent' [MAMMAL : *x] since the rule above can generate [MAMMAL : Clyde] which certainly implies the goal. Thus the matching of goals to rules must be performed by a more flexible operation than simple graph comparison. In order to combine the relevant details in the goal with the 'head' of the rule we may

consider that we can use the maximal join, as Sowa suggests. However, this is not correct since we must be aware of the following type of case:

Query: [CAT : Clyde]

Rule: ([ELEPHANT : *x] ([MAMMAL : *x]))

Query unified with rule:

([ELEPHANT : Clyde] ([CAT : Clyde]))

In other words, we have specialised, as a result of the maximal join, the evenly enclosed 'head' of the rule to a more specialised level than we are allowed since the type label MAMMAL cannot be specialised by the rules of Peirce logic (the *x can be specialised to Clyde by insertion into the oddly enclosed context and then iteration). Thus the normal maximal join, which would allow this spurious unification, is unsound for this purpose. We must restrict unifications by disallowing the result to contain any specialisations of fields which are not joined by lines of identity to dominating oddly enclosed graphs, and those that are must not be specialised beyond the point at which specialisations cease to be supertypes of the type of the dominating concept. Additionally, we must also ensure that the query graph is only unified with a head that is at least the same size and topology as the query since we may otherwise make the mistake of assuming that the successful deduction of an otherwise enlarged head was logically valid. It is not since we would be adding information to the head by insertion, which is not allowed in evenly enclosed contexts. We must now state that two graphs can be unified if a maximal join does not violate these conditions. The result of the unification is the maximal join of the two graphs.

Deduction does not end with the mechanism of 'firing' of single rules or of chaining of rules. A graph *p* may be proved in the absence of known simple facts on which to ground the deduction. The following example illustrates the point:

Given: (*p* (*q*)) (*p q*)

We need to know if the proposition *p* is true:

Iteration: (*p* (*q* (*p q*))) (*p q*)

Deiteration: (*p* (*q* (*q*))) (*p q*)

Deiteration:	(*p* (*q* ( )))	(*p q*)
Erasure:	(*p* (( )))	(*p q*)
Double negation:	(*p*)	(*p q*)

Thus *p* is false. Since we wish our system to be logically complete we must be able to prove this. For this reason we must submit simple graphs to an additional mechanism. This mechanism is the general purpose theorem prover to be described next.

We note that, if a simple graph does not contain any generic referents and it has already been proved by lookup then there is no requirement to submit it to the deductive mechanism at all. This follows since there cannot be any alternative specialisations in the absence of the generic marker.

4.5.3 A Deductive Mechanism

We must now define more formally how a general deductive mechanism based upon Peirce logic will work. We start with some definitions.

Definition 4.4: Rule

A *rule* is any compound graph. It does not conform to any particular clausal form other than the conjunctive form imposed upon all compound graphs by the logical primitives of Peirce logic.

Definition 4.5: Goal

A *goal* is any simple graph submitted to the deductive mechanism.

Definition 4.6: Consequent

The *consequent* of a rule *r* is a simple graph *g* in *r* which unifies with a goal.

Definition 4.7: Antecedent

An *antecedent* of a rule *r* is any simple graph *g* in *r*, other than the consequent.

Deductive Mechanism

The detailed algorithms for the deductive mechanism are given in Chapter 5. We discuss general principles here. The deductive system will possess the following properties:

* It will be used for the proof of simple graphs.
* It will not distinguish between 'antecedents' and 'consequents'.
* It will implement both *modus ponens* and *modus tollens* automatically and may mix both rules of inference in any one proof.
* It will be fully recursive.
* It will use an intelligent matching algorithm for unifying goals with 'heads' of rules. In principle this will be the restricted maximal join as discussed in 4.5.2 but certain efficiency measures will be incorporated.

4.6 THEOREM PROVING

Theorem proving techniques can be placed into one of the following categories:

* Data driven/bottom up.
* Goal driven/top down.

In a data driven method the approach is to attempt to derive the formula to be proved by construction using proper axioms as a starting point and rules of inference or axiom schemata as the building tools. In a goal driven method the

approach is to attempt to reduce the formula to be proved to some recognisable endpoint. We will consider the merits of each.

Within each of the above categories, theorem proving techniques can also be placed into either of the following two categories:

* Non-uniform.

* Uniform.

In a non-uniform technique there is a selection of move types at each step of a proof and there is no specific form that clauses must take. This selection of moves corresponds to a choice of inference rule. Each example of a data driven or a goal driven method will also be of the non-uniform type. In a uniform technique there is one inference rule and a standard clausal form.

We will look at a small selection of existing techniques covering each of the four types and then summarise the above points. Since this is a brief review and to save space, the discussion will concentrate on the applications to propositional logic but the arguments apply equally to the more general first order equivalent methods.

4.6.1 Data Driven Non-Uniform Approaches

The traditional approach to proving theorems is one in which the rules of inference, along with appropriate axioms, are used to attempt to generate the theorem to be proved from the axioms and a set of premises. We will briefly consider two variants of this theme: *natural deduction proof methods* and *sequent proof methods.*

4.6.1.1 Natural Deduction Proof Methods

A natural deduction proof method is a non-uniform method which is supposed to mimic the way in which humans carry out proofs. Such methods were introduced independently in 1934 by Gentzen [Gentzen34] and Jaskowski [Jaskowski34]. A natural deduction proof of a formula *f* proceeds by attempting to show that the proper axioms entail *f* by applying rewrite rules of construction. An important aspect of a natural deduction proof, the aspect that makes it human-like, is that an appropriate set of proper axioms are assumed or

supposed. It is then shown that these suppositions themselves entail *f* and that the suppositions themselves are entailed by the proper axioms.

To illustrate, we will assume the following rules for a natural deduction system of propositional logic [Frost86]:

a) From (p, q) infer p ∧ q
b) From p infer p ∨ q
c) From p infer q ∨ p
d) From (p |- FALSE) infer ~p
e) From (|- q) infer (p ⇒ q)
f) From p ∧ q infer p
g) From p ∧ q infer q
h) From (p ∨ q, p |- r, q |- r) infer r
i) From p, ~p infer FALSE
j) From ~~p infer p
k) From (p, p ⇒ q) infer q

Fitch [Fitch52] classified the rules of natural deduction systems into *introduction* rules and *exploitation* rules. In the set above, rules a) to e) are introduction rules since they add propositions and the remainder are exploitation rules since they remove propositions.

We now wish to prove the formula (p ∧ q) ⇒ (r ⇒ q) under a theory with the proper axiom p ∧ q. Beginning by making the supposition p ∧ q, we can proceed as follows:

Supposition:	(p ∧ q)
Rule b on 1:	(p ∧ q) ∨ (p ∧ q)
Rule g on 1:	q
Rule e on 3:	(r ⇒ q)
Rule e on 1 and 4:	(p ∧ q) ⇒ (r ⇒ q)
Thus:	(p ∧ q) |- (p ∧ q) ⇒ (r ⇒ q)
Proper axiom:	p ∧ q
Rule h on 2, 6 and 7:	(p ∧ q) ⇒ (r ⇒ q)

This procedure has taken 8 steps and its successful outcome has relied heavily on the ability and experience of the human who constructed it (note the nature

of the initial supposition - it bears a striking resemblance to the proper axiom used later). Nevertheless, there were not any normalisation stages and thus the original structure of the formula, and hence the possibility of applying heuristics on the basis of that structure, is still present. What these heuristics are, in any general sense, is anybody's guess.

4.6.1.2 Sequent Proof Methods

The sequent proof method was also devised by Gentzen [Gentzen34] and is a non-uniform proof method. A *sequent* is a clause with the general form:

$$A_1 \wedge A_2 \wedge \ldots \wedge A_{na} \rightarrow C_1 \vee C_2 \vee \ldots \vee C_{nc}$$

The terms A_n are *antecedents* and the terms C_n are *consequents* or *succedents*. Sequents can be changed by the application of rewrite rules which fall into four groups:

Antecedent Introduction Rules

$$A\rightarrow \quad \frac{P \rightarrow Q,\ a \qquad\qquad b,\ R \rightarrow S}{a \rightarrow b,\ P,\ R \rightarrow Q,\ S}$$

$$A\wedge \quad \frac{a,\ P \rightarrow Q}{a \wedge b,\ P \rightarrow Q} \qquad \frac{b,\ P \rightarrow Q}{a \wedge b,\ P \rightarrow Q}$$

$$A\vee \quad \frac{a,\ P \rightarrow Q \qquad\qquad b,\ P \rightarrow Q}{a \vee b,\ P \rightarrow Q}$$

$$A\neg \quad \frac{P \rightarrow Q,\ a}{\neg a,\ P \rightarrow Q}$$

Consequent Introduction Rules

$$C\rightarrow \quad \frac{a,\ P \rightarrow Q,\ b}{P \rightarrow Q,\ a \rightarrow b}$$

$C\wedge$ $\quad P \to Q, a \qquad\qquad P \to Q, b$

$\quad\quad$ ----------

$\quad\quad P \to Q, a \wedge b$

$C\vee$ $\quad P \to Q, a \qquad\qquad P \to Q, b$

$\quad\quad$ ---------- $\qquad$ ----------

$\quad\quad P \to Q, a \vee b \qquad\qquad P \to Q, a \vee b$

$C\neg$ $\quad a, P \to Q$

$\quad\quad$ ----------

$\quad\quad P \to Q, \neg a$

Antecedent Modification Rules

Weakening $\quad P \to Q$

$\quad\quad$ ----------

$\quad\quad a, P \to Q$

Contraction $\quad a, a, P \to Q$

$\quad\quad$ ----------

$\quad\quad a, P \to Q$

Interchange $\quad P, a, b, Q \to R$

$\quad\quad$ ----------

$\quad\quad P, b, a, Q \to R$

Cut $\quad P \to Q, a \qquad\qquad a, R \to S$

$\quad\quad$ ----------

$\quad\quad P, R \to Q, S$

Consequent Modification Rules

Weakening $\quad P \to Q$

$\quad\quad$ ----------

$\quad\quad P \to Q, a$

Contraction	$P \rightarrow Q, a, a$ ---------------- $P \rightarrow Q, a$
Interchange	$P \rightarrow Q, a, b, R$ -------------------- $P \rightarrow Q, b, a, R$
Cut	$P \rightarrow Q, a \qquad a, R \rightarrow S$ -- $P, R \rightarrow Q, S$

We will illustrate the use of these rules by using them to prove the same formula that we used for the illustration of natural deduction, $(p \wedge q) \rightarrow (r \rightarrow q)$ but we note without further comment the slightly different symbolism ($\rightarrow$ instead of $\Rightarrow$). We start with a tautology and show that the formula can be derived from this:

Tautology:	$q \rightarrow q$ -----------
Antecedent weakening:	$p, q \rightarrow q$ --------------
Antecedent weakening:	$p, q, r \rightarrow q$ ------------------
Consequent introduction rule C$\rightarrow$:	$p, q \rightarrow r \rightarrow q$ ----------------------
Antecedent introduction rule A$\wedge$:	$(p \wedge q) \rightarrow r \rightarrow q$ ----------------------------
Consequent introduction rule C$\rightarrow$:	$\rightarrow (p \wedge q) \rightarrow (r \rightarrow q)$

This procedure has taken 6 steps, which is 2 less than the natural deduction proof. But its successful outcome has again relied heavily on the ability and experience of the human who constructed it (the initial tautology is carefully chosen to include the one proposition that cannot be introduced otherwise - in other words its choice was based upon prior knowledge of how to obtain a successful outcome). Once again, there were not any normalisation stages and thus the original structure of the formula, and hence the possibility of applying

heuristics on the basis of that structure, is still present.

4.6.1.3 Conclusion Of Analysis Of Data Driven Non-Uniform Methods

There is a problem with the data driven approach in that, in general, it is impossible to write a program that is both logically complete and guaranteed to terminate. It is easy to see why; the proof proceeds by adding information to the developing theorem and then pruning it as appropriate. By doing this it is hoped that the final theorem can be arrived at, and the limited range of axioms provides some direction for a proof to take since at any stage there will be a finite number of next moves. However, it is not generally possible to know how much to add because the next piece of information added may, after some pruning, lead to a proof or it may not. If it does then there is not any problem but if not then it is not possible to say whether backtracking ought to occur or whether the next addition of something else will be enough to allow the appropriate pruning to give a proof. Whilst it is true that a breadth first search of the proof tree will eventually terminate if there is a proof, this is not guaranteed to occur in a reasonable time. A depth first search could easily become infinite.

One way round some of this problem is to impose a maximum depth to the search or to impose some restrictions on which branches to take. Whilst this can be made to guarantee termination it introduces the possibility (certainty) of incompleteness. Such theorem provers are called 'safe' theorem provers since they will always give up eventually. Another method of overcoming the problem is to provide domain dependent heuristics. Whilst these can certainly guide proofs of problems in a particular domain they often do not work in other domains and even where they do work they must be taught explicitly to the system. This necessity has two consequences. The first is that the user or producer of such a system must know how to solve certain problems in order to be able to provide useful heuristics. This requirement is contrary to our thesis which is that the machine should perform the difficult tasks using some form of 'common sense'. The second consequence is that any system which is not in possession of a 'complete' set of heuristics may still not be able to solve some problems in a reasonable time which might have simple solutions.

Thus we must develop a system that is both complete without the need for domain dependant heuristics and which is guaranteed to terminate in a reasonable time. Nevertheless, we must also recognise that search spaces will

be larger in such a general system than in a restricted system but we argue that this is an acceptable price to pay for a system which is completely general. A corollary to the generality of the system is that it becomes completely declarative.

4.6.2 Goal Driven Non-Uniform Approaches

In a goal driven approach to proving a formula the aim is to start with the formula and show that it could have been generated from a set of premises. In other words, the formula is taken apart a step at a time. We will consider the technique of the *semantic tableau.*

4.6.2.1 Semantic Tableau Methods

As a second type of a non-uniform proof method, this method of proof is essentially the opposite of the sequent method. A tableau proof proceeds in the most general terms by a series of steps in which parts of a formula are proved separately and eliminated. Elimination is signalled by the derivation of a contradiction between a proposition generated at one point in the proof and its negation being generated at a later stage. When this occurs the tableau is said to be closed and the proof is complete. Proofs usually consist of branches and subbranches and when each of these is closed the proof is complete. If it proves to be impossible to close the tableau then there is not a proof.

An obvious benefit of the tableau method over the data driven approach is that the proof is guaranteed to terminate. This follows from the fact that any branch which does not close can only be split into a finite number of additional branches, a point which itself follows from the fact that all logical formulae are finite by definition. In addition it is also complete as long as the allowed moves do not prevent this from being the case. However, if the number of possible moves is large then there is still some degree of difficulty over the time taken to find a proof.

The rules of the tableau method to be used are the inverse of those of the sequent proof shown earlier. They can be summarised as follows:

R→ If p → q appears on the right of the tableau the write p on the left and write q on the right.

L→ If p → q appears on the left of the tableau then create two branches in the tableau and write p on the right of one branch and write q on the left of the other branch.

R∧ If p ∧ q appears on the right of the tableau then create two branches in the tableau and write p on the right of one branch and write q on the right of the other branch.

L∧ If p ∧ q appears on the left of the tableau then write p on the left and write q on the left.

R∨ If p ∨ q appears on the right of the tableau then write p on the right and write q on the right.

L∨ If p ∨ q appears on the left of the tableau then create two branches in the tableau and write p on the left of one branch and write q on the left of the other branch.

R¬ If ¬p appears on the right of the tableau then write p on the left.

L¬ If ¬p appears on the left of the tableau then write p on the right.

We now show how the formula (p ∧ q) → (r → q) can be proved in the tableau method. To begin, we locate the → with the 'highest priority' and write the formula in such a way that the antecedent of the → is shown on the left and the consequent is shown on the right. Following that we apply the rules as shown:

Start position:	p ∧ q | r → q
L∧:	p |
	q |
R→:	r | q

At this point we see that q appears on both sides of the tableau. This signals the end of the proof (of this branch) and the tableau (branch) is said to *close* at this point. In this simple proof there was only one branch. There will in general be several branches, each of which must close for the proof to be complete.

Although not too clear from this simple example (chosen to save space

and to be directly comparable to previous proofs) the principle of the tableau method depends upon splitting formulae and subformulae in this way until freestanding formulae are derived such that a branch *b* is closed when any freestanding formula on one side of *b* also appears freestanding on the opposite side of *b* or a parent of *b*. This implies that there is a finite number of possible moves in the proof since there must be a finite number of times any finite formula or subformula can be split. Hence the proof is guaranteed to terminate. Another point to note is that we were able to prove our standard formula without recourse to any proper axioms or suppositions. This is not unreasonable since the formula is a tautology and yet both the natural deduction proof and the sequent proof needed to be primed at some stage with additional information. The final point to note is that the proof was completed in three steps.

4.6.2.2 Conclusion Of Analysis Of Goal Driven Non-Uniform Methods

This brief survey of goal driven methods of theorem proving has demonstrated a technique in which a goal was proved by reduction to some definite endpoint which is a part of the formalism and therefore domain independent. The importance of this lies in the fact that a system can be designed which use this endpoint, and shortcuts to its attainment, as an heuristic. Since the endpoint is the same whatever the domain we have here a domain independent heuristic.

In addition to the universal heuristic we have the possibility of domain dependent heuristics. In the example above this is especially the case since the tableau method does not require any normalisation of the goal and hence there is no loss of any information that might guide a proof.

Also, as we discussed earlier, the goal driven method is guaranteed to terminate by its very nature. This makes it a very appealing method for an automated system.

As a final point, the above proof only took three steps.

4.6.3 Data Driven Uniform Methods

Uniform proof methods consist of a single inference rule which operates on a standardised form. In this way the choice of inference rules is reduced to the minimum and the structure of a formula is predetermined. These two points

are intended to combine to reduce complexity. We will review uniform proof methods with this in mind.

4.6.3.1 Resolution

The standard method of resolution is that developed by Robinson [Robinson65] and subsequently used in expert systems and databases. This method is also the basis of the Prolog programming language although its use here is restricted and Prolog does not provide a complete system of logic since many theorems can only be proved by writing programs.

Since resolution is a uniform method it requires that all formulae are converted to a standard form. In the case of resolution this is *clausal form* and is derived from the conjunctive normal form (CNF) of a formula. Thus, before any proof of some formula can take place it must be converted to CNF. The rules for this are:

$\rightarrow$ Elimination	For $p \rightarrow q$ write $\neg p \vee q$
$\neg$ Scope Reduction	For $\neg(p \vee q)$ write $\neg p \wedge \neg q$
	For $\neg(p \wedge q)$ write $\neg p \vee \neg q$
	For $\neg\neg p$ write p
Distributivity Law	For $p \vee (q \wedge r)$ write $(p \vee q) \wedge (p \vee r)$

We will use the example formula that we have used previously and attempt to provide a resolution proof of its validity under the theory with proper axiom $p \wedge q$. In order to do this we must adopt some strategy. This will be one of refutation in which we add the negation of the formula to be proved to the theory and then show that a contradiction exists. If a contradiction does exist then the negation of the formula is false therefore the formula itself is a theorem.

The formula to be proved (once again) is $(p \wedge q) \rightarrow (r \rightarrow q)$. We begin by adding the negation of this to the proper axiom and then converting to CNF:

Negation of goal plus proper axioms:

$$\neg((p \wedge q) \rightarrow (r \rightarrow q)) \wedge (p \wedge q)$$

→ Elimination:

$$\neg(\neg(p \wedge q) \vee (r \rightarrow q)) \wedge (p \wedge q)$$

→ Elimination:

$$\neg(\neg(p \wedge q) \vee (\neg r \vee q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$\neg((\neg p \vee \neg q) \vee (\neg r \vee q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$\neg(\neg p \vee \neg q) \wedge \neg(\neg r \vee q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$(\neg\neg p \wedge \neg\neg q) \wedge \neg(\neg r \vee q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$(\neg\neg p \wedge \neg\neg q) \wedge (\neg\neg r \wedge \neg q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$(p \wedge \neg\neg q) \wedge (\neg\neg r \wedge \neg q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$(p \wedge q) \wedge (\neg\neg r \wedge \neg q)) \wedge (p \wedge q)$$

¬ Scope Reduction:

$$(p \wedge q) \wedge (r \wedge \neg q)) \wedge (p \wedge q)$$

The formula is now in CNF. Ten steps were required to achieve CNF (although some of the steps may be carried out simultaneously). We must now convert to clausal form, which involves replacing all connectives with commas, surrounding the whole conjunction in braces (to generate a *clause*) and surrounding all nested conjuncts in braces (to generate *literals*):

Conversion to Clausal Form:

$$\{ \{ p \}, \{ q \}, \{ r \}, \{ \neg q \}, \{ p \}, \{ q \} \}$$

It is on this final form that the single inference rule of resolution is applied. This rule allows two literals of the form:

$\{ p, q \}$ $\{ \neg q, r \}$

to be *resolved* by 'cancelling' the q and ¬q and combining the remainders:

{ p, r }

The symbol { } is *null clause*. If this is provable by the inference rule then the original clause is false, since { } will result from the resolution of the two literals { p } and { ¬p }. Since the connective between any two literals is ∧ this represents a contradiction.

The approach that we adopted above was a refutation approach. With this in mind we apply the resolution inference rule to the clause that we have produced to give the final step of this proof:

Resolution:

{ { p }, { }, { r }, { p }, { q } }

We resolved { q } with{ ¬q } to give the resolvent literal { } and the proof is complete. This shows that the original negated clause is false. Nevertheless, since we negated our original goal we infer that the original must be true (as its negation lead to a contradiction). Thus the procedure has refuted the proposition that our original goal was false.

This particular resolution proof only took a single step (step 12) but it must also be remembered that the normalisation took 11 steps. Thus it can be argued that this resolution proof actually took 12 steps. However, it may also be argued that goals would only be submitted to a resolution theorem prover in CNF in the first place. This may be the case but the fact remains that any goal must be transformed at some stage by either a machine or a person. If it is the latter then that person must be sufficiently experienced in such transformations otherwise there will be mistakes. In addition, the very extensive transformation of even the simple example here has removed any structural information present in the original goal so that any opportunity to employ heuristics based upon that structure are almost certainly lost.

Resolution Variants

There is a further problem that also exists with resolution but which did not become apparent in this simple example. This problem is that it is possible for very large (even infinite) chains and webs of resolutions to be generated.

Indeed, the problem is similar to that for a data driven methodology. Thus a resolution theorem prover must be controlled by applying various strategies for detecting the most promising route at any stage. Despite these problems, restricted systems based on resolution can be implemented and are common.

Prolog is an example of a resolution system. However, Prolog is not complete since it adopts a restricted form of Horn clause and resolution is only complete for full Horn clauses. The restriction is necessary for efficiency and represents an admission that full resolution is not quite as good a practical tool as it could be.

Other ways of improving the efficiency of resolution have been developed. One such way is that which is termed LUSH ('Linear resolution with Unrestricted Selection function for Horn clauses') [Hill74]. In this application of resolution the selection of literals which are to be resolved is made on the basis of some simple rule that always selects particular resolvent literals from the previously generated resolvent clause. Another method is 'Selected Literal Resolution', or SL resolution [Kowalski71] [Kowalski75] in which an extension to LUSH is made in which a literal may be resolved with a literal derived at some step other than the previous step as long as it did not occur in the original clause.

4.6.3.2 Conclusion Of Analysis Of Data Driven Uniform Methods

The data driven uniform method described here suffers from the problem from which any data driven procedure suffers; that of the possibility of infinite search spaces and execution times. These can be tamed by applying restrictions to the logic to produce a safe system but the cost is loss of completeness. It is argued that the use of a single inference rule and a standard clausal form improves efficiency. Whilst it must be true that this principle reduces the number of choices it only reduces the number of ways in which an infinite search space can be created.

4.6.4 Goal Driven Uniform Methods

A range of goal driven uniform methods exists including: the *matrix reduction* of Prawitz [Prawitz76], the *matings method* of Andrews [Andrews81] and the *connection method* of Bibel [Bibel76] [Bibel82] [Bibel83]. In this section we will review Bibel's *connection method* to prove the formula $(p \wedge q) \rightarrow (r \rightarrow q)$.

In Bibel's original method a formula to be proved was first converted to disjunctive normal form and then to negative clausal form. It is, however, not uncommon to see CNF in use and since we have already performed this part of of the proof of $(p \wedge q) \rightarrow (r \rightarrow q)$ for the resolution proof we will save space by adopting CNF. Thus we will use the connection method to show that the clause { { p }, { q }, { r }, { ¬q }, { p }, { q } } is inconsistent and that, since we are performing a refutation proof, the original formula must be true.

A proof by the connection method involves setting up a matrix of literals from the CNF:

{ p }
{ q }
{ r }
{ ¬q }
{ p }
{ q }

This being done the objective is to show that any path through the matrix which passes through one proposition in each literal also passes through contradictory propositions. If all such paths pass through a pair of contradictory propositions then the clause set is inconsistent.

We can see that the formula that we have chosen for all our examples has produced a matrix through which there is only a single path and that this path passes through both q and ¬q and is therefore inconsistent. Hence the original formula is true.

The matrix above happens to only contain only literals with a single proposition. In general this will not be the case and there may be potentially large numbers of paths through the matrix. It is possible to greatly optimise the search for paths through a matrix since any path terminates as soon as a contradiction is found. In this way it is not necessary to complete the extension of any path after a contradiction is located on it. In addition, the conversion to clausal form marks the end of the generative phase of the proof.

4.6.4.2 Conclusion Of Analysis Of Goal Driven Uniform Methods

Being goal driven a goal driven uniform method possesses the same advantage that a non-uniform goal driven method possesses; that of tractability. In a goal driven approach the proof is guaranteed to terminate. Since there is only a single rule of inference there are less options at each step and a goal driven uniform proof method may therefore proceed more efficiently than a non-uniform goal driven method. Nevertheless there may still be scope for heuristics to optimise the search in a goal driven method.

4.6.4.3 General Conclusions Relation To Uniform Methods

We have shown four distinct approaches to theorem proving. Whilst some of these methods are very successful in some ways (such as resolution being used in Prolog) they may not produce proofs which are particularly intuitive to humans. For instance, resolution requires that clauses be normalised to CNF and the connection methods use disjunctive normal form, the form which Peirce used for his early entiative graphs but which he abandoned because of its unnatural feel. Methods based on clausal forms also require a certain amount of preprocessing of their inputs before proofs of queries can proceed. Whilst this is not a theoretical problem, in some ways such preprocessing may alter the way that the problem is expressed. This may alter the way that explanations related to it are to be interpreted so that they become more obscure, and any possibility of using domain dependent heuristics based upon the original structure of the problem may be lost. Not only that but, as we have shown, the conversion to clausal form may take many steps and is itself a significant part of a proof.

4.6.5 Summary Of Review Of Proof Methods

Uniform methods were developed to overcome the problems of having a wide choice of axioms and inference rules. In this respect they score well as long as the compromise that is made by the alteration of the structure of the original goal is acceptable. Thus the principle of reduction of complexity by this means is established. Nevertheless it is still necessary to consider whether a data driven approach or a goal driven approach is best. We must conclude that the goal driven approach is more secure than the data driven approach since it is guaranteed to terminate.

Several of the methods required some ingenuity on the part of the human performing them about how they started. This requires certain insights which link the the goal to possible starting places and which suggests that in some sense the direction in which a proof must proceed is known in advance. In supposing that such advanced knowledge is possible in general we only move the problem of knowing how to start a proof to the point where the proof strategy is selected. This selection will itself require some insight. One of the example proof method types, goal driven, did not require such metaknowledge and yet the proof by this method generally required fewer steps than the data driven methods. The starting point of thee proofs was the goal itself. Within this group there is a further consideration. Uniform goal driven methods proceed from a normalised clause which has lost its original structure whilst non-uniform goal driven methods proceed from the original form of goal. Although the proof part proper of the uniform methods was shorter than that of the non-uniform methods the normalisation of even the simple example that we used was lengthy.

Thus we must conclude that the goal driven approach has advantages over the data driven approach and that the uniform methods may have advantages over the non-uniform methods if the loss of the original structure of a goal is acceptable.

4.7 PEIRCE LOGIC AND PROOF STRATEGIES

Sections 3.6.1 and 3.6.2 describe a set of inference rules for conceptual graphs. These rules form a complete system of first order logic. It js not necessary to prove the soundness and completeness of these rules here since this has already been done [Roberts73]. What we wish to discover is how these rules might be used in computations and thereby gain some insights into how a theorem prover for Peirce logic may be developed.

4.7.1 The Nature Of Proofs In Peirce Logic

The discussion in Section 3.6.2 showed that the basic seven rules of inference of Peirce logic can be broken down into a much larger set of truly atomic rules. With these rules it is possible to construct from the only logical axiom (the empty set) and a set of proper axioms (the true set of graphs which

includes all negated contexts since these are also true) any theorem implied by those graphs. We will now perform a proof of the formula $(p \wedge q) \Rightarrow (r \Rightarrow q)$ with the beta rules.

We begin by converting the formula into Peirce form. To do this we must define the following rewrite rules:

P⇒ For $p \Rightarrow q$ write (p (q))

P∧ For $p \wedge q$ write p q

P∨ For $p \vee q$ write ((p) (q))

P¬ For $\neg p$ write (p)

Before we can perform the conversion we must first replace the parentheses in the standard notation with square brackets (at the risk of confusion with concepts) since the Peirce notation (as defined by us) uses parentheses to represent negation rather than operator priority. This generates the goal formula $[p \wedge q] \Rightarrow [r \Rightarrow q]$. Since the square brackets here are precedence operators and since Peirce logic shows operator precedence by the nesting of negated contexts the square brackets can be dropped in the final form. Thus the conversion is:

Start: $[p \wedge q] \Rightarrow [r \Rightarrow q]$

P⇒: $[p \wedge q] \Rightarrow [(r (q))]$

P⇒: $([p \wedge q] ([(r (q))]))$

P∧: $([p q] ([(r (q))]))$

Thus: (p q ((r (q))))

These rules do not form a system for normalising formulae. They are simply rules to convert between different notations. Since in Peirce logic there is only negation and one connective (∧) we can say that all formulae of Peirce logic are in effect normalised by default. Since all formulae of Peirce logic are by default normalised then a proof must be uniform with respect to the representation. Nevertheless, since there are several rules of inference, proofs using the beta rules are not uniform proofs.

The Peirce form of the formula shows us what is the endpoint of the proof. Here is one possibility:

Axiom:	{ }	
Double negation:	(( ))	braces dropped since they only delimit the axiom.
Double negation:	(((( ))))	
Insertion:	(p ((( ))))	
Insertion:	(p q ((( ))))	
Insertion:	(p q ((r ( ))))	
Iteration:	(p q ((r (q))))	

This proof is complete since we have generated our goal formula from the axiom. The proof did not require any other information. This means that the starting point for any proof is fixed and does not rely on judicious selection of suppositions or tautologies. Nevertheless, the proof could be carried out in many ways and the selection of the best way may depend upon the experience of the human who performs it. This is particularly the case where formulae from the sheet of assertion may be needed during a proof. Indeed there is no guarantee that a proof will be found even if one exists.

Since they provide a means of constructing theorems from a set of axioms, Peirce's beta rules form a data driven approach to theorem proving, with all the problems of such methods. It is for this reason, as discussed in the examples in Section 4.6, that there does not exist a logically complete theorem prover for the beta rules in their native form. It is for this reason that workers in conceptual graphs turn to existing methodologies when developing proof procedures and it is for this reason that we, also, will seek an alternative to the data driven approach. We will, however, retain Peirce logic and the existential graphs as the vehicle for our methodology since it is ideally suited to conceptual graphs and we would not be using conceptual graphs otherwise.

In this work we wish to establish an approach to theorem proving which is

based upon the philosophy of the goal driven method which uses Peirce logic as its logical basis. We see this as providing an opportunity to develop a logically complete and therefore purely declarative system. However, we must carry out some extra work before we can perform goal driven proofs since Peirce logic, by its very nature, will not permit them as it stands.

The theorem prover itself is developed in a later section but we must now consider the implications of the goal driven approach for Peirce logic in general and the beta rules in particular. Peirce's beta rules produce what Peirce called *endoporeutic evaluation*. This term was meant to suggest that formulae could be constructed from the outside in. We will show that a goal driven approach will produce an inside out approach.

4.7.2 Peirce Logic And The Goal Driven Method Of Proof

In order to be able to perform a goal driven proof using Peirce logic we must be able to use it to show that it is possible to reduce a formula *f* of Peirce logic to { }. If this is possible then *f* is a theorem. Furthermore, since we have adopted open world semantics we can expect that there will be formulae which cannot be reduced to { } but which can be reduced to ( ), in which case the formula is false. Yet other formulae may not be reducible to either { } or ( ) in which case they are unknown. In this latter case we require that the undecidability of the formula is guaranteed to be shown and shown in a reasonable time.

Any proof must be possible in all and only those cases where it is also possible to construct the formula by a data driven proof. Thus we are showing that a data driven proof can be reversed to an equivalent goal driven proof. We now consider the inference rules of Peirce logic from this aspect. Where appropriate in this and later sections, example goal driven proofs will be supported by data driven proofs. Since the data driven proof methodology has been shown to be sound and complete we will adopt it as a criterion for proving the soundness and completeness of our results.

Note that we have already considerably reduced the search space since an open world goal driven proof does not know in advance whether { } or ( ) will result and will simply stop when either is derived or nothing more can be removed. An open world data driven proof requires that two attempts be made:

one starting from { } to try to show the formula to be true and one starting from ( ) to try to show the formula to be false. The second attempt will only be necessary if the first attempt fails since no theorem can be shown to be true and false at the same time unless the knowledge base is inconsistent. However, since the execution time for the failure to prove that a formula is true by a logically complete data driven proof system is potentially infinite the attempt to show it false may never be made even if a trivial proof would have been found at the first attempt.

We must now show how Peirce's beta rules can be used in a goal driven proof methodology.

4.7.2.1 Reversibility Of Inference Rules

It is immediately apparent that any data driven proof of a theorem must also be a goal driven proof if all the rules of inference used in the data driven proof can be used in reverse in the goal driven proof. Thus we can see by way of a simple example that if the sheet of assertion contains the proposition *p* then the following is a data driven proof of (*p* (*p*)):

Axiom:	{ }
Double negation:	(( ))
(braces dropped for clarity)	
Iteration of *p* on sheet of assertion:	(*p* ( ))
Iteration:	(*p* (*p*))

The next step is to show that the proof can proceed in reverse:

Start:	(*p* (*p*))	(1)
Deiteration:	(*p* ( ))	(2)
Deiteration of *p* on sheet from assertion:	(( ))	(3)
Double negation:	{ }	(4)

Thus we have shown that in this case it is possible to reverse the data driven proof and generate a goal driven proof. Not only that but each step is more or

less predetermined by what action is possible on the current state of the proof. Thus in step 2 the only possible move is one which removes information. It does not matter whether deiteration or erasure is used at this stage since the results are the same in each case. Similarly, steps 3 and 4 can only proceed in one way.

Another example will take the discussion somewhat further. If the sheet of assertion is { } then the following is a data driven proof of (*p* (*p*)):

Axiom:	{ }
Double negation:	(( ))
Insertion:	(*p* ( ))
Iteration:	(*p* (*p*))

In other words it is very similar to the previous data driven proof. We now attempt to reverse it to give a goal driven proof:

Start:	(*p* (*p*))
Deiteration:	(*p* ( ))

Now we are stuck. The rules of Peirce logic do not allow us to proceed as there is nothing we can do to remove the remaining *p* now that the sheet of assertion is { }. The conclusion is that Peirce's rules are not in general reversible (though some are) and that we must find some way round this if we wish to provide a complete goal driven method of proof.

So far we have not considered the full first order scope of the beta rules. We will do so after developing the general principles in terms of the propositional alpha rules.

Reversible Rules

The following is a list giving pairs of reversible rules with examples:

*	addition of double negation	{ } => { (( )) }
	removal of double negation	{ (( )) } => { }

*	iteration	*p* => *p p*
	deiteration	*p p* => *p*
*	coreferent join	[ ]->( )->[: *x] [: *x]->( )->[ ]
		=> [ ]->( )->[: *x]->( )->[ ]
	detachment	[ ]->( )->[: *x]->( )->[ ]
		=> [ ]->( )->[: *x] [: *x]->( )->[ ]

This last example uses the rule of detachment that we reintroduced into the theory in Chapter 3. From this lists we can see that any data driven proof which uses only the reversible rules of inference is also a goal driven proof when reversed.

Irreversible Rules

The following rules are not reversible:

* Erasure.

* Insertion.

Not only is erasure irreversible but it exhibits another undesirable property when used in a goal driven proof. In attempting to derive the empty set { } from any formula, erasure could be used as the first and only step in any goal driven proof since all formulae are nested at depth 0. By this means any formula whatever, including ( ), could be shown to be true and so we therefore observe that the rule of erasure is unsound when used as part of a goal driven proof. This is not a problem since we will show that we do not require it in any case.

The rules of erasure and insertion might be reversible in some cases if it were known how the parts of a formula arrived there in a data driven proof, but it is unreasonable to suppose that this is ever the case since the validity of any operation on any logical formula cannot depend on how that formula came to exist. Since the irreversible rules affect graphs at specific depths of nesting we must now consider how the irreversibility can be overcome.

4.7.2.2 Reversing The Effect Of An Erasure

The rules of Peirce logic allow any evenly enclosed structure to be

removed during a data driven proof. We also observe that, in any theorem (since insertion only operates on oddly enclosed contexts), any evenly enclosed graph can only have arisen in a data driven derivation either as a proper axiom or from an iteration. This implies that its removal during a goal driven proof is therefore always possible by deiteration if it can be removed at all. If any evenly enclosed graph cannot be removed by a deiteration then the formula cannot be a theorem.

We must consider the nature of deiteration somewhat more carefully since any evenly enclosed graph *g* in some formula may have resulted, in a data driven derivation, from an iteration and a series of erasures. Thus *g* may not be an exact copy of the graph which was originally iterated. Following the original iteration the following operations may have been performed on *g*:

* Generalisation of type labels by erasure.
* Generalisation of relational labels by erasure.
* Removal of complete relations by detachment and erasure.
* Removal of lines of identity by erasure.

It may be that a compound graph *c* was iterated and then modified. In this case the modifications might have included only the above four rules in the evenly enclosed graphs of the copy of *c* and the oddly enclosed graphs might only have been modified by:

* Specialisation of type labels by insertion.
* Specialisation of relational labels by insertion.
* Addition of complete relations by insertion and coreferent join.
* Addition of lines of identity by insertion.

If we recall our definition of projection of both simple and compound graphs we will see that the above ways in which an iterated graph *u* can be subsequently modified to *v* are precisely those which allow the projection of *v* onto *u*. Thus we can extend the definition of the rule of deiteration to include all those cases where any graph *v* can be deiterated if there exists a dominating graph *u* onto

which *v* projects.

Projection may therefore be used as a derived rule of inference and we recall that if a graph *u* projects onto a graph *v* then *u* is a generalisation of *v*. Having discussed Peirce's beta rules we also note that the use of combinations of each form of the rule of erasure can be applied to *v* to construct *u*. This allows us to make the following four observations:

* Where an evenly enclosed graph *u* dominates an evenly enclosed graph *v*, the graph *v* could have arisen by an iteration of *u* followed by erasures.

* Where an evenly enclosed graph *u* dominates an oddly enclosed graph *v*, the graph *v* could have arisen by erasure followed by iteration.

* Where an oddly enclosed graph *u* dominates an evenly enclosed graph *v*, the graph *v* could have arisen by an iteration of *u* followed by erasures.

* Where an oddly enclosed graph *u* dominates an oddly enclosed graph *v*, the graph *v* could have arisen by erasure followed by iteration.

In each case the graph *v* has been derived from *u* by the removal of information. Therefore the graph *v* in each case is a candidate for deiteration. The converse of each of these cases is the situation in which the graphs *u* and *v* are interchanged in the cases above. When this is done we see that the graph *v* can never be generated from *u* without adding information from elsewhere. In these cases the graph *v* is not a candidate for deiteration.

The result of the foregoing arguments is that the operation of projection can be used to determine if a graph is a candidate for deiteration. In each case where *v* is a generalisation of *u* we note that *v* will project onto *u* because in each case the graph *v* was generated from *u* by erasures. In this way the efficiency of a theorem prover is greatly improved since the use of a single projection operation prevents the need to perform each individual step in the generation of *v* from *u*. The section below which extends the goal driven discussion to first order logic will show that, for first order logic, there is more to

the detection of deiterable graphs than the simple projection criterion. However, for a propositional logic the projection criterion is sufficient.

4.7.2.3 Reversing The Effect Of An Insertion

The rule of insertion allows the following derivation:

(( )) => (*p* ( ))

However, there is not a rule that allows us to show that:

(*p* ( )) => (( ))

in the absence of any dominating copy of *p*.

Any use of the rule of insertion in a goal driven proof would only result in something that must be removed later by some means. Nevertheless, any formula that is being proved by the goal driven method may contain oddly enclosed graphs whose presence could only be the result of an insertion during a data driven derivation. In order to deal with this situation we must consider the truth table for implication:

p	*q*	*p => q*
T	T	T
T	F	F
F	T	T
F	F	T

We see that in both cases where *q* is true (when *q* is { }) then the implication is also true. Therefore it does not matter what the truth of *p* is. This allows us to conclude that the goal driven derivation:

(*p* ( )) => (( ))

is indeed valid. With this new rule we are able to reverse the effects of the rule of insertion in those cases where the graph ( ) arises by removal of evenly enclosed graphs. Since the removal of evenly enclosed graphs can only have occurred by deiteration we can guarantee that the derivation (*p* ( )) => (( )) is sound because the data driven derivation is:

Axiom:	{ }
Double negation:	(( ))
Insertion:	(*p* ( ))

Evenly Enclosed Graphs

There is a second aspect to this, namely the following axiom:

$p \wedge FALSE \Rightarrow FALSE$

With this rule we are in a position to allow the derivation:

$p\ (\ ) \Rightarrow (\ )$

This is the exact counterpart for evenly enclosed contexts to the previously derived rule.

Taken together these two rules form what we term the rule of *empty clause elimination*. In introducing it we have presented a new result which allows a goal driven proof method for Peirce logic that can retain the desirable property of completeness.

4.7.3 Further Considerations Related To Iteration

During the data driven derivation of any formula it might be the case that a graph is iterated and subsequently modified so that it no longer projects onto the dominant copy. This is true if either of the following occurs:

* The graph is a simple graph iterated into an oddly enclosed context and subsequently specialised.

* The graph is a compound graph which has been iterated into a context which permits modification by any rule operating in the 'wrong' context.

We illustrate with the following example:

Given:	(p (q)) (p q)
Prove:	(p)

A data driven proof might be:

Iteration:	(p (q (p q))) (p q)
Erasure:	(p (q (p q)))
Deiteration:	(p (q (q)))
Deiteration:	(p (q ( )))
Erasure:	(p (( )))
Double negation:	(p)

In the goal driven case the goal (p) is already undecomposable since it is a negated simple graph. Also, it will not project onto either proper axiom since it cannot be derived from either by erasures or insertions. The only goal driven proof is one which includes iteration of proper axioms:

Goal:	(p)
Iteration:	(p (p q))
Deiteration:	(p (q))
Lookup:	{ }

In effect, the goal has been 'resolved'. Thus we have introduced a data driven rule into our goal driven method and have apparently destroyed the advantage of the goal driven approach. However, we have already shown that the rule of iteration is only needed to add compound graphs to a proof (since the iteration of simple graphs does not allow a previously disallowed deiteration) and we will also show that for the 'resolution' of some graph most known compound graphs are irrelevant and any that are relevant only ever need be iterated once. These restrictions restore the property of guaranteed termination. Not only that but the goal driven proof required only three operations whilst the data driven proof required five. Chapter 5 will suggest some heuristic approaches to the problem of optimising the iteration of compound graphs. The heuristics suggested will be domain independent.

4.7.4 Propositional Proofs And First Order Proofs

Peirce's alpha rules are equivalent to propositional calculus and as such comprise the following rules:

* Erasure.
* Insertion.
* Iteration.
* Deiteration.
* Addition and removal of double negations.

We have added the rule of empty clause elimination and removed the rules of erasure and insertion, which makes a total of four rules by which we can generate equivalent propositional proofs by a goal driven method.

Peirce's beta rules are equivalent to first order predicate logic with equality. Section 3.6.2 showed that the seven basic rules as presented by Sowa can be viewed as a set of many different separate rules. After adding the rules of empty clause elimination and detachment and removing the rules of erasure and insertion we apparently have a large set of rules with which to perform first order goal driven proofs. Computationally, this is unrealistic and we must now look at ways of solving this problem.

4.7.4.1 The Projection Operation

We have already shown that projection can be used to detect deiterable graphs in a proposition proof. In order to extend the principle to first order proof we must consider how lines of identity come into existence in a data driven proof.

4.7.4.2 The Effect Of Restriction

By restricting each concept to the type label to which its referent conforms we are preventing either of the following two cases from blocking the progress of a proof:

* ([T1 : #x] ([T2 : #x]))
* [T1 : #x] ([T2 : #x])

where T2 < T1 & T2 :: #x.

In the first of these cases a proof would succeed in the event that the oddly enclosed concept were specialised by insertion to [T2 : #x]. The rule of

insertion is rather too arbitrary for our purposes because to use it in this way would be to try any number of possible insertions before happening to find one that worked. In addition the theorem prover would be burdened with an extra rule, with the consequential increase in complexity.

The second example could never be shown to be false because neither the evenly enclosed concept can ever be specialised by Peirce's rules nor the oddly enclose concept be generalised. Thus the oddly enclosed concept can never be deiterated.

When the rule of restriction is applied to each of the above cases we see that they are converted to:

* ([T2 : #x] ([T2 : #x]))

* [T2 : #x] ([T2 : #x])

where T2 < T1 & T2 :: #x.

In each case the deiteration is now possible without any further work and each can be proved.

We can therefore use the rule of restriction to specialise all individual concepts to their proper types. Once this is done there will not be any need to perform the operation of restriction again and thus we have removed a source of complexity.

4.7.5 The Rules Of Inference For The Goal Driven Method

We are now in a position to discuss in detail the rules of inference that we shall use in our goal driven theorem prover. To summarise, we have shown how to reduce all proofs to propositional proofs and that these can be constructed by using the following rules:

* Deiteration.
* Disjunctive elimination.
* Removal of double negation.
* Restriction.
* Iteration.

Definition 4.8: Deiteration

Any graph *u* which is generalisation of a dominant graph *v* may be deiterated subject to the following restrictions:

* Where *u* is evenly enclosed, no concept of *u* is connected by coreference link to a dominating concept in some graph other than *v*.

* No concept of *u* is connected by a coreference link to a dominated, evenly enclosed concept.

These restrictions to deiteration, which ought to be part of all versions of this rule, prevent the production of 'dangling' coreference links. In order to deiterate a graph containing these links they must first be removed. In a data driven methodology they can all be removed by erasure, which can be performed as part of the deiteration, but in a goal driven methodology they cannot be removed in this way.

Definition 4.9: Empty Clause Elimination

If the empty clause ( ) appears in any context during the course of a proof then all other graphs, either simple or compound, within that context and all lines of identity connected to them may be removed.

Definition 4.10: Removal Of Double Negation

If a double negation is derived at any point in a proof then it may be removed.

Definition 4.11: Restriction

Any concept whose type label is not the proper type label for its referent may have its type label replaced with the proper type label for that referent.

Definition 4.12: Iteration

Any compound graph on the sheet of assertion may be copied into any context.

4.7.5 Previous Work

We have already discussed the work which has been carried out into various linear resolution methodologies for conceptual graphs. There is also some work which bears some resemblance to, but was not responsible for, that presented above. This is the work of Foo [Foo93]. In this work Foo uses conceptual graphs in an inverse resolution process. With this process Foo is able to perform constructive induction to introduce new relations into graphs. He also shows how the inverse resolution process is analogous to the use of Horn Clauses as used for inverse resolution in predicate logic. He then cites applications where this form of reasoning may be useful.

The rules that Foo derives are similar to our rules for a goal driven theorem prover since they are quoted as being literal inverses of the beta rules. Whilst this is the case for some rules as we have shown, he appears not have taken into account the full implications of the endpoint of a goal driven proof and allows a data driven proof to be reversed by simple reversal of the beta rules. We illustrate with an example in which we attempt to use Foo's rules to prove the formula *p*:

Double negation:	(( ))	(1)
Insertion (reversed):	((*p*))	(2)
Double negation:	*p*	(3)

The derivation rests on the assumption that one can simply reverse erasure by

insertion and this is carried out at step (2). He sidesteps this problem in general by reversing data driven examples in the one way that does give the inverse proofs for which he is searching but we have shown that the insertion of new graphs requires heuristics (or some sense of how the proof must proceed) and results in computational complexity. In addition, he does not take full account of the nature of lines of identity and allows illegal removals of the kinds that we identified previously.

Foo does not claim that his approach will enable a general purpose theorem prover of the kind that we envisage to be built. Indeed, since Foo adopts an analogous approach to that of the use of Horn clauses we may argue that Foo's logic violates one of the requirements of our logic, that of preserving structural information in the topology of a query. Thus we argue that our approach represents a more general approach to problem of reversal of proofs and theorem proving by this means.

4.8 PROOF BY LAST RESORT

The 'last resort' method of proof is provided to allow the system to ask the user if any graph that it cannot prove itself is true or false. This allows a proof to proceed that would otherwise have failed and makes the system identify to the user areas where the knowledge is incomplete.

It is envisaged that this method of proof will provide the point of interaction with a user and will allow the implementation of expert systems if the knowledge base is given the appropriate kinds of knowledge. We offer an example of this application in Section 5.10.

4.9 CONCLUSIONS OF CHAPTER 4

We have shown how to use a canonical basis in a semantic validity testing system, We have compared this approach with a logical approach and shown that the advantages of the use of the canonical graphs was that of efficiency.

A means of retrieval of simple graphs by use of the open world evaluation game has been described and it was demonstrated that the semantics of this method matched the three valued logic that we desire.

A deductive mechanism was described and it was shown that this mechanism subsumed those of many other formalisms. In particular we developed a system that provided *modus tollens* in addition to the traditional *modus ponens*.

Approaches to theorem proving were compared with the result that we were able to recommend a goal driven approach for conceptual graphs. This necessitated that the rules of inference of Peirce logic be used to provide an appropriate set for this approach.

Finally, a means of providing a point of interaction with a user was mentioned. With this interface it will be possible to provide expert system capability.

CHAPTER 5

FROM THEORY TO IMPLEMENTATION

5.1 INTRODUCTION

This chapter is concerned with the description of a computational model and working prototype system which organises and implements the ideas presented in the previous chapters to produce a fully automated declarative knowledge based reasoning system. With this system it is possible to create models by 'telling' the system plain facts, which it verifies against its current knowledge base, and subsequently to query the model in an arbitrary manner. The operation of the system is carried out in a completely declarative way.

Both the model and the system are developed together since we believe that this approach permits the testing of the model in real situations as its development progresses. Such testing is important since it is intended that the system be usable and the concurrent development of each aspect of the project provides cross checking to ensure that this aim is achieved. In addition it allows other users, possibly conceptual graph naive, to evaluate the system, at least informally.

This system will be known as the CGP (conceptual graph processing) system, or CGP. Attention is concentrated on the general principles of the main algorithms, including mechanisms to increase efficiency by reducing the size of the search tree as far as possible and to provide domain independent heuristics. These points are not directly a part of the theory of the individual rules of inference as described in Chapter 4 but are necessary considerations for any working system and as such must be dealt with in a sound way in order not to compromise the completeness of the rules. As such the treatment will throw light on further generally applicable theoretical points.

The discussions which follow give some indication of how the prototype is implemented and as such will suggest recommended approaches. There are nevertheless areas of the prototype which are not implemented in the best possible way. Such aspects of the implementation result directly from the nature of the platform chosen for the implementation. Towards the end of this chapter (Section 5.13) there will be some discussion of these aspects.

5.2 GENERAL POINTS ABOUT THE SYSTEM

5.2.1 Choice Of Platform

Ultimately, the choice of platform for a production version of the system would be dictated by factors such as cost of hardware, processor speeds, possibility of parallelism and many other factors. Choice of language would be dictated by considering the merits of different kind of programming approach (procedural, declarative, object orientated). For the purposes of the present work the following criteria had to be met:

* Time spent learning a complex modern language to a sufficient degree to justify its use over a less advanced language was time that was not spent on the main thrust of the work. Languages such as C++ were therefore ruled out.

* Time spent on the development of basic data structures and language features which were already present in some languages but not others ruled out these other languages. This means that the majority of procedural languages were ruled out.

* The nature of the theorem prover was that of a system which required a backtracking mechanism. Such a mechanism is not difficult to program but it is work that is not necessary when it already exists within Prolog.

The third of these requirements indicates the final selection of language. However, Prolog has its own problems, the major one being that of speed. Even if a very efficient Prolog program is written, any program that requires to perform operations on individual members of Prolog's only (official) data structure, the list, is bound to require that the list is searched from the head in

order to locate the *n*th item. The use of lists is inevitable and so is the problem associated with it, even if list structures are optimised as trees.

Notwithstanding the possible problems of Prolog, this was chosen as the language for the prototype for the following reasons:

* The built-in data structure, the list, is sufficiently flexible to allow the representation of anything.

* Although slow, a compiled Prolog's execution speed is not prohibitive for the purposes of prototyping.

* Prolog's declarative nature and simple syntax make rapid program development possible.

* Prolog's built-in backtracking mechanism saves an enormous amount of programming work which is peripheral to the main purpose.

The Prolog that was chosen for the main development work was that supplied by Acornsoft Ltd (version 4.011) for Acorn's Archimedes computer. This Prolog is entirely compatible with the Edinburgh standard and is adequate for a research project, but not ideal since it is an interpreted Prolog. In addition, SICSTus Prolog on a Sun Sparc Station was used as an alternative platform for subjective efficiency tests and allowing availability of the system to other users. This Prolog is also entirely compatible with the Edinburgh standard and, as it can be compiled, gives an indication of the sort of enhancements that an efficiently written version of the system would be like. The Sun and Archimedes versions of the program are identical.

5.2.2 Logical Requirements Of The Knowledge Base

The knowledge base is developed according to the open world assumptions. For completeness these are:

* There is a set of TRUE graphs *T*.

* There is a set of FALSE graphs *F*.

* There is a set of domain laws *L*.

* There is s set of individual markers *I*.

The set *T* is stored as asserted Prolog clauses, each of which represents a single graph. The set *F* is stored, not as graphs in some compartment labelled *F*, but as negated graphs along with the members of the set *T*. There is a logical reason for this, which is that all 'negative knowledge' represents a true statement about the falsity of some proposition. Thus if *p* is false then it is true that *p* is false. In practice, false graphs are stored in a form which allows the system to search for them without also needing to search through the true graphs, and so they are in effect stored in some compartment labelled *F*. However, graphs which represent domain laws are not stored in some compartment labelled *L* but are stored in the same manner as the graphs of the set *F*. The members of the set *I* of individual markers are generated as required.

Our open world for conceptual graphs must be enhanced by the requirement to include selectional constraint information. Therefore there is a set of canonical graphs *C* which is additional to the above three components. In addition there are certain other components which exist for efficiency reasons to be discussed later. These are:

* Type lattice.

* Relational lattice.

* Conformity relations table.

* Lambda abstractions.

Each of these contain information that is implicit within graphs. As such, their inclusion within CGP (and within the theory) is made for implementational reasons rather than theoretical reasons. Further components of the system will be introduced as they are encountered.

Given this set of system components there are some logical considerations to be addressed:

* The open world model requires that each individual marker in *I* occurs only once in *T*. The corollary to this is the requirement for all graphs within *T* to be joined on identical concepts into as few graphs as possible. Thus, all new graphs which are added to the set *T* are joined by the coreferent join operation. We have already noted that the canonical formation rule of join is subsumed by this rule.

* The set *F* contains arbitrary rules and laws. For efficiency reasons, to be considered later, this set must be maintained in as compact a form as possible by applying techniques of rule subsumption and simplification.

* All type labels for individual concepts in *T* must be the proper type labels for the individual marker. Where any marker exists in two different concepts of types *t1* and *t2* where the type labels *t1* and *t2* do not have a common subtype other than ABSURD then a new subtype *t3* which is a subtype of both *t1* and *t2* must be created. *t1* and *t2* must both be replaced by *t3*, the proper type, and the concepts joined. The replacement of type labels with proper type labels in this way is a manifestation of the canonical formation rule of restrict.

* All implicit lattice and conformity relations knowledge must be added automatically to the appropriate component of the system. Thus if some type lattice relation is implied by some rule or second order graph then the relation must be added to the type lattice after the rule or graph has been accepted. If this is not done then the system will fail to correctly utilise this information at the appropriate times (during projection operations).

* The knowledge base is a conjunction nested at depth 0 and therefore all the Peirce operations that are allowed at depth 0 are allowed on the knowledge base.

5.2.3 General Description Of The System

We have developed a system which can be used as the basis of a general purpose knowledge base with the ability to maintain its own consistency and to answer arbitrary queries. In order to accomplish this it must accept new assertions in the form of conceptual graphs and test them for semantic correctness and logical consistency. It must then be able to answer any query whose graphs follow from those within the system. The whole process must be automated in that the system itself must decide which strategies to use.

Communication with the system is made *via* a simple command line which accepts commands of the form:

single_character_command [parameter_list].

The parameter list will normally be a context represented in linear form. The two most useful commands are the ! and ? commands, which allow the user to make assertions and queries respectively. Indeed, the whole system can be almost fully utilised by the use of these two commands only, which allow communication with CGP to proceed in a conversational manner. There is no attempt to provide a query language as such because the conceptual graphs that make up the parameter list must inevitably be adequate to model any possible graph in the knowledge base. This point follows from the fact that all knowledge in the system is represented by conceptual graphs. The use of conceptual graphs as the query language therefore allows the full expressiveness of the formalism to be used within queries and the completeness of the underlying system allows arbitrary queries to be answered.

5.3 EXECUTION CYCLE

The following cycle drives the system:

* The system waits for new graphs to arrive. These will either be queries or assertions. Queries are to be answered and assertions are to be added to the knowledge base if possible.

* Newly arrived graphs are subjected to a process of standardisation before being tested for violations of selectional constraints. This stage involves possible updating of lattice information as the position of newly encountered type and relation labels within their lattices must be taken into account.

* If there are any selectional constraint violations within the input then it is rejected and the cycle returns to the first stage.

* The logical consistency of the input is tested.

* If the input is FALSE or INCONSISTENT then it is rejected with explanations and the cycle returns to the first stage.

* If the input is a query then the truth value is reported with explanations along with variable instantiations. If there is more than one result then these are also reported. The cycle returns to the first stage.

* If the input is a statement then the truth value is reported. If the statement has truth value TRUE then the cycle returns to the first stage without any further action.

* If the input has truth value UNKNOWN then the new knowledge is added to the knowledge base.

* If the system is operating in goal driven mode then the cycle returns to the first stage otherwise the system enters a data driven deductive phase before returning to the first stage.

This cycle is embodied in the following algorithm:

Algorithm 5.1: Main Cycle

Let *D* be the input data;

```
1     Read D from input;
2     Test D for selectional constraint violations and update lattices;
3     IF NOT D contains selectional constraint violations THEN;
4         Evaluate D:
5         Report results;
6         IF D is an assertion AND D is unknown THEN;
7             Add D to the knowledge base;
8         ENDIF(6);
9     ELSE;
10        Report results;
11    ENDIF(3);
```

We now discuss the system in detail.

5.4 INPUT AND STANDARDISATION

Before anything can be done with the input it must be parsed into the correct form. Whilst this is a relatively simple task, for the purposes of the system we are describing the following points may be noted:

* All labels are allocated a unique reference number when first encountered. This includes all type labels, relational labels and name referents. Thus the label becomes a human-readable surrogate for the reference number. Reference numbers are equivalent to Sowa's # markers and the two are interchangeable, although in any real knowledge processing environment there would not be any need for a user to use the reference numbers as all types, relations and individual referents would be quoted *via* the original labels.

* The system of allocation of reference numbers to new labels ensures that when any particular name or label is used it always refers to the same reference number.

* Each generic referent is renamed with a unique reference number (all generic referents on any one line of identity each receiving the same reference number). This ensures that any two concepts such as:

  [TYPE : *]

  [TYPE : *]

  are not treated as being necessarily the same concept, which they are not.

Although there is not any strict clausal form required by the inference engine there are sound efficiency reasons for preprocessing all inputs. The preprocessing will be directed towards the aim of reducing the size and complexity of the input so that the inference engine has fewer steps to perform in its testing. The preprocessing therefore consists of the removal of duplicate and unnecessary parts of the input by simplification. With the augmentations to the rule of simplification as defined earlier the possible simplifications to the input are:

* Removal of double negations.
* Copying of individual referents to dominated coreferent concepts.
* Joining of graphs on identical concepts.
* Removal of duplicate concepts.
* Removal of duplicate relations.

It will be noted that the last two of these operations are simply manifestations of Sowa's canonical formation rule of simplification. The remaining three operations are operations of Peirce logic. The first two operations are performed first because they (in particular the second one) may affect the outcome of the last three operations.

One could also attempt to perform other operations on the input at this point. For example, much more deiteration than is specified here would further

reduce the complexity of the input but a balance must be drawn at this point between simplifying the input and actually carrying out too much real work. The real work of proving the input is a matter for the full theorem prover which has the power to explain its actions. It is therefore more realistic to send this partially reduced input to the theorem prover rather than have the standardisation phase perform all the work.

As a final point of standardisation, each context within the input is sorted so that graphs are placed before nested contexts. This ensures that two different contexts with very similar morphology will compile to very similar structures within the system. In this way a small improvement in efficiency is produced.

Once the input has been simplified in the ways shown here it is then in a correct state to proceed to its further processing. The following algorithm summarises this part of the system:

Algorithm 5.2: Input And Standardisation

1 Read a string of characters *D* from the input;
2 Parse *D* into graphs and extract all new labels;
3 Allocate unique markers to all new labels and add the labels and markers to the lexicon;
4 Replace all generic markers in *D* with unique system generated markers;
5 Copy all objects to dominated coreferent fields;
6 Remove duplicate concepts and relations from each context (leaving duplicates in other contexts);
7 Sort the graphs in each context into some canonical order;

5.5 SELECTIONAL CONSTRAINTS AND THE LATTICES

The first stage of the proof of any input is to establish its semantic validity against the selectional constraints. There are four stages to the testing of the input. These are:

* Placing new types and relations in the lattices.

* Requesting definitions of new types and relations (not strictly necessary at this point but convenient).

* Requesting canonical graphs for any new relations.

* Testing each relation within the input context against the appropriate canonical graph.

5.5.1 Placing New Type And Relation Labels Within The Lattices

In this section we adopt the convention that the term 'label' means either a type label or a relational label and the term 'lattice' means either the type lattice or the relational lattice. The terms 'supertype' and 'subtype' will be used to refer to either a supertype or superrelation, or to a subtype or a subrelation respectively; the context will disambiguate the use of the term.

If a new label has been encountered its position within the appropriate lattice will not be known. In this case it is impossible for the checking of selectional constraint violations to proceed because there is insufficient information. A question and answer mechanism exists within CGP which is invoked automatically and which interacts with the user to correctly place each new label within the correct lattice.

The system assumes that all new labels are subtypes of the appropriate universal type and then uses the following algorithm:

Algorithm 5.3: Placing Of Labels Within A Lattice

Let *u* be a supertype (initially the universal type) of a new label *t* which is to be placed within the lattice.

PLACE_LABEL(*u*,*t*)

1 **FOR** each immediate subtype $v_{i(i=[1,n])}$ of *u* **DO;**
2 **IF NOT** v_i = ABSURD **THEN;**
3 **IF** v_i is a supertype of *t* **THEN;**
4 **PLACE_LABEL**(v_i,*t*);

```
5               ELSE(3);
6                   IF vi=vn  AND NOT t < vi(i=[1,n]) THEN;
7                       Add t to the lattice as an immediate subtype of u;
8                   ENDIF(6);
9               ENDIF(3);
10          ENDIF(2);
11      ENDFOR(1);
12      IF NOT the relation ABSURD < t has been added to the lattice THEN;
13          Add the relation ABSURD < t to the lattice;
14      ENDIF(12);
```

This algorithm is used to place new relational labels in the relational lattice. In this case the word *type* is replaced with the word *relation.*

This algorithm will correctly handle situations in which the new label *t* is a common subtype of more than one label. The CGP system ascertains the truth of step 3 by asking the user, who replies with 'yes' or 'no'.

5.5.2 Requesting Definitions

As an additional refinement the algorithm can be made at step 7 to request a definition of the new label in terms of the type v_i. Whilst this may in one sense be seen as nothing more than a simple facility it does allow the system the opportunity to gather any extra knowledge at the earliest possible moment and avoid the risk of not being able to respond in full to queries which involve any understanding of the meaning of the new label. Humans generally do the same thing when presented with a word that they do not know.

When such a definition is supplied it can be checked for consistency, both for selectional constraint violations and logical consistence, at this point and accepted or rejected as appropriate.

5.5.3 Lattice Structure And Efficiency

The existence of the lattices in Sowa's theory provides conceptual graphs with an order sorted semantics as opposed to a less ordered many sorted semantics in which there are many types but no hierarchy. Sowa introduces the

lattice as the means of representing the sort hierarchy and also introduces the < relation between labels which participate in a relationship of the form *subtype < supertype.*

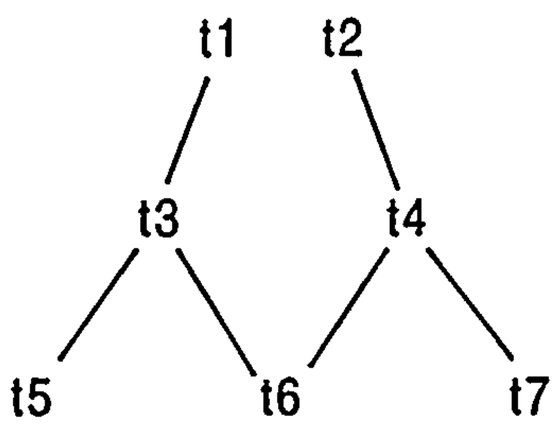
It can be argued that the lattice is unnecessary since the information it contains is implicit within the definitions. To give two examples:

```
1       ( [ TYPE1 : *x ] ( ...[ TYPE 2 : *x ]... ) )
        ( ...[ TYPE2 : *x ]... ( [ TYPE1 : *x ] ) )

2       TYPE1 = λ(x) ...[ TYPE2 : *x ]...
```

Example 1 is the logical equivalence form of definition and example 2 is one form of the abstraction form of definition. From example 1 we conclude that TYPE1 is a subtype of TYPE2 and that the relation TYPE1 < TYPE2 holds. Example 2 is slightly more explicit and again leads to the conclusion that TYPE1 < TYPE2 holds. It would seem that there is no need to store the lattice explicitly if the relation TYPE1 < TYPE2 can be efficiently retrieved from either of these forms. If the particular implementation does not admit of efficient retrieval of lattice relations from these structures then a separate lattice must be constructed and maintained. The CGP system is of the type that retains the explicit structures.

The next case shows how the computational complexity can increase when the type lattice, implicit or explicit, starts to grow. We will consider the following partial lattice:



Thus to make the test t3 < t1 we can directly look up the relation in the lattice. However, to make the test t6 < t1 we must rely on the transitivity of the < relation to allow us to prove t6 < t3 and then t3 < t1. This represents a doubling of the search space (in the case where the correct link t6 < t3 is found first time) and hence execution time. The situation is worse in the following case where it

is required to show that t6 < t7. In this case the system must try to find the link by trying the following tests:

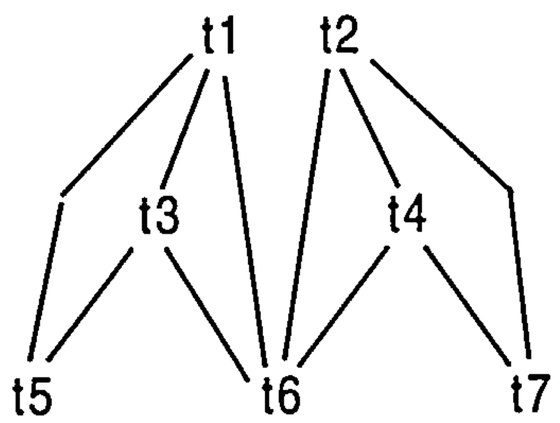
t6 < t7	no
t6 < t3	yes
t3 < t7	no
t6 < t4	yes
t4 < t7	no

Thus our simple lattice necessitates five tests in this case. The general formula for the average number of tests to find a supertype of *t* is:

> (1 + number of immediate supertypes *ti1* of *t* +
> number of immediate supertypes *ti2* of each type *ti1* +
> number of immediate supertypes *ti3* of each type *ti2* + ...)/2

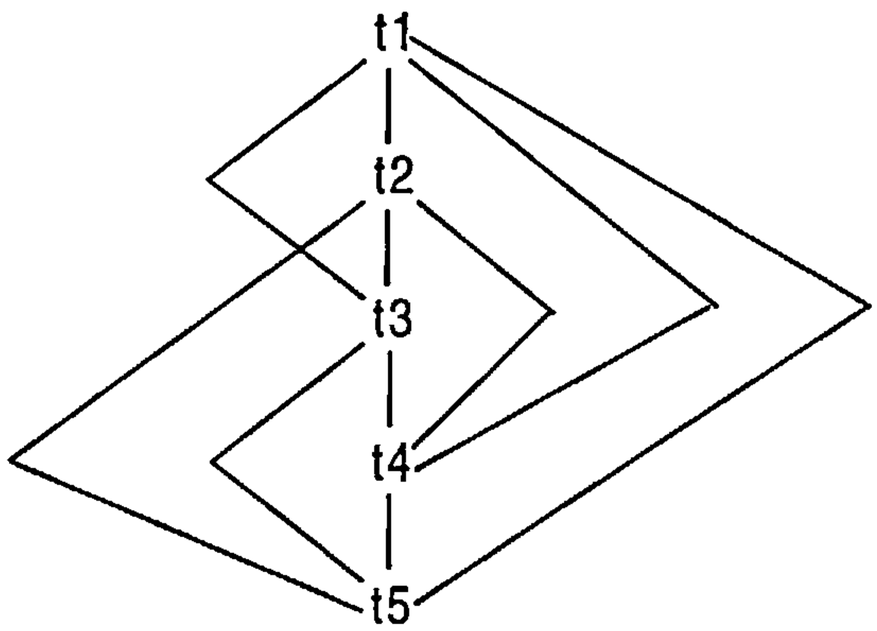
In other words, the number of tests in a realistically large lattice would become quite large (although it will not explode). As this kind of test is required very often (by general purpose use of the projection operation for instance) we must find a way of improving the efficiency.

The CGP system achieves this efficiency by storing direct links from each type label in the lattice to all its supertypes. Thus each test consists of a single search. The above fragment of a lattice would look like:



This lattice contains an additional four links between labels and supertypes separated by one 'layer' of the lattice. More storage is required for a lattice stored in this way. We can analyse the worst case in which the lattice is very deep and nonbushy:

In this case the number of additional links required to complete the lattice is six, to give:



The total number of links in a chain of *n* labels is given by the formula:

$$((n-1)^2 + n - 1)/2$$

This number may become large with large deep lattices but we may assume that the typical lattice is likely to be broad and shallow since if it were significantly otherwise the lattice would degenerate into a useless list. In these cases the length of each chain in such a lattice is likely to be small and hence the number of extra links will be small. For instance, the partial lattice used previously contains 7 labels and six links, which if they formed a single chain, would require 21 links in total. In the actual example above the lattice was somewhat bushy and only required 10 links in total and only 4 of these were extra links.

The CGP system tested informally with fully connected lattices and sparse lattices and was found to operate significantly faster with the fully connected lattice even where the lattice contained only a small number of nodes. This

reflects the heavy use of the lattice that the system makes.

5.5.4 Requesting Canonical Graphs

This function is vital to the processing of newly encountered relational labels. For each relational label known to the system there must be a canonical graph as described in Chapter 1. At the first encounter of any relational label the system will not be able to ascertain if the input is semantically valid until it possesses the appropriate canonical graph. This acquisition is made by the system asking the user (under the assumption that the correct one will be supplied) for the canonical graph. Once the graph is supplied it can be used to test the input. The test is necessary since the relation in the input may contain concepts which are subtypes of those in the canonical graph.

5.5.5 Testing For Selectional Constraint Violations

To test for violations of selectional constraints in an input it is necessary to take each relation *r* from the input in turn and perform the following tests with the canonical graph for that relation:

1 If the canonical graph *u* projects onto *r* then the *r* passes the canonicity test.

2 If *r* fails the test at the first attempt then produce a generalisation *g* of *r* by replacing each of the individual referents in *r* with different generic referents. Test each concept *cg* in *g* against the corresponding concept *cu* in the canonical graph by projecting *cg* onto *cu* or by projecting *cu* onto *cg*. If all concepts project then *r* passes the canonicity test otherwise the relation fails.

After testing each relation in turn then a list of failed relations will be empty if the input is well formed and non-empty otherwise.

Since we previously defined all malformed relations as false there is not any need to test any oddly enclosed relations for selectional constraint violations.

Step 2 requires further clarification. We note that there is a different

requirement for canonicity between queries and assertions. If the knowledge base contains the following graph:

[MAN : #1]->(FATHER)->[MAN : #2] (1)

then the following graph, which apparently violates the semantic constraints of the FATHER relation since it does not have MAN at the right hand end, is a reasonable query:

[PERSON : #1]->(FATHER)->[PERSON : #2] (2)

However, for assertions we must not allow such ambiguous graphs to be added to the knowledge base and so we must reject graph (2) as an assertion in case PERSON #2 later turns out to be a woman. Thus, for queries, step 2 is included in the checking but for assertions it is not.

5.6 COMPUTING DENOTATIONS AND TRUTH VALUES

This section describes various strategies that are required to generate a proof and as such provides the major computational model. Additionally there needs to be an overall strategy to control the selection of proof techniques.

For the system to be described, the denotation operator δ has the following properties:

* For a conjunction of simple graphs *G* the denotation δG is a truth value and a set of instances. Simple graphs can therefore be considered as the lambda abstractions that are used in other systems.

* For any compound graph *G* the denotation δG is a truth value only and shows the validity of the logical argument represented by *G*.

5.6.1 Overall Strategy

Proof Strategies

The system is a mixed strategy system. In order to compute truth values and denotations there are four main stages which make up the evaluation

procedure, although there is some overlap between them. These are:

* Lookup.
* Deduction.
* Theorem proving.
* Last resort.

Any part of the input may be proved by lookup. The deductive mechanism is reserved entirely for simple graphs and negated simple graphs. The theorem prover can be used for the proof of any graph. 'Last resort' is also available for any form of graph and provides the system with a means of interacting with a user during a proof. These proof strategies together form a complete system of exact reasoning.

When the evaluation mechanism receives an input it attempts to prove graphs within it by the appropriate means and then deiterate them, copying referents to any coreferent concepts, according to the goal driven philosophy previously developed. By adopting this approach the system is attempting to reduce the input to one of the following:

* The empty set { }.
* The empty clause ( ).
* Some non-reducible residue.

If the empty set is achieved then the input is TRUE. If the empty clause is achieved then the input is FALSE. If neither can be achieved then the input is UNKNOWN. We showed earlier that the rules of inference are complete and that UNKNOWN really means UNKNOWN. We will show later in this chapter that the system is also guaranteed to terminate and so UNKNOWN graphs will not require infinite time for their proof.

There are three pieces of information which are carried around the proof mechanism. These are:

* A record of all graphs currently being proved as subgoals of the main goal. This record is kept as a list and is termed the *trail* in view of its similarity to the similarly named structure in some implementations of Prolog [Campbell84].

* A record of variable instantiations. This is termed the *instantiation list* and is retained to allow the mechanism to correctly prevent the matching of one variable with more than one other object.

* A record of the steps taken in the proof for the purposes of reporting explanations.

Modes Of Operation

The system can operate in several modes. Some of these modes are simple ones that either enable or disable parts of the system. These will not be considered in detail in the main body of the thesis, but there is one mode of operation that fundamentally changes the way the system behaves. When this mode is active the system will make all possible deductions (of simple graphs and negated simple graphs) from any new knowledge and these will be stored. It will not, however, generate all possible complex graphs as these are infinite in number. The result of this is that when the system is operating in this mode the knowledge base is deductively closed. Therefore the deductive mechanism is not required when the system is answering any query. When the system is not operating in the deductive closure mode then there is no guarantee that any implicit simple graphs will exist explicitly within the knowledge base.

The existence of two modes of operation has repercussions on one important property of conceptual graphs theory - that the topology of the graphs can be used to direct proofs. If the knowledge base is deductively closed then any simple graph that can be at all satisfied will be satisfiable (after its reduction to its most contracted form) by the projection operation (the projection operation observes graph topology). However, if the knowledge base is not deductively closed then any simple graph may contain some relations that are explicitly known and some which are only implicit. To prove such a graph it is necessary to split up the graph into separate relations and prove these individually. In any

knowledge base that is not deductively closed there is no other way round this problem if the system is to be complete.

Thus, when the system is operating in the deductive closure mode the graphs that make up a query are not split up but when operating in the non deductive closure mode they are.

5.6.2 High Level Structure Of The System

We must now show how the basic components of the automated proof strategy are built, at the highest level, into a single scheme. In this scheme, formulae are passed to appropriate modules according to whether they are simple graphs and whether they have already been proved by earlier parts of the whole scheme.

Algorithm 5.4: Coordination Of Proof Strategies

If *G* is a formula then the testing of *G* for semantic and logical correctness in some world *W* is organised by the following high level algorithm:

```
Let T be the trail;
Let I be the instantiation list;
 1    Test G for semantic constraint violations;
 2    IF G is semantically acceptable and not in T THEN;
 3        IF NOT G is a compound graph THEN;
 4            Submit <G,T,I> to the lookup system;
 5            IF G is UNKNOWN OR G contains generic referents THEN;
 6                Submit <G,T,I> to the deductive mechanism;
 7            ENDIF(5);
 8        ENDIF(3);
 9    IF G is a compound graph or (G is an unproved or generic simple
      graph) THEN;
10        Submit <G,T,I> to the theorem prover;
11    ENDIF(9);
12    IF G has not been proved in steps 3 to 11 THEN;
13        Submit <G,T,I> to further reasoning schemes;
```

```
14      ENDIF(12);
15          IF G has not been proved in step 13 THEN;
16              Submit <G,T,h> to the user interaction mechanism;
17          ENDIF(15);
18      ELSE(2);
19          G is either semantically unsound or G is being proved as part of its
            own proof;
20      ENDIF(2);
21      Report results;
```

This algorithm hides a considerable amount of complexity. However, it does provide a framework on which to hang the rest of the system. We will show later in this section and in the appendices how other forms of reasoning fit into this algorithm at step 13.

At several points within the components of the scheme there is a requirement to recursively enter the algorithm. Such recursive entries are made at step 3 since all necessary testing for selectional constraint violations are made on the first entry.

We can analyse this algorithm and identify an efficiency measure which it contains. Step 6 only submits graphs to the deductive mechanism if they were not proved by step 4 or if they contain generic referents. In justifying this we note that all simple graphs which do not contain any generic referents have a denotation consisting of a single graph. If this has been found at step 4 then there is not any need to try step 6. We also note that any simple graph with generic referents has a denotation of unknown size and so the proof of such a graph at step 4 does not guarantee that the entire denotation has been found.

Step 10 sends all unproved simple graphs and all compound graphs to the theorem prover. Any compound graph may also exist explicitly within the knowledge base and so could be found by a simple lookup procedure. This is included as part of the theorem prover and is an optional efficiency measure but such an explicitly known compound graph could also be proved without lookup by the use of iteration.

Each main section of the mechanism (steps 4, 6, 10, 13 and 16) returns four items:

* A truth value which represents the truth of the graph processed by that module.

* A record of the reasoning steps used in the proof.

* A trail, which will be the trail which was passed in and modified by the addition of each graph proved by the module.

* An instantiation list, which will be the instantiation list that was passed in and modified by the addition of new instantiation tuples (as described later).

In addition to these some modules will also record other information for their own use in preventing unnecessary work. The purposes of these will be explained as required.

5.7 LOOKUP

The rule of *lookup* is the first rule to be applied. For a simple graph *g* It principally consists of the simple projection of *g* onto the set *T* and projection of negated simple graphs in *F* onto *g*. The procedure adopted for this is the open world evaluation game.

In addition to finding the projective extents of a graph onto *T* and *F* it is also necessary, in the case of a projection onto the set *T*, to record instantiations of generic referents so that these may be propagated to all other generic referents on lines of identity. These are recorded during the reduce move where the projective extents are found. These have the following structure:

$$\Pi(u,T) = \{< \pi_1(u,T),\ i_1 >, \ldots , < \pi_n(u,T),\ i_n >\}$$

where each tuple represents a single projection of *u* into *T* along with *i*, the corresponding set of instantiations of generic referents made during the projection. *I* is a list of instantiation tuples each with the following structure:

$<var, [r_1, \ldots , r_n] >$

where *var* is a generic referent and the list $[r_1, \ldots , r_n]$ is the complete field with which *var* has been unified. In allowing the unification of a variable with a set of objects in this way we are allowing the system to recognise the fact that two different markers may come to represent the same individual. Thus if a knowledge base contains the concept:

[PERSON : Rosalie = Rosanne]

then the concept:

[PERSON : *x]

will unify with it. Thus, any other concept such as:

[PERSON : Rosalie = *x]

in the same context will also be a reference to a concept such as:

[PERSON : Rosanne]

An additional aspect of the lookup mechanism is the lookup of graphs from the set of deductions made by the deductive mechanism. More will be said later about how this set comes into existence but at this stage it is only necessary to state that the deductive mechanism generates complete sets of deductions and that these are stored for later reuse. The lookup section of the system uses this set as extensions to the sets *T* and *F* in addition to the graphs in the knowledge base *per se*.

Algorithm 5.5: Lookup

The algorithm for lookup is a direct implementation of the open world evaluation game algorithm given in Section 4.4.2 and will not be repeated in full.

It is worth commenting upon the efficiency of this algorithm, with particular reference to the use of the projective extent operation in the project and reduce moves. By definition the projective extent of a graph *g* is the set of all projections of *g* onto the sheet of assertion. It can either be the projective extent of *g* into *T* or of *F* into *g*. In either case the algorithm as given in Section 4.4.2 and by Sowa refers to complete projective extents. The computation of a complete projective extent of a relatively general graph in a realistic system may yield very many graphs. There are two possible consequences of this; the computation will take a long time (especially important since it may have to be done more than once) and the large number of graphs generated may require impossibly large amounts of storage.

If we consider the project move we see that we only ever use one graph from the projective extent at any one time. Therefore we ought to use a mechanism that returns a single graph (with backtracking to return the others if required). In the reduce move we can also use the same approach but we must return one graph from each of $\Pi(g,T)$ and $\Pi(F,g)$ in order to allow the detection of inconsistencies (which are signalled by both projective extents being non-empty). We can further optimise the reduce move by only allowing backtracking into $\Pi(g,T)$ to find alternative solutions in the case where $\Pi(F,g)$ is empty, since if $\Pi(F,g)$ is non-empty *g* is either false or the sheet of assertion is inconsistent and further proof does not have any meaning. The current system does not yet support these efficiency measures.

5.8 THE DEDUCTIVE MECHANISM

5.8.1 Top Level Of The Deductive System

Section 4.5.2 demonstrated the basic principles of a deductive mechanism. It is now necessary to develop a mechanism which fulfils the following criteria:

* Is deductively complete.
* Is reasonably efficient.
* Is robust and does not fall into avoidable nonterminating loops.

Deductive completeness means that any simple graph that is implied by the knowledge base can be deduced by the deductive mechanism. The term 'deduction' here is used to imply the use of *modus ponens* and *modus tollens* and the application of rules to a set of premises. Since we can also show that simple graphs may be proved without the consultation of a set of premises there is a second class of simple graphs that will not be proved by such a procedure. The following case is an example:

Given:	(*p* (*q*))	(*p q*)
Prove:	*p*	

We might attempt to take either of these rules and show that *q* is true or false. If *q* were true then the rule (*p q*) would show *p* to be false whilst if *q* were false then the rule (*p* (*q*)) would also show *p* to be false. However, there is not any direct reference to *q* in the knowledge base and another approach is required. The correct proof, assuming the previous knowledge base, is:

Prove:	*p*		
Iteration:	*p*	(*p* (*q*))	
Deiteration:	*p*	((*q*))	
Double negation:	*p*	*q*	
Iteration:	*p*	*q*	(*p q*)
Deiteration:	*p*	*q*	(*q*)
Deiteration:	*p*	*q*	( )

The empty set has been derived and therefore the addition of *p* to the knowledge base is inconsistent. A similar proof shown that (*p*) is consistent. Such simple graphs will be proved by the theorem prover to be described later. Indeed, any deduction at all could be carried out by the theorem prover but we will demonstrate that by factoring out this form of reasoning we will be able to derive a dedicated deductive mechanism which is more efficient than the equivalent theorem proving approach.

In order to develop a deductive mechanism we have already shown that it is sufficient for our mechanism to work by attempting to prove each 'antecedent' of a rule to leave a residue. This gives rise to the following simple algorithm for the deductive mechanism:

Algorithm 5.6: Deductive Mechanism

let *g* be a simple graph.

1 Make a set *S* of rules that could result in a proof of *g*;
2 Select a rule *R* from *S*;
3 **FOR** each 'antecedent' *a* in *R* **DO**;
4 **PROVE** *a*;
5 **IF** a is **TRUE THEN**;
6 There will be several sets of variable instantiations generated in its proof. Deiterate a and copy all new referents from any one set of variable instantiations to coreferent concepts in *R* to give *R'*. Add new instantiations to the instantiation list;
7 **ENDIF(5)**;
8 **NEXT(3)**;
9 The residue *D* is the set of deductions;
10 Play the evaluation game with *g* using *D* as the model;

This will be recognised as a development of a basic Prolog like engine [Campbell84] with enhancements to deal with our broader definition of the term 'antecedent'. We must look in more detail at steps 1 and 9. In addition we must address three problems: prevention of nonterminating loops, prevention of repetition of work and completeness.

5.8.2 Selection Of Rules

Since the system is not restricted to the use of a particular clausal form there are many forms that rules can take. In particular, any simple graph within a rule may be considered the 'head' of the rule. Not only that but the totally declarative nature of the system that we wish to develop means that the ordering of rules within a knowledge base is not under the control of the user and so it is not possible to bury additional procedural knowledge within the knowledge base as it is with Prolog. With these points in mind we note that there will in general be more rules available for the proof of any simple graph than there would be in a Prolog program for instance. Many of these rules will be fruitless in the long run but the system will not know this until it has tried

them. For this reason it is necessary to provide a means of ordering rules into some form that promotes the use of the most promising ones first.

In addition to optimising the order in which rules are tried it is also necessary to ensure that all relevant rules are found. These will fall into two groups:

* Rules that will show the formula to be TRUE.

* Rules that will show the formula to be FALSE.

We may feel entitled to assume (without any justification) that it is more likely that a formula be TRUE than that it be FALSE. Therefore we arrange our machinery to place the rules that will show this at the head of the list of potential rules. The selection of rules and the unification of the formula *g* with the 'head' of the rule is accomplished by the following algorithms:

Algorithm 5.7: Rule Selection

Let T_1 = the trail on entry
Let T_n = intermediate trails
Let T = the trail on exit
Let SC = the list of rules that may prove g true
Let SA = the list of rules that may prove g false
Let g be the graph to be deduced
MAKE_RULE_SET(g,SC,SA,T_1,T);

```
1     FINDALL rules on sheet of assertion in S;
2         UNIFY_EVEN_HEADS(g,S,SC,T1,T2);
3         UNIFY_ODD_HEADS(g,S,SA,T2,T);
4         IF SC and SA are both empty THEN;
5             FAIL;
6         ENDIF(4);
```

Algorithm 5.8: Production Of Rules By Unification Of *g* With Evenly Enclosed Graphs Within The Rules

This algorithm produces rules in which *g* has been unified with an evenly enclosed graph in *R*. In doing so it generates rules that may produce results from evenly enclosed graphs in *R* by *modus ponens*.

UNIFY_EVEN_HEADS(*g*,*S*,*SC*,T_1,*T*)

1 **FOR** each rule *R* in *S* **DO;**
2 Find an evenly enclosed graph *u* in *R* that unifies with *g*;
3 Copy referents from *g* to all concepts in *R* that are coreferent with concepts in *u* that have been unified with referents in *g* to give *R'*;
4 Specialise all type labels in *R'* to the proper type labels for each individual referent to give *R''*;
5 **IF** *R''* occurs on the trail **THEN;**
6 *R''* has already been generated;
7 **ELSE;**
8 Place *R''* into *SC*;
9 Place *R''* into *T1* to give *T*;
10 **ENDIF(5);**
11 **ENDFOR(1);**

The unification at step 2 is accomplished by the modified maximal join discussed in Section 4.5.2. The algorithm above backtracks at steps 6 and 9 to find possible alternatives for *R''* that have not yet been generated from *R*. Once each alternative unification has been found then the next *R* in *S* is taken.

Algorithm 5.9: Production Of Rules By Unification Of *g* With Oddly Enclosed Graphs Within The Rules

This algorithm produces rules in which *g* has been unified with an oddly enclosed graph in *R*. In doing so it generates rules that may produce results from oddly enclosed graphs in *R* by *modus tollens*.

UNIFY_ODD_HEADS(g,S,SA,T_1,T)

1 **FOR** each rule R in S **DO;**
2 Find an oddly enclosed graph u in R that projects onto g;
3 Copy referents from g to all concepts in R that are coreferent with concepts in u that have been unified with individual referents in g to give R'';
4 Specialise all type labels in R' to the proper type labels for each individual referent to give R'';
5 **IF** R'' occurs on the trail **THEN;**
6 R'' has already been generated;
7 **ELSE;**
8 Place R'' into SA;
9 Place R'' into $T1$ to give T;
10 **ENDIF(5);**
11 **ENDFOR(1);**

This algorithm possesses the same backtracking properties as the previous algorithm.

These three algorithms will together produce the correct set of rules that match the graph to be proved. As shown here they will also order the rules so that those which might result in a proof of g as being true will be tried first. There are some points that arise from these algorithms, in particular, why there should be any difference between the steps 2 in the previous two algorithms. Additionally, there is in the case of algorithm 5.8, some additional simplification of R.

Unification Of *g* With The Head Of The Rule

Algorithm 5.8 uses the modified maximal join operation at step 2 to unify g with the head of R, whereas algorithm 5.9 uses projection. To see why there should be such a difference we must consider what happens when 'antecedents' are proved, with the consequent instantiation of variables.

The generation of consequents by proof of oddly enclosed graphs will in general be performed by the projection of those graphs into the knowledge

base. This may result in the instantiation of generic referents by universal instantiation. Such instantiations will be passed to the remainder of *R*. Since the individual referents that have been added to *R* may conform to more specialised types than were present in the original generic concepts the type labels in the head of the rule may become more specialised as the proof progresses. This specialisation may result in a head onto which *g* will now project even though this was not the case with the 'raw' rule. Thus, the selection and modification of rules on the basis of projection of an evenly enclosed graph onto *g* will not in general be complete. We will illustrate with the following example:

```
Given:      [ ELEPHANT : Clyde ]
            ( [ ELEPHANT : *x ]
                  ( [ MAMMAL : *x ]->( PART )->[ TUSK ] ) )
            ELEPHANT < MAMMAL
Prove:      [ ELEPHANT : Clyde ]->( PART )->[ TUSK ]
```

We can see that the query will not project onto the evenly enclosed graph within the rule and so this method of unification will result in the rule not being selected although the proof of the query with this rule ought to be trivial. In this case we see that the evenly enclosed graph will actually project onto the query, but there is no sound reason why this means of unification should be adopted since if the query was:

```
[ ANIMAL : *x ]->( PART )->[ TUSK ]
```

and it was known that MAMMAL < ANIMAL then the unification would again fail erroneously. By using the modified maximal join operation for the unification we ensure that either unification will succeed, even if the query and evenly enclosed graphs consist of more than one relation, each at different relative levels of specialisation.

Algorithm 5.9 uses projection to perform the unification. In the case where the head of the rule is oddly enclosed the projection of the head onto *g* is sufficient because the generation of any false graph does not imply that all or any of its generalisations are also false. Thus, if *g* is to be shown false by *modus tollens* it must be a specialisation of an oddly enclosed head. For example:

Given: ([MAMMAL : Colin])
([ELEPHANT : *x]
([MAMMAL : *x]->(PART)->[TUSK]))
ELEPHANT < MAMMAL
Prove: [ELEPHANT : Colin]

The proof proceeds as follows:

Rule: ([ELEPHANT : Colin]
([ELEPHANT : Colin]->(PART)->[TUSK]))
Proof of 'antecedent': ([ELEPHANT : Colin]->(PART)->[TUSK])
Result: ([ELEPHANT : Colin])

This proof does not show the details of the proof of the 'antecedent', but this is done as a straightforward specialisation of a known false graph. Suppose that the unification were performed by Sowa's maximal join and the query was:

[ANIMAL : Colin]

The proof would be:

Rule: ([ELEPHANT : Colin]
([MAMMAL : Colin]->(PART)->[TUSK]))
Proof of 'antecedent': ([MAMMAL : Colin]->(PART)->[TUSK])
Result: ([ELEPHANT : Colin])

The result is that Colin is not an elephant, which does not prove the query about Colin being an animal but which has taken an amount of time equal to the proof of the query that Colin was an elephant and yet the proof was bound to fail. It is for this reason that the projection operation is used for the unification step in this case.

Further Simplification Of Rules

In the case where the query was unified with an evenly enclosed graph we can make a further simplification. We note that the context in which the evenly enclosed graph occurred may contain other graphs which were not unified with *g* and will not therefore provide any solutions. Since they are evenly enclosed we may remove them by erasure. If we leave them in we do not alter the result

of the proof but we may spend time unnecessarily proving them.

5.8.3 Handling The Results Of A Deduction

Following the maximal proof and removal of the 'antecedents' of the rule, a residue, which represents a deduction, remains. In order to complete the proof of the original graph *g* it remains to attempt to prove *g* against the deductions. Any projections that are found at this stage represent solutions.

The way in which this works takes account of the fact that either the original query may have been a simple graph or a negated simple graph and the result could also be either a simple graph or a negated simple graph for each case of query. In other words, the result of a deduction is a set of graphs which are all true and which may be either simple graphs or negated simple graphs. Since the result of a deduction is a small knowledge base then the evaluation game may be used to prove the query against the deductions. Thus the final phase of a deductive proof involves simply passing the original query to the evaluation game along with the sets of deductions, which act as the knowledge base.

Special Case

There is one case in which the deduction mechanism may generate information that can lead to a proof but which does not generate a proof directly within the list of deductions. As an example we consider the following:

Given:

[P : 1]->(R1)->[Q : 2]->(R2)->[S : 3]

([P]->(R1)->[Q : *x] ([A : *x]))

When the query:

[A]->(R2)->[S]

is submitted to the system as described the reply is UNKNOWN. This is because the rule, whilst it can (and will) generate the concept [A : 2] will not generate the required relation. Instead it is necessary to add the new information to the knowledge base and reprove the original query. In this case

the reproof of the query will find a solution by lookup at the second attempt. In general, this process may be required at several levels within the proof of a complex query and so the mechanism to perform the reproof is simply a call to the whole proof procedure at times when the above situation is detected (*i.e.* when the proof of a relation generates a single concept result).

5.8.4 Nonterminating Loops

Since one of the objectives of the present work is to produce a fully automated declarative reasoning system it is necessary for that system to provide safeguards against trivial situations that lead to nonterminating loops. To see how such loops may arise we shall consider as an example the use of the following rule, which expresses the symmetry of some relation REL:

```
( [ T : *x ]->( REL )->[ T : *y ]
        ( [ T : *x ]<-( REL )<-[ T : *y ] ) )
```

When the deductive mechanism is required to prove the formula:

```
[ T : #1 ]->( REL )->[ T : #2 ]
```

the following rule will be constructed:

```
( [ T : #2 ]->( REL )->[ T : #1 ]
        ( [ T : #2 ]<-( REL )<-[ T : #1 ] ) )
```

This rule contains the antecedent [T : #2]->(REL)->[T : #1]. In the case where this graph is not represented explicitly in the knowledge base it will be presented to the deductive mechanism, which will subsequently generate the following rule:

```
( [ T : #1 ]->( REL )->[ T : #2 ]
        ( [ T : #1 ]<-( REL )<-[ T : #2 ] ) )
```

Now, the antecedent of this rule is identical to the original graph that is being proved. Unless steps are taken this antecedent will again enter the deductive mechanism. If this is allowed to occur a nonterminating loop has been created.

From the above example we can see that in order to prevent loops such as this we require the deductive mechanism to fail if any graph is submitted to it

at any later stage within its own proof. In order to prevent this situation we must record which graphs are currently being proved. This is the function of the trail. The trail is a list of graphs which currently have an incarnation of the deductive mechanism associated with them. Each time a graph is submitted to the deductive mechanism we must first check to see if it already exists on the trail and if it does then further processing is prevented.

Placing Modified Rules Onto The Trail

The loop detection mechanism prevents loops developing in the proofs of antecedents of a rule by preventing the proof of any graph as a part of its own proof. We can extend this idea to include the prevention of the use of rules which are exact copies of rules already being evaluated. This will also help to prevent loops and will increase efficiency by making checks at a point before that at which antecedents are submitted to the proof mechanism.

5.8.5 Repetition Of Work

During the course of any deduction it is possible, on backtracking, for any particular simple graph to be generated by instantiations caused by the proof of other simple graphs and that this graph is identical to another graph that has already been proved. In such cases there is not any point in submitting the graph to the deductive mechanism again as its entire denotation will have already been found. It should only be necessary to look up the answers that were generated previously. The obvious benefit of this is a great increase in speed.

For this reason, CGP maintains a list of simple graphs that have been generated by the deductive mechanism. Before any graph is submitted to the deductive mechanism this list is consulted as part of the lookup form of proof. Should there be an entry for the graph then this is used and the deductive mechanism is blocked.

The success of this approach depends on the completeness of the deductive mechanism. It must be guaranteed that once any graph has been submitted to the deductive mechanism then all possible solutions have been generated and stored. In addition, it also requires that any graphs which are proved in this way are not more general than the graph whose proof originally

generated the deductions.

5.8.6 Completeness Of The Deductive Mechanism

The deductive mechanism provides a system of *modus ponens* and *modus tollens*. Since there is no restriction as to what is regarded as the 'head' of a rule and what are its 'antecedents' completeness from the point of view of rule selection, and the ability to perform both *modus ponens* and *modus tollens*, follows immediately. In addition, we must also show that the proof of any 'antecedent' is itself complete in order to show that the mechanism as a whole is complete.

To do this we need only observe that the proof of any 'antecedent' is performed by submitting it to the whole proof mechanism (not simply the deductive mechanism). Thus its proof will involve all steps. It only remains to show that the final step, the theorem prover, is itself complete.

5.9 THE THEOREM PROVER

The theorem prover uses the five rules of inference that were developed in Chapter 4 and it attempts to reduce any formula to either { } or ( ). All the rules, except for iteration, remove information and guarantee that the proof moves nearer to the solution. In addition, we must assume that a backtracking mechanism exists to allow for the possibility of failure of any branches of the proof to close (we do not discuss this in the present work since the prototype is written in Prolog and Prolog's backtracking mechanism is used). Because of the possibility of excessive backtracking (and combinatorial explosion if iteration is not controlled) it is necessary to organise the theorem prover into a form which tries the most likely rules first and which optimises the proof into a form which minimises the need for backtracking. In addition it is possible to make some stringent restrictions to the application of the most potentially expensive rules and thereby keep their effects under control.

5.9.1 Organisation Of The Theorem Prover

For reasons of efficiency it is necessary to identify which rules are likely to move nearer to a solution than are others and to apply these first. In the previous sections we identified three rules that resulted in the removal of

information from a formula:

* Deiteration.
* Removal of double negations.
* Empty clause elimination.

We can make the following observations about a typical formula *g*:

* If *g* contains ( ) then any other graphs of any complexity within the same context as ( ) can be immediately removed by empty clause elimination.
* The operation of deiteration will tend to produce derivatives which contain ( ).
* Deiteration from simpler negated contexts will generate ( ) more quickly than it will from more complex negated contexts.
* The generation of ( ) at shallow depths may allow empty clause elimination to remove larger portions of the formula than would be the case if ( ) were generated more deeply.
* The operation of removal of double negations will only produce ( ) from ((( ))), which in turn is likely to have been produced by a series of deiterations.

From these points we can conclude that it is probably most efficient to apply these rules in the following order:

* Deiteration (from less deeply nested contexts).
* Removal of double negations.
* Empty clause elimination.

It is not possible to state that this ordering is always the most efficient since proofs may take any one of a number of directions. Nevertheless, the above ordering is empirically supported by long experience of manual proofs.

These rules are sufficient by themselves to prove many formulae (and will prove *all* formulae whose proof does not depend upon the contents of the knowledge base) and such proofs will tend to be rapid. For completeness, the

rule of deiteration must take into account graphs within the knowledge base when attempting to deiterate a simple graph. Each of these simple graphs is subjected to the whole proof system and therefore their proof may be more expensive than the 'simple' case of deiteration. For this reason we split the rule of deiteration into two parts: deiteration against graphs within the formula and deiteration against graphs within the knowledge base.

We have shown previously that for the theorem prover to be complete with respect to the knowledge base we also require restriction and iteration. The following points apply to these rules:

* Restriction - this rule adds information in the form of specialisation of type labels but it does not add new relations or logical connectives and therefore does not add more than one step (the restriction step itself) to the proof. With the use of this rule the nature of any proof that uses it is more like a simpler propositional proof than otherwise.

* Iteration - this rule adds a compound graph from the sheet of assertion to a context within the formula being proved. In doing so the size of the formula has increased and several additional steps, possibly including further iterations, will be needed. We note that the rule of iteration is only necessary to provide the possibility of deiteration of otherwise nondeiterable graphs. Therefore we need only try iteration on the smallest residue which remains after the proof and deiteration of as much of the formula as possible.

With these two points in mind we observe that the rule of restriction is likely to simplify proofs and is never going to prevent a proof, so we should actually perform this rule first of all. The rule of iteration potentially adds a considerable amount to the proof and should therefore be tried last, following which it should not be tried again until all other options have failed.

Taking into account the effect of restriction and the splitting of deiteration into two, we can now give the sequence in which the theorem prover uses its rules:

* Restriction.
* Deiteration against graphs within the formula.
* Removal of double negations.
* Empty clause elimination.
* Deiteration against the knowledge base.
* Iteration.

We now turn to the precise details of these six rules and show how they can be applied to enable the reasonable efficient generation of proofs in our methodology.

5.9.2 Restriction

The rule of restriction, when applied to each concept in a formula, forces the formula into a state in which it is guaranteed that all concepts on some line of identity are either as specialised as allowed by the conformity relations table or as specialised as the most specialised dominating concept. The result of this is that any possible deiterations whose success depends upon the dominating concepts being sufficiently restricted are guaranteed to succeed since those concepts will exist in their most restricted state.

As an example, we wish to prove the trivial case:

([P : 1] ([Q : 1]))

where Q < P and Q :: 1. In this contrived case (which might result after a series of steps in a bigger proof) the knowledge base states that Q :: 1 and we know that Q :: 1 $\Rightarrow$ [Q : 1] and that [Q : 1] $\Rightarrow$ [P : 1], so the inner concept of the formula could be removed by deiteration against the knowledge base. But we observe that this deiteration of [P : 1] would need to be accomplished as a result of a full proof of the concept, which (although it would be accomplished by simple lookup in this case) is relatively inefficient. Instead we wish to restrict the formula to:

([Q : 1] ([Q : 1]))

which we can do for two reasons, such restriction within an oddly enclosed context is always logically sound by insertion and, also, we know the relation

Q :: 1. Once this has been done we are in a position to deiterate the inner copy of [Q : 1] directly from within the formula, a process which does not require a complete proof but, rather, a simple comparison with a dominating concept. This proof now concludes with a empty clause elimination. Thus the use of restriction has enabled this proof to proceed in a more efficient way.

Completeness Of Restriction

To show that the rule of restriction does not prevent any proof that would otherwise have occurred we must consider three points:

* The restriction of any type label which takes place as the result of the presence of a more specific type label in a dominating context will only result in a type label which is as specific as the dominating one and not more. Therefore any deiteration is still possible.

* The restriction of any type label as the result of consulting the conformity relations table will not prevent any deiteration against a graph in the knowledge base since the knowledge base is maintained in its most restricted form and all deiterations will still take place.

* All concepts within the formula that contain a particular individual marker will be restricted and so any deiterations from within the formula will still be possible since all restricted concepts will still be dominated by a concept of at least the same specificity.

Since we have shown that any proof that could have proceeded before a restriction will also proceed afterwards we can also state that there is no need for any backtracking at this point.

5.9.3 Deiteration Against Graphs Within The Formula

The rule of deiteration allows the deiteration of both graphs and lines of identity. Since a goal driven proof will require the eventual removal of all graphs and lines of identity we can say that any graph *g* which could be the result of an iteration and which contains lines of identity, all of which could have resulted

from the same iteration, can be deiterated along with the lines of identity in one operation. All lines of identity extending from *g* to graphs dominated by *g* must also be removed if they could also be the result of iterations (or oddly enclosed).

We combine the deiteration of graphs and lines of identity since the deiteration of any graph must be accompanied by the prior removal of lines of identity as these cannot be left dangling. The simple removal of lines of identity without the accompanying removal of a graph does not add anything to our logic because:

* Any line of identity which extends to an evenly enclosed graph can only have arisen as the result of an iteration of the graph from which it originates. Such an evenly enclosed graph can therefore be deiterated without first removing the lines of identity in a separate act of deiteration.

* Any oddly enclosed graph can be deiterated along with its attached lines of identity since all such graph and lines of identity could have arisen from insertions and are therefore implied by the result of the deiteration.

Caveats

* The deiteration of any graph *u* cannot proceed if it is connected by a line of identity to a more deeply nested evenly enclosed graph *v* which could not have arisen from an iteration of *u*. To do so would be to use the rule of erasure which we have already shown to be unsound for a goal driven approach.

* The deiteration of an evenly enclosed graph *u* cannot proceed if it is connected by a line of identity to a less deeply enclosed graph *v* from which *u* cannot have arisen by iteration. To do so would again be to use the rule of erasure.

Efficiency Of Deiteration Against Graphs Within The Formula

The deiteration of graphs at random causes combinatorial explosion. To see why we can consider the following simple example:

(*p q* (*p q r*))

In order to prove this formula we must attempt to deiterate all three evenly enclosed graphs to give:

(*p q* ( ))

from which stage the proof is trivial. However we note that the situation is contrived to fail by the presence of *r* which cannot be deiterated. A naive system first attempts the proof by the following steps:

Start:	(*p q* (*p q r*))
Deiteration of *p*:	(*p q* (*q r*))
Deiteration of *q*:	(*p q* (*r*))
Failure	

It then attempts the following:

Start:	(*p q* (*p q r*))
Deiteration of *q*:	(*p q* (*p r*))
Deiteration of *p*:	(*p q* (*r*))
Failure	

Both proofs are equivalent and only one ought to have been attempted therefore. The situation deteriorates rapidly when there is more scope for deiteration, as in the case where there are three graphs that are candidates:

(*p q r* (*p q r s*))
(*p q r* (*q r s*))
(*p q r* (*r s*))
(*p q r* (*s*))
Failure

(*p q r* (*p q r s*))
(*p q r* (*q r s*))
(*p q r* (*q s*))
(*p q r* (*s*))
Failure

(*p q r* (*p q r s*))
(*p q r* (*p r s*))
(*p q r* (*r s*))
(*p q r* (*s*))
Failure

(*p q r* (*p q r s*))
(*p q r* (*p r s*))
(*p q r* (*p s*))
(*p q r* (*s*))
Failure

(*p q r* (*p q r s*))
(*p q r* (*p q s*))
(*p q r* (*q s*))
(*p q r* (*s*))
Failure

(*p q r* (*p q r s*))
(*p q r* (*p q s*))
(*p q r* (*p s*))
(*p q r* (*s*))
Failure

In general, the branching factor for the search space is factorial on the number of possible deiterations at each stage. This is unacceptable. Fortunately the solution to this problem is straightforward. It is based upon the fact that having deiterated any graph there cannot be any further deiterations that are now not possible that were possible before. To see why we consider the following:

(p^1 (p^2 (p^3))) where p^{n+1} could be the result of an iteration of p^n.

In this case each p^i $(i>1)$ can be deiterated because of the presence of p^{i-1}. But we also note that deiteration is transitive. This means that if we first deiterate p^2 to give:

(p^1 ((p^3)))

we can still deiterate p^3 because of the presence of p^1. Thus there is no need to

regard a deiteration as a branch point in the search space and backtracking is not necessary. When backtracking is not allowed we see that each possible proof is only carried out once.

Completeness Of Deiteration From Within The Formula

To show that the system does not loose completeness when backtracking after deiteration is forbidden we note that for a formula to be provable the outermost copy *g* of any graph can only have arisen from either an insertion into an oddly enclosed context or an iteration from the knowledge base. If the former is true then the rule of empty clause elimination will allow *g* to be removed and if the latter is true then *g* will be deiterated by proof against the knowledge base. Thus *g* can always be removed.

There is a special case to consider in the light of the fact that the system splits all graphs into separate relations. The following example shows the possible problem that this produces:

```
( [ p ]->( r1 )->[ q : *x ]                                   {depth 1}
        ( [ p ]->( r1 )->[ q : *y ] [ q : *y ]->( r2 )->[ s ]  {depth 2}
                ( [ p ]->( r1 )->[ q : *z ] [ q : *z ]->( r2 )->[ s ]  {depth 3}
                )
        )
)
```

In this case either relation *r1* in depths 2 and 3 can be deiterated against that at depth 1. Also, that at depth 3 can be deiterated against that at depth 2. Suppose the relation *r1* at depth 3 was deiterated, without backtracking, against that at depth 1. This would leave:

```
( [ p ]->( r1 )->[ q : *x ]                                   {depth 1}
        ( [ p ]->( r1 )->[ q : *y ] [ q : *y ]->( r2 )->[ s ]  {depth 2}
                ( [ q : *z ]->( r2 )->[ s ]                    {depth 3}
                )
        )
)
```

However, this would result in the variable **z* at depth 3 being matched with the variable **x* at depth 1. Thus the remaining **z* at depth 3 could not match the **y*

at depth 2 and the deiteration of the relation *r2* at depth 3 could not occur. Yet it is clear (without showing the derivation) that the whole formula reduces to:

([*p*]->(*r1*)->[*q* : **x*])

In order to ensure that the correct deiteration takes place it is necessary to rejoin all graphs before attempting any deiteration. When this is done the formula becomes:

```
( [ p ]->( r1 )->[ q ]                                  {depth 1}
    ( [ p ]->( r1 )->[ q ]->( r2 )->[ s ]               {depth 2}
        ( [ p ]->( r1 )->[ q ]->( r2 )->[ s ]           {depth 3}
        )
    )
)
```

It is now clear that any graph can only be deiterated against the correct dominating graph and the result is now correct. There may, however, be cases where it is possible to deiterate only a part of some graph against a graph within the formula, the remainder requiring deiteration against the knowledge base. In such cases we note that all graphs will be split back into separate relations and so any relations that require such proof will be correctly proved, the remaining deiterable portions having been detached and made available for deiteration against relations in the formula.

5.9.4 Removal Of Double Negations

The removal of double negations follows deiteration and removes any double negations that have been created by deiteration. In this way the formula is kept as simple as possible.

Efficiency Of Removal Of Double Negations

We observe the following point:

(((*p*)) ((*q*)))

This formula can have its double negations removed in two different ways:

(((*p*)) ((*q*)))

(*p* ((*q*)))

(*p q*)

or:

(((*p*)) ((*q*)))

(((*p*)) *q*)

(*p q*)

Once again we see that the same result is obtained either way and that the branching factor of the search space at this point is greater than necessary. As with the case of deiteration we see that the branching factor of any point in the search space where several removals of double negations are possible is factorial on the number of double negations present. We must therefore ensure that the system does not backtrack after making a removal of double negations.

Completeness Of Removal Of Double Negations

The removal of double negations can never reduce the depth of nesting of any graph *p* to a level that is less than that of a graph *q* such that *p* could have been deiterated against *q* before the removal of double negations but cannot be deiterated afterwards. To see why we note that the graph *p* must have been either in the same context as *q* or in a context dominated by *q* and surrounded by a double negation. In the first case any removal of double negations that reduced the depth of nesting of *p* would also have reduced that of *q* by the same amount and both *p* and *q* would still be in the same context. In the second case the double negation around *p* must have been in the same context as *q* and therefore the removal of double negations around *p* could only have reduced the depth of nesting of *p* to the point where *p* was in the same context as *q*. Thus the removal of double negations never prevents a previously possible deiteration from taking place.

There are further implications concerning the rule of iteration. These are dealt with in Section 5.9.6.

5.9.5 Empty Clause Elimination

The rule of empty clause elimination allows large parts of a formula to be removed in one operation. Any context which contains ( ) can be emptied of all other graphs of any form.

Efficiency Of Empty Clause Elimination

It is possible to conceive that a series of deiterations may produce a formula of the form:

(*p q* ... ( ) ( ))

Thus it is possible to use empty clause elimination with either empty clause. It is not necessary to do so since each empty clause is identical in appearance and the results of using either are the same. As a result it is not necessary to backtrack and all such branch points are removed.

Completeness Of Empty Clause Elimination

To show that backtracking is not necessary with this rule we note that all graphs removed by this rule would have to have been removed by some other rule in order to generate a proof. The only other rule which could do this is deiteration. But we note that, because all deiterable graphs have been deiterated (against graphs in the formula) by the time that empty clause elimination is used, any graph left in the residue is the least deeply nested version of the graph. Therefore the removal of graphs by the rule of empty clause elimination does not remove any graph that dominates any graph in the residue. As such it does not prevent any future deiteration of any graph in the residue that could lead to a proof.

An additional point arises when iteration is considered. The purpose of iteration is the 'resolution' of otherwise nonremovable graphs. In this case we note that the introduction of any graphs by iteration must be followed by deiterations if there is to be any progress. We could therefore argue that the removal of any graph *u* by empty clause elimination might just prevent such a deiteration. However, thus is not the case since the removal of any graph *v* by iteration requires an iteration into the same context as *v*, a context dominated

by *u*. But the use of empty clause elimination to remove *u* must also have removed *v* in any case so such an iteration will never be necessary.

5.9.6 Deiteration Against The Knowledge Base

As stated in Section 5.9.3, it is possible to deiterate graphs against graphs in the knowledge base. There are a number of subtleties associated with this form of deiteration. In particular we must pay special attention to the matching of generic referents with referents in the knowledge base, especially where there are lines of identity involved. There are four cases to consider when attempting to deiterate a graph *g* from. We will first define the term *independent graph*:

Definition 5.10: Independent Graph

A graph *g* is *independent* in a formula *F* if *g* does not contain any coreference links to any evenly enclosed graphs in *F*.

Proof

We have already shown that any line of identity that extends to an oddly enclosed graph may have arisen by insertion or iteration and can therefore be removed. In addition, any line of identity that extends to an evenly enclosed graph can only have arisen by iteration. Thus the deiteration against some graph *v* of such an evenly enclosed graph is dependent upon all its lines of identity being derived by iteration of *v*. Any evenly enclosed graphs which does not contain lines of identity is therefore independent of any other graph. Any other graph that is not linked to an evenly enclosed graph by lines of identity is also independent.

Cases Of Deiteration Against The Knowledge Base

We start by outlining each case of the deiteration of a graph *g* from a formula *F* and then follow with a discussion of important theoretical issues.

I *g* is oddly enclosed and independent. In this case *g* may have arisen from either an iteration or an insertion. If *g* was iterated then its proof will allow its deiteration. If *g* was inserted then the only way to remove it is by empty clause elimination. This will always be sufficient as discussed previously.

II *g* is oddly enclosed and is not independent. In this case if *g* is false it may be deiterated since the truth of *F* only depends upon the truth of its evenly enclosed graphs. This point is shown by the standard truth table for implication. We observe that all evenly enclosed graphs in the original *F* that had any lines of identity emanating from *g* can only have come by iteration of *g*. Since we remove such graphs and lines of identity at an early stage in the proof of *F* then any that still remain could not have resulted from iteration and we must look for concrete examples from the sheet of assertion. We return to this in a moment. Any instantiations of generic referents that were found during the proof of *g* must be propagated to all graphs that are dominated by *g*.

III *g* is evenly enclosed and independent. In this case *g* can only have arisen from an iteration and possible erasures and if it is either true or false it can be deiterated.

IV *g* is evenly enclosed and not independent. In this case, if *F* is a tautology, *g* can only have arisen from an iteration of the less deeply nested graph with which it is linked by lines of identity. However, because *g* is still present in *F* following the use of deiteration from within the formula it cannot have been placed in *F* by such iteration. Therefore the only way in which *g* can be removed is if it is false, otherwise there will remain dangling lines of identity and the possibility of spurious proofs.

The explanations of these four cases suggest that *g* can be deiterated if it is true or false but fail to indicate what happens when a graph turns out to be false. In such cases the graph is replaced with a single cut. We see that the deiteration of a true graph actually proceeds in a similar manner except that the

graph is (effectively) replaced with an empty positive context { }. Thus we see that the following examples might occur in each case:

Case I

Given:	(*p*)
Prove:	(*p* (*q*))

We see that, in a proof from { }, the oddly enclosed (*q*) can have arisen by insertion into the already known (*p*) but can only be removed in a goal driven proof by empty clause elimination. However, the graph (*p*) already exists and so the oddly enclosed *p* can be replaced by ( ) to give:

(( ) (*q*)).

An alternative proof would be:

Prove:	(*p* (*q*))
Double negation:	(((*p*)) (*q*))
Deiteration of (*p*):	(( ) (*q*))

but we do not actually need to add double negations in this way.

Case II

Given:	p'
Prove:	(p^{*x} (q^{*x}))

In this case we see that the formula contains two graphs, including an evenly enclosed graph, which are linked in some way by a line of identity *x. Therefore neither graph is independent. The rules above allow the deiteration of p^{*x} if it is false followed by instantiation to give:

((q'))

The graph q' is now independent.

As a further example we consider:

Given:	(*p*)
Prove:	(p^{*x} (q^{*x}))

In this case we know that there are no *p*s at all and so replace the oddly enclosed p^{*x} with ( ) to give:

(( ) (q^{*x}))

This allows us to conclude that *F* was true in this case since false implies true.

Case III

Given:	q'
Prove:	(p (q))

In this case we may deiterate the evenly enclosed *q* to give:

(p ( ))

and the proof is complete.

Case IV

This case allows the derivation ($p \Rightarrow$ FALSE) $\Rightarrow$ FALSE:

Given:	p' (q')
Prove:	(p^{*x} (q^{*x}))
Deiteration:	((q'))
Deiteration:	((( )))
Double Negation:	( )

Thus the evenly enclosed graph can be deiterated, even though is is not independent, on the basis that it is false.

Efficiency Of Deiteration Against The Knowledge Base

Deiteration against the knowledge base is entirely dependent upon the efficiency of the system as a whole since this form of deiteration involves a recursive call to the whole proof mechanism. Thus the more efficient the system is in general the more efficient will be deiteration against the knowledge base. We have already discussed the efficiency of most of the other parts of the system.

The backtracking of deiteration of graphs against the knowledge base has the same restrictions as that for deiteration against graph in the formula.

Completeness Of Deiteration Against The Knowledge Base

To show that deiteration against the knowledge base is complete we only need to show that the remaining parts of the system are complete since any, otherwise unprovable, simple graph within *F* will be subjected to this form of deiteration at some stage. It is the purpose of other sections of this chapter to demonstrate this. In addition the general form of this form of deiteration is the same as that of deiteration against graphs in the formula and the completeness of this other form has already been considered.

5.9.7 Iteration

The rule of iteration is used to add logical relationships to a formula and thus allow possible deiterations and empty clause eliminations in the future. It is used when all else has failed since it is the one rule which may cause an increase in execution times.

Efficiency And Completeness Of Iteration

The rule of iteration adds new information and is therefore prone to increasing the complexity of the proof. In its simplest form we can see that any proof that requires an indeterminate amount of iteration would soon run into an infinite loop if certain measures were not taken. In particular we can see that we might attempt to iterate the same graph repeatedly. Fortunately it is possibly to guarantee that any proof that involves iteration will terminate. To do this we must consider ways of identifying when it is necessary to use the rule and placing suitable restrictions on its use.

Simple Graphs: We start by noting that it is not necessary to iterate simple graphs in order to allow future deiterations and empty clause eliminations because simple graphs can never add any negated contexts to the formula. All simple graphs within the original formula will be removed by one form or other of deiteration. The rule of iteration should therefore only be used when no more simple graphs can be removed from the formula by any other means. As an example:

Given: q

Prove: $(p (q))$

1)

Iteration: $(p\ q (q))$

Deiteration: $(p\ q ( ))$

Empty clause elimination: $(( ))$

2)

Iteration: $(p (q\ q))$

Deiteration: $(p (q))$

Deiteration: $(p ( ))$

Empty clause elimination: $(( ))$

3)

Deiteration: $(p ( ))$

Empty clause elimination: $(( ))$

The first proof shows that the inner copy of q can be deiterated against the iterated copy of q from the knowledge base. The second proof shows that the iteration of q can be followed by an immediate deiteration to restore the original formula. This is then proved by deiterating the evenly enclosed q against its copy on the knowledge base, an operation that could have been carried out without the original iteration, as proof 3 shows. Thus we can see that any simple graph that can be deiterated after an iteration of a copy from the knowledge base could always have been deiterated without the initial iteration.

Relevant Graphs: In order to make a useful iteration we observe that there is no point in iterating any graph which does not contain some graph that is connected (by way of generalisation) with some graph already present within the formula (we will use the term *marker graph* for a graph in the original formula). This must be the case since otherwise there would not be any possibility of subsequent deiterations. It is therefore only necessary to iterate a graph which contains a graph that could be deiterated after the iteration has taken place. Such a deiterable graph is any graph that projects onto a marker graph in the formula. Thus:

Given: (*p* (*q*)) (*s* (*t*))

Prove: (*p* (*r*))

In this case there can never be any point in iteration the graph (*s* (*t*)) since this will not lead to any deiterations and will only require a large amount of further work (a further iteration of the same graph) to remove it again.

Enclosure: Any graphs in common between the graph to be iterated and the formula need only be oddly enclosed within the iterated graph. This is so since we are attempting to add odd numbers of negations to the original formula so that the rule of empty clause elimination can eventually be used. By iterating graphs which contain relevant graphs which are oddly enclosed we are generating a situation of the following form:

Given: (*p* (*q*))

Prove: (*p* (*q*))

Iteration: (*p* (*p* (*q*)) (*q*))

We can now deiterate *p* and (*q*) to give (*p* ( ) (*q*)) which leads to a proof. If we had iterated a compound graph in which the relevant graph had been evenly enclosed then we might have created a single negation which was oddly enclosed with respect to the iterated graph. This would simply have lead to the elimination of the iterated graph without any change to the original formula.

Depth Of Iteration: Any iterated graph need only be iterated into the context that contains the marker graph. To see why we observe that it is the marker graph that we are trying to eliminate. If the graph is iterated into a context which is less deeply nested than the marker graph then the equivalent graph in the iterated graph cannot be deiterated and the iteration has not gained anything. If the iteration is into a more deeply nested context than the marker graph, or into a context not dominated by the marker graph, then, although some deiteration may happen to be possible we will not eliminate the marker graph as intended. The only way to eliminate the marker graph is to iterate into the same context as the marker graph.

Given:	(*p* (*q*))
Prove:	(*p* (*q*))
1)	
Iteration:	(*p* (*q* (*p* (*q*))))
Deiteration:	(*p* (*q* ((*q*))))
Deiteration:	(*p* (*q* (( ))))
Double Negation:	(*p* (*q*))
Failure	
2)	
Iteration:	(*p* (*p* (*q*)) (*q*))
Deiteration:	(*p* ((*q*)) (*q*))
Deiteration:	(*p* ( ) (*q*))
Empty clause elimination:	(( ))

The first proof fails because the whole of the iterated graph is removed in the subsequent steps. This is prevented in the second proof by iterating into the same context as the marker graph. The second proof also demonstrates that the marker graph is oddly enclosed within the iterand. In this particular case either *p* or (*q*) can be regarded as the marker graph.

Repeated Iteration: Each graph need only be iterated once for the removal of any one marker graph. To see why, we need only recall that the rule of iteration is only used when the formula cannot be reduced by any other rule of inference. Therefore each simple graph within the formula only ever either occurs once within the formula or once in each disjoint part of the formula (in which case, as soon as one part of the disjunction has been proved true it all has). We have already shown that it is only ever necessary to iterate into the same context as the marker graph. Therefore a repeated iteration can never add any more useful information. This restriction on iteration is sufficient to guarantee that all proofs will terminate, no matter how much iteration is required. It is not possible, however, to guarantee that termination of a complex proof which uses a large knowledge base with many potential candidates for iteration will occur very rapidly. In order to do that we would need to ensure that there was a possible chain of iterable graphs that would ultimately connect the marker graph with some other graph in the formula. In other words, we would require to almost

know the answer in advance. The following examples show how nothing further is gained by repeated iteration of some graph:

Given:	(*p q*)
Prove:	(*p* (*r*))
Iteration:	(*p* (*p q*) (*r*))
Deiteration:	(*p* (*q*) (*r*))

1)

Iteration:	(*p* (*p q*) (*q*) (*r*))
Deiteration:	(*p* (*q*) (*q*) (*r*))
Deiteration:	(*p* (*q*) (*r*))

2)

Iteration:	(*p* (*q* (*p q*)) (*r*))
Deiteration:	(*p* (*q* (*q*)) (*r*))
Deiteration:	(*p* (*q* ( )) (*r*))
Empty clause elimination:	(*p* (( )) (*r*))
Double Negation:	(*p* (*r*))

These examples show that nothing is gained by the second iteration of (*p q*) in each case.

5.9.8 Algorithm Of The Theorem Prover

The theorem prover is a recursive function. It has as input parameters the formula *F* to be proved, the trail, the current trace and the instantiation list. At the start of the function a check is made to determine if a result has been reached. This is the case if *F* is either { } or ( ). In this case the function exits and returns the appropriate truth value (TRUE if *F* = { } or FALSE if *F* = ( )) along with the current trail, trace and instantiation list. Next, the system must ascertain that *F* is not on the trail. If it is then a loop has been entered and that branch of the proof must terminate in failure.

Following the checks the rules of inference are applied in the manner described above and in the order given in Section 5.9.1.

Algorithm 5.11: Organisation Of The Theorem Prover

Let F_1 = the formula to be proved;

Let F_2 = the formula returned by a rule of inference;

Let T_1, T_2 = the trail;

Let A_1, A_2 = the trace;

Let I_1, I_2 = the instantiation list;

THEOREM(F_1,T_1,T,A_1,A,I_1,I) =

```
1   IF F1 = { } THEN;
2       RETURN TRUE, T = T1, A = A1, I = I1;
3   ELSE IF F1 = ( ) THEN;
4       RETURN FALSE, T = T1, A = A1, I = I1;
5   ELSE IF F1 is in T1 THEN;
6       FAIL;
7   ELSE;
8       RESTRICT(F1,F2);
9       IF F1 <> F2 THEN;
10          RETURN THEOREM(F2,T1,T,A1,A,I1,I);
11      ELSE;
12          DEITERATE_FORMULA(F1,F2,A1,A2);
13          IF F1 <> F2 THEN;
14              RETURN THEOREM(F2,T1,T,A3,A,I1,I);
15          ELSE;
16              DENEGATE(F1,F2,A1,A2);
17              IF F1 <> F2 THEN;
18                  RETURN THEOREM(F2,T1,T,A3,A,I1,I);
19              ELSE;
20                  ELIMINATE_DISJUNCTIONS(F1,F2,A1,A2);
21                  IF F1 <> F2 THEN;
22                      RETURN THEOREM(F2,T1,T,A3,A,I1,I);
23  ELSE;
24  DEITERATE_SA(F1,F2,T1,T2,A1,A2,I1,I2);
25      IF F1 <> F2 THEN;
26          RETURN THEOREM(F2,T2,T,A3,A,I2,I);
27      ELSE;
28          ITERATE(F1,F2,T1,T2,A1,A2,I1,I2);
```

```
29                          IF F1 <> F2 THEN;
30                              RETURN
                                    THEOREM(F2,T2,T,A3,A,I2,I);
31                          ELSE;
32                              FAIL;
33                          ENDIF(29);
34                      ENDIF(25);
35                  ENDIF(21);
36                      ENDIF(17);
37                  ENDIF(13);
38              ENDIF(9);
39          ENDIF(1);
```

The algorithm tries each rule of inference in the order specified previously. The deiteration and iteration steps each require backtracking, as discussed, since some branches may reach a dead end, with some graphs remaining.

5.10 LAST RESORT

The final attempt at proving a graph is one which involves the user of the system in a direct conversational dialogue. For any graph that has passed through the previous stages and is still UNKNOWN the system interacts with the user to find out if he/she can resolve the difficulty. At this point it is worth mentioning that the system might be interacting with another similar system which has different knowledge and that this is therefore the point of contact between several processors each conversing with each other.

To illustrate the last resort method of proof we consider the following example which depicts a rule specifying (very simply) the symptoms of the disease poliomyelitis, which can be confused in its early stages with influenza for which a rule is also given:

```
Given:      ( [ PERSON : *x ]-
                ( SYMP )->[ FEVER ]->( ATTR )->[ RECUR ]
                ( [ PERSON : *x ]<-( SUFF )<-[ POLIO ] ) )

            ( [ PERSON : *x ]->( SYMP )->[ FEVER : *y ]
                ( [ FEVER : *y ]->( ATTR )->[ RECUR ] )
                ( [ PERSON : *x ]<-( SUFF )<-[ INFLUENZA ] ) )

            [ PERSON : John ]->( SYMP )->[ FEVER ]

Prove:      [ PERSON : John ]<-( SUFF )<-[ DISEASE ]

Rule:       ( [ PERSON : John ]-
                ( SYMP )->[ FEVER ]->( ATTR )->[ RECUR ]
                ( [ PERSON : John ]<-( SUFF )<-[ POLIO ] ) )

Lookup of [ PERSON : John ]->( SYMP )->[ FEVER ] :
                ( [ FEVER ]->( ATTR )->[ RECUR ]
                ( [ PERSON : John ]<-( SUFF )<-[ POLIO ] ) )

Last resort for [ FEVER ]->( ATTR )->[ RECUR ] :
                FALSE (say)
Record:     ( [ FEVER ]->( ATTR )->[ RECUR ] )
Rule:       ( [ PERSON : John ]->( SYMP )->[ FEVER : *y ]
                ( [ FEVER : *y ]->( ATTR )->[ RECUR ] )
                ( [ PERSON : John ]<-( SUFF )<-[ INFLUENZA ] ) )

Lookup of [ PERSON : John ]->( SYMP )->[ FEVER ] :
            (   ( [ FEVER : *y ]->( ATTR )->[ RECUR ] )
                ( [ PERSON : John ]<-( SUFF )<-[ INFLUENZA ] ) )

Lookup of ( [ FEVER : *y ]->( ATTR )->[ RECUR ] ) :
                ( ( [ PERSON : John ]<-( SUFF )<-[ INFLUENZA ] ) )

Double negation:
                [ PERSON : John ]<-( SUFF )<-[ INFLUENZA ]
```

Thus, neglecting the oversimplification in this example, the system has been able to rescue the proof by requesting extra information as required. With this

form of proof the means to implement expert systems which show some intelligence with respect to finding additional information is possible.

5.11 UPDATING THE KNOWLEDGE BASE

Since the system is intended to accept new information and add this to its model there are various considerations to be made in order to maintain consistency and validity:

Nature Of The Model:

The model is currently very simple since there is no provision for alternative models or nested models. There are two reasons for this:

* The objective of the system is to show that a sound reasoning system with a single model represented by conceptual graphs is viable.

* Sowa's conceptual graphs theory does not provide adequate support for multiple models and nested models and it has not been the aim of the present work to provide such support.

These two points greatly simplify the maintenance of the knowledge base. There is no need to maintain contradictory knowledge (as is the case, for instance, with the truth maintenance system [Doyle79]) since the use of a single model, which is required to be consistent, enforces restraint in this respect - any new graphs that contradict the current model are simply rejected.

Addition Of Simple Graphs

In order to maintain the open world assumptions, simple graphs are added to the knowledge base by attempting to join them to existing graphs with the join operation. The strict join operation requires that the join be performed on identical concepts. However, it is imaginable that a new graph may contain an individual concept *c* whose type label is a known supertype of the proper type of *referent(c)*. In this case the join would fail. There are two ways around this:

* Change the join operation so that it joins on identical referents, specialising one of the type labels if necessary.

* Automatically specialise the type label of each concept in the new graph to the proper type for each referent.

Either of these approaches will work. We adopted the second as the primary means of adding knowledge and we now discuss some corollaries.

In addition to the above it may also be useful to contract the knowledge base (by type and relational contraction) in order to provide the following:

* Reduced memory consumption.

* Quicker retrieval of contracted forms.

The first of these follows from the fact that by contraction a large graph may be reduced to a single concept or relation. The second follows since the operations of contraction are performed once and for all when new knowledge is added. Since contraction has already been performed it will not be necessary for proofs to be performed when a query requests such information in the contracted form. CGP performs such contractions and it is for this purpose that a lambda abstraction equivalent form of each type definition is retained; the system can retrieve and use this form slightly more efficiently than the logical equivalence form in this case.

Specialisation Of Type Labels

Any simple graph within the knowledge base is likely to contain concepts with individual markers. Each of these markers will also possess an associated conformity relation to its proper type. In order to guarantee that the projection operation will locate any individual marker it is necessary to ensure that each individual marker is associated in the concept in which it occurs with its proper type. Thus, the type labels of all individual concepts in the set *T* are maintained at their most specialised level. In doing this we guarantee that the join of new graphs following their specialisation (if the joining of new graphs is done that way) will always work.

Another aspect of this is the possibility that, for some existing referent *r* of type *u*, there will arise a new concept with referent *r* of type *v* such that it is not known that either *u* < *v* or *v* < *u*. In such a case it is necessary to accept that *u* :: *r* and *v* :: *r*. Thus there exists a type *u_v* which is a common subtype of *u* and *v* such that *u_v* :: *r*. An example might be the case where it is known that:

ELEPHANT :: Clyde

and it later becomes apparent that:

PET :: Clyde

In this case it is necessary to induce the following relations:

PET_ELEPHANT < PET
PET_ELEPHANT < ELEPHANT
PET_ELEPHANT :: Clyde

and to adjust the knowledge base accordingly.

Adding Other Formulae

Compound formulae are added to the knowledge base without further processing except that any lattice relations that are implied by them are extracted. Such lattice relations will be present in the case where the new compound graph represents a type definition or a relational definition.

There is a requirement for subsumption testing at this stage since any new compound graph may subsume an existing graph. In this case the existing graph will not imply the new graph and so the new graph will not have evaluated to true (or false) and so will be unknown. However, the new graph may still imply an existing graph. In such a case it is necessary to remove the existing graph in order to reduce search spaces in the future.

Subsumption of compound graphs can be tested for by projection with the enhanced projection operation that we have already described. If some compound graph *u* projects onto some compound graph *v* then *v* subsumes *u*. Where *u* already exists in the knowledge base it can be removed without any loss of knowledge. The situation is exactly the same for any simple graphs *u*

and *v* in which we wish to retain the graph which contains the most information.

Deletion Of Knowledge

The system described in this work is a monotonic reasoning system which can optionally record its deductions for later use. As such, the deletion of knowledge is potentially dangerous unless a careful record is maintained of how graphs which are possibly consequents of that knowledge arose. This is especially a problem when the system is operating so that the results of deductions are stored for the future.

The problem of deletion highlights the need for any comprehensive system to be able to maintain separate alternative models and to be able to evaluate graphs within these models. It is not adequate in the long run to completely dismiss graphs which are currently thought to be contradictory. Psychologically this is not realistic for the following reasons:

* It will be necessary in the future to be able to deal with uncertain knowledge.

* It will be necessary in the future to be able to make predictions by use of a theory.

* It will be necessary in the future to build alternative theories to explain a set of facts.

* It will be necessary in the future to be able to process nested contexts in general.

These points illustrate the fact that the deletion of knowledge is really the creation of an alternative model in which that knowledge has a different truth value. When humans change their mind they do not completely and literally forget what they previously believed but they remember their previous beliefs.

For the CGP system, rather than create alternative models, a facility to edit the statements made to the system is incorporated. It depends upon a record being kept of all assertions made to the system. When invoked, the mechanism for this takes each recorded statement in turn and in reverse order from their initial submission, evaluates them and then adds them to the new knowledge base as appropriate. In this way it is possible to contradict a previous statement

and rebuild the knowledge base on the basis of the new statement. The original contradicted statement will now be rejected as being itself a contradiction.

5.12 EVALUATION OF CGP

The prototype was developed along with the theory and was subjected to evaluation at two levels:

* Informal evaluations by colleagues who tested the implementation by attempting to use it.

* More formal evaluations by colleagues who verified both the theoretical notions and their embodiment in a more rigorous and systematic manner.

5.12.1 Informal Evaluation

Evaluations of this kind were conducted by many of the undergraduate students and the results have been reported in many undergraduate documents. The general opinion has always been one of interest but lack of experience in logic has resulted in some misunderstanding of the output from the system. Nevertheless, these informal evaluations have exposed problems with the implementation that might otherwise have passed unnoticed.

5.12.2 Formal Evaluation

The system has been more formally evaluated at two stages of its life:

* At a very early stage by Bokkers [Bokkers92].

* At a more advanced stage by Lukose [Lukose93].

5.12.2.1 Bokkers

Since the theory and system were very incomplete Bokkers was able to find many problems with the implementation. Nevertheless, the basic philosophy, that of CGP providing a completely declarative environment, was present from the start and is recognised by Bokkers where he states that it is very easy to work with the system once the basics of Conceptual Graphs theory are known.

As he points out, there were many bugs in the program. These existed mainly because the system was not at that stage intended to be a full working production system and for the purposes of the present work it was an adequate thinking tool.

Bokkers also commented on the lack of efficiency of the system. Much of this lack stemmed from the incompleteness of the theory and hence many of the domain independent heuristics and efficiency measures later discovered and incorporated. Many of the suggestions that he makes concerning this problem were already part of the research and have since been adopted.

We have stated that future systems must be capable of more than pure first order reasoning and Bokkers was of the same opinion. In particular, he was concerned about the removal of knowledge from a knowledge base as a system developed. We return to this theme in Chapter 6.

5.12.2.2 Lukose

The evaluation by Lukose was performed at a much later stage in the development of both the theory and the system. At this point the system had already been subjected to many informal tests and many implementational problems had been removed. Some of the efficiency measures had been identified and implemented and the system was altogether in a more usable form since it was now in general use.

Lukose provided a thorough comparison between the theory as presented by Sowa and that as implemented in CGP. He attempted to show that each of Sowa's operations could be performed within the system. Certain difficulties were experienced since, because of the declarative nature of CGP, the individual operations were not made available for testing. Indeed, many of Sowa's operations were not implemented as such, as we have stated. Thus it was necessary for Lukose to emulate certain operations. This he managed to do.

Although the system was much more advanced by this time there were still omissions and errors and Lukose found several. Nevertheless, he found that in many cases several incorrect results all resulted from a single error and the overall system was reasonable robust. All the problems that Lukose found were

solved and the solutions incorporated into CGP.

Lukose was also interested in the overall approach of the system and in particular its ability to fully utilise negation. In verbal communication he compared CGP with the system of [Lukose92] used by himself and was able to express an opinion that CGP had many advantages over other systems. These were advantages of semantic correctness (with respect to negation in particular) and completeness. The fully declarative nature was also a strong point.

5.13 CONCLUSIONS OF CHAPTER 5

In this chapter we have built upon the theory presented in previous chapters and have shown how to construct a working system from those building blocks. In doing so we have also shown how to optimise the use of the rules of inference and how to direct the progress of a proof. Such direction of proofs by guiding them all towards the same goal, { }, ( ) or some irreducible residue, has provided us with domain independent heuristics. We have also shown how to guarantee that any proof in our system will terminate. Thus we have established the effectiveness of the goal driven approach. Since our logical basis is also a syntactic one we have shown how to derive reasonably efficient proof procedures without the need for the system to possess any insight into how a proof must proceed.

Although the present work does not address the point it is also probable that further improvements in efficiency will be made at such a time that the theory of Conceptual Graphs is developed to the point where a natural scheme of case based reasoning is available.

The system has been used and tested by a number of people. Although problems occasionally come to light the response of such people is generally favourable both to the use of conceptual graphs as a knowledge engineering formalism and to the use of CGP as a conceptual graph processing engine.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 INTRODUCTION

In this chapter we will summarise the work described previously and draw conclusions showing where the work has improved both the representational and computational aspects of the theory. During this summary we will indicate where we have introduced new theoretical points.

We will also discuss the prototype system from the points of view of efficiency and usability and we will demonstrate that the prototype has shown that, at least for small knowledge bases, even a relatively crude implementation in Prolog can be very effective. In the future we envisage that the system would be implemented in fast, efficient modern languages and run on fast machines and that in this way any residual implementational problems will be removed.

Finally, we will discuss areas of work yet to be carried out into extending the theory to accommodate other forms of reasoning in a consistent manner. We will show that in anticipation of this future work we designed our rational reconstruction to contain certain features. Nevertheless, there are certain unanswered questions concerning the nature of human reasoning and these have consequences for future enhancements to the representation of and logical operations on conceptual graphs.

6.2 SUMMARY OF THE RATIONAL RECONSTRUCTION

The rational reconstruction of Sowa's Conceptual Graphs theory that we presented in Chapters 2 and 3 represents the outcome of a process inspired by many hours of attempting to use the theory for representing knowledge and performing computations with conceptual graphs. Those attempts lead to a realisation that the theory would benefit from some revision on two levels; representation and computation. Indeed, most workers seem to accept the

theory as it stands but then find that they are compelled to introduce special cases into their systems. What we intended to show in this thesis was a consistent means of addressing many of the commonly found difficulties.

We believe that the effort to reconstruct the theory was worthwhile since we are now able to produce a computational model and a working prototype for a theorem prover based upon the rational reconstruction. This is the first time that any such system, which is not simply a notational variant of some other formalism, has been built out of Conceptual Graphs theory.

We will consider the areas of representation and computation separately.

6.2.1 Representation

It seemed that, despite its richness (or because of it), there was some ambiguity within the notation of conceptual graphs and indeed some aspects were simply wrong. We will recapitulate these points and summarise the ways in which they showed how the original theory was flawed. In doing so we will be summarising and drawing conclusions from the aspects of the theoretical parts of this work that are new since no-one has yet made anything other than a very limited attempt to suggest any changes.

We considered every aspect of the notation and made the following recommendations:

* A greatly reduced range of referents.

* Isomorphism of field types.

* Additional special notation for the semantically more correct means of representation of negation, the cut.

* Rationalisation of the notation of definitions, which we argued to be special cases of the more general implication.

* That the lattices and conformity relations table are not a fundamental part of the notation but are useful abstractions as implementational features.

6.2.1.1 Referents

Sowa's list of kinds of referent was very comprehensive in his 1984 book. Since then it has grown even larger as he and others have added new referent types as situations have arisen in which the existing ones appeared inadequate. We showed that many of these referents contained information that ought to be more explicit (part of the type definition for the individuals represented by the referent) since there is no general means of extracting it from referents of some arbitrary form that may arise in the future. We now summarise the problems and the solutions that we identified with a selection of examples.

Measure Referent

We saw that a referent such as:

@ 2 miles

can only be meaningful if either a language of measure referents, with an associated logic, is developed or the expanded form (equivalent to Sowa's measure expansion) is employed exclusively. Similarly, we argued against name referents since the role of NAME is a relationship between some entity and some symbol.

We argued that the removal of 'complex' referents from the theory represents an advance in expressiveness, and not a reduction, since the equivalent, 'full' form makes all relationships explicit in a manner which is consistent with the general nature of Conceptual Graphs theory. Not only that but if any mechanisms for dealing with different kinds of referent were ever to be proposed they would undoubtedly suffer greatly from computational complexity problems.

Set Referent

We also argued that some kinds of referent were not only cumbersome but semantically dubious. Possibly the most dubious was the set referent in which we argued that a 'normal' concept of the form:

[T : #x]

had its semantics founded upon the notion of the representation of a single entity (#x) of type T. Sowa, however, also allowed the set referent which permitted the following concept:

[T : { #x, #y, ... }]

where T is the same type as before. It is now ambiguous to claim that #x is of type T when it could be the case that the whole set is an entity of type T. Sowa claims that the former is the case but then goes on to allow graphs into the referent. Such graphs are referents of type PROPOSITION. If Sowa's view of a set referent be adopted the following concept says that *p* and *q* are individual PROPOSITIONs:

[PROPOSITION : { *p*, *q* }]

But we observed that a set of graphs represented a conjunction of those graphs. In this case we do not know if it is intended that the referent contains the separate PROPOSITIONs *p* and *q* or whether the conjunction *p.q* is intended.

In order to overcome difficulties with the set referent we adopted the approach of Kocura and replaced it with an alternative treatment in which the relationships of each set member to the complete set were represented explicitly by conceptual relations.

Quantifiers

Sowa and others often use the universal quantifier in the referent field. We argued that the use of the universal quantifier within the referent was at best ambiguous (since its scope could not be shown) and at worst logically inconsistent with the semantics of the concept box. As a result we recommended a logical representation for the universal quantifier. This representation is made especially easy by our introduction of a Peircean notation for negation.

Even though the generic referent is implicitly existentially quantified one sometimes sees the existential quantifier used explicitly within the referent field. Whilst this may be semantically acceptable since the generic concept is existentially quantified it is rather unnecessary.

6.2.1.2 Isomorphism Of Field Types

We allowed each field of a conceptual graph (type, relation and referent) to contain exactly the same kinds of object. This lead to a consistent and computable means of expressing higher order statements. Until recently this subject had been either ignored or treated in complex ways by Sowa and others. Sowa [Sowa93] introduced two new operators, τ and ρ, to be placed before a marker in either the type field or the relational field respectively to show that those markers were type or relational labels. Thus Sowa can now state the following:

[SPECIES : ELEPHANT] [ELEPHANT : Clyde]

and ask the following question:

[SPECIES : *x] [τx : Clyde]

We argued that it was not clear why there must be a special operator (τ). Indeed, in the display form it was especially difficult to see why it was necessary since we defined that the line of identity terminated within the referent field of one concept and within the type field of another concept and thus its semantics are unambiguous:

Thus, in the linear form we would write:

[SPECIES : *x] [*x : Clyde]

which has exactly the same semantics. We used this form in Chapter 1 having first introduced this notation and semantics in the CGP User Guide [Heaton92] and also in [Heaton91].

6.2.1.3 The Cut

Sowa attempted to produce a notation that was consistent and lean. However, he failed to recognise that the semantics of negation and its central

position as a logical primitive required that it not be treated in the same manner as general modalities, since it is not such a modality, and that it be given a unique notation. We showed that the use of the monadic NEG relation was logically flawed in more than one way and therefore introduced an alternative, more Peircean form of cut. This approach had not been attempted before. The results that followed from the adoption of a simple enclosure to show negation were:

* Negation was given its correct place as a logical primitive.

* It was no longer possible to alter the truth value of a context by the rule of erasure.

* The standard operations did not require special cases when a monadic relation happened to be NEG or a type label happened to be PROPOSITION.

* The cut negated the proposition within it, rather that denying that the nested graph of a concept of type PROPOSITION was a proposition.

* It was possible to define projection over negated contexts in a manner consistent with Peirce logic and which mirrored and preserved the workings of 'ordinary' projection of 'ordinary' concepts.

* The implementation of the cut and associated Peirce logic operations within the prototype was simplified.

6.2.1.4 Definitions

We argued that definition by equivalence allowed a more unified approach to proof since in this form a definition is simply a compound graph like any other 'general' compound graph. Nevertheless we also recognised that a suitable set of lambda abstractions, one for each definition, could be used to implement a discrimination net and to provide a means of contracting any context to its most contracted form. This contracted form is that in which graphs are stored and to which graphs are converted before evaluation.

6.2.1.5 Lattices

The lattices are abstractions derived from definitions. We showed that a suitable data structure for conceptual graphs representation in a computer would automatically generate a lattice structure. We also showed that the prototype system used lattices to speed up the operation of projection since heavy use is made of the information within them. A comparison of possible arrangements of lattices was made and we showed that a lattice that contained links from some type to all its supertypes would substantially speed up a system with the cost of some increased memory use.

6.2.2 Computation

Sowa provided three sets of computational operations:

* Canonical formation rules.
* Operations of expansion and contraction.
* Operations of logic.

He did not, however, provide much in the way of how he envisaged these rules to be combined into an integrated system. Before we could develop a computational model we felt that we needed to address the definitions of many of Sowa's operations. This need resulted mainly from some ambiguity in the presentation of Sowa's rules, in particular in his rendering of Peirce's rules. As an example of possible problems with Sowa's exposition of Peirce logic we highlighted a potential problem with derivations carried out using linear notation and display notation.

6.2.2.1 Canonical Formation Rules

In the rational reconstruction we showed how to distil from Sowa's canonical formation rules a set of rules which retained the completeness without the complexity which is associated with an over-large set of rules. This set of rules was conceived from the notion that some of Sowa's procedures were simply particular sequences of Peirce logic and that by including the procedures explicitly within a working system we would be creating the possibility of duplication of work.

During the course of the reconstruction we clarified the operations of join and maximal join. We showed that the join operation was a special case of the maximal join operation in which the join proceeded between two concepts with the same referent. This was a change from Sowa's definition which permitted the join only on identical concepts and was necessary to allow our knowledge base to fully conform to the open world requirements.

6.2.2.2 Operations Of Expansion And Contraction

We argued that, and showed how for each form, a Peirce logic expression for definitions provided the means whereby the full power of the rules of inference was automatically made available for computation on these objects. In doing so we showed that every expansion and contraction operation that could be performed with a lambda abstraction could also be carried out in Peirce logic. This meant that we could remove the explicit operations of expansion and contraction without loss of computational power.

6.2.2.3 Operations Of Logic

This area was the one where we were able to overcome many of the problems that are encountered when attempting to produce computationally complete reasoning systems with any logic. We stated that the possible problems were related to tractability since, without some form of control, it was not possible to guarantee termination of a proof. We also argued that we preferred the control of a proof strategy to be domain independent since it was the possibility of this characteristic that we were attempting to show.

In developing our logic we compared the four basic approaches to proof in any formalism and we linked the discussions to the properties of conceptual graphs and Peirce logic. As a result we concluded that, for conceptual graphs, the preferred approach should be a non-uniform, goal driven strategy.

Having defined the strategy we went on to show that Peirce's rules of inference were inadequate since they were data driven in nature. We then showed that in order to produce a goal driven calculus we needed to derive the appropriate set of complementary rules. This we did and in showing the validity of each we had regard to the computational complexity properties of each.

With the complementary set of rules of inference we had provided a set of rules that would make it possible to implement a goal driven system in (a form of) Peirce logic. We also claimed that this was the first time that such an adaptation of Peirce logic had been made and that this finally opened the door to the possibility of a working system which was computationally complete but yet tractable. We also argued that proofs would terminate in a reasonable time, although some proofs might take longer than average.

6.3 SUMMARY OF THE MODEL AND THE PROTOTYPE

By the use of our rules we have been able to develop a system with the following properties:

* A mechanism for verifying that all new graphs could be generated by the canonical formation rules from a set of canonical graphs and are therefore well formed.

* A deductive mechanism (that subsumes that of Prolog) which employs the more realistic open world assumption.

* A small engine based upon the set of derived rules of inference which implements a reasonably efficient goal driven proof procedure.

* A system which can be guaranteed to maintain its own consistency by rejecting new information which it can show to be false with respect to the current knowledge base.

6.4 FUTURE WORK

There are several aspects of the theory of conceptual graphs that suggest the directions of future work. We will first consider some aspects of the prototype where further work would improve its performance. This discussion will show the general principles involved in processing conceptual graphs. Following that we will attend to some important theoretical issues which must be addressed if conceptual graphs are ever to become a serious formalism for modern knowledge representation and processing.

6.4.1 Efficiency Of Computation

Most of the computational model has been shown to be reasonably efficient, with generally not worse than linear execution times in the prototype which improve dramatically for those parts which could be handled by the use of efficient tree structures instead of Prolog's lists.

Graph Retrieval

The lookup mechanism would be improved beyond linearity by suitable indexing or compilation. This would be particularly important where the system was built upon large database systems where access times were orders of magnitude greater than rapid memory accesses.

Schemes for improving the speed of graph retrieval must take the following points into account:

* Unless a knowledge base is deductively closed (unlikely since such a database may be prohibitively large or even infinite) then any graph consisting of more than one relation may not exist explicitly in its entirety although some parts might be.

* Any query graph might be set at any level of abstraction and different parts of it may be at different levels of abstraction. Thus there must be a means of preventing fruitless projections where a more abstract part or a query is first projected onto a graph which cannot satisfy the more specialised parts of the query.

* Any query graph may contain subgraphs that are more or less contracted than the potentially corresponding part of the knowledge base. Thus it is necessary to ensure that the query graph and knowledge base are fully contracted.

* For any query graph which is more abstract than any graph in the knowledge base there may be a vast number of possible matches. Projective extent procedures ought not attempt to follow the strict theoretical route by returning every projection

but should backtrack as required.

* The type lattice ought to be used to narrow the search space by rejecting attempts to project relations within query graphs onto fruitless relations in the knowledge base.

The prototype largely ignores all these points with the result that graph retrieval is probably the most expensive operation.

Computation

The deductive mechanism and most parts of the theorem prover use algorithms to reduce the search space and derived rules to shorten proofs. Nevertheless, although the system is guaranteed to terminate, the rule of Iteration can still lead to efficiency problems. This results from the fact that extra information is being added to a formula and as such it is inevitable that more work, including possible further iterations, will be necessary. Further work on the efficiency should minimise this problem by reducing the options on which graphs can be iterated and where they can be iterated to (as the system already does up to a point). However, as the problems of complexity are a property of logic itself it will probably have to be accepted that there will always remain some kinds of situation in which any system will be slow.

Apart from work on algorithms, much work remains to be carried out into internal representations of conceptual graphs. That this is the case is borne out by the informal observation that different implementors produce many different designs, each with the particular needs of their view of the theory in mind. Each of these is necessarily optimised to that particular need. Since the theory is not yet complete it still remains to produce an efficient data structure that can stand modification in the future.

6.4.2 Completeness

Whilst the system and the theory are logically complete (programming errors notwithstanding) with respect to first order logic, it is nevertheless incomplete with respect to many other forms of reasoning. Such gaps include the inability to represent uncertainty in a consistent way or to represent alternative models. Until these topics can be fully addressed the theory of conceptual graphs will lag behind other formalisms in which such concepts are

easily handled.

Sowa argues that conceptual graphs are equivalent to first order predicate logic. If this is the case then it must be possible to treat logics other than first order as one would in FOPL. However, this is not yet possible since Conceptual Graphs theory does not adequately support operations on nested graphs.

6.4.3 Open Versus Closed Worlds

We argued strongly for the adoption of the open world assumption and as a result we adopted a three valued logic. The use of the open world assumption produced a system whose conclusions are much more realistic than those of a closed world system since it is not generally considered by humans that something which is not known by them is necessarily false.

Nevertheless, there are situations in which the closed world assumption is meaningful. Such a situation exists where the possession of an attribute necessarily precludes the possession of another attribute. There appears to be two ways in which this can be viewed:

* All closed world relations can be defined by laws.
* There is a kind of relation which is intrinsically closed world in nature.

If the former is true then we must expect either of:

* A very large set of laws, one for each closed world relation.
* A single law which expresses the closed world nature of such relations and which recognises that such relations are instances of EXCLUSIVE relations.

Intuitively, the first option appears incorrect since all these laws would be of the same general form and it also seems pragmatically incorrect since all these separate laws would have to be stored. The second option appears to be more appealing and recognises the general nature of exclusive relations. Nevertheless, the idea that some relations are intrinsically closed world in a way that does not even require the single general law for its expression seems even more appealing.

This result requires further investigation since the discovery of general principles would enable a system to adopt either a closed world assumption or an open world assumption whenever appropriate and the chosen assumption could be 'hard wired' into the system. Such hard wiring would lead to very rapid computation of denotations for exclusive relations.

6.4.4 Explanation And Theory Building

The current system has a very limited ability to build a theory (and as such is only documented as an appendix) and answer queries of it. These limitations are caused by two factors:

* The current work is not primarily aimed at solving this problem (although many aspects of the rational reconstruction are designed to take this problem into account for the future).

* The theory of Conceptual Graphs is not yet strong enough to provide the necessary equipment in terms of representation or rules of inference.

We now deal with the second of these points.

6.4.4.1 Representation

The basic representation of theories has been dealt with but we must consider some consequences that follow from it. We recall that we stated that a theory is a subtype of the more general modal type PROPOSITION which takes as its referent a context. This context potentially consists of all components of a knowledge base if we admit that any part of a knowledge base can be expressed as a conceptual graph. This must be so since we wish to perform the operation of projection between two modal concepts by evaluating the referent of one against that of the other. Thus a PROPOSITION is a nested knowledge base. Furthermore, the contents of a modal concept only have a truth value in relation to that context and not to the global context. Thus the concept [PROPOSITION : { *p* }] simply states that *p* is a PROPOSITION and does not imply that *p* is actually true. Indeed, the graph (*p*) does not contradict the graph [PROPOSITION : { *p* }].

Thus we see that any knowledge base may contain arbitrarily many knowledge bases nested within it in the form of modal concepts, each of which can contain further modal concepts nested within it. Each modal concept can represent either some derivative of its global knowledge base or it may be completely contradictory to it, or even to itself. We must consider the ways in which these knowledge bases arise and how they are handled within an automated reasoning system.

6.4.4.2 Generation Of Nested Knowledge Bases

The simplest form of generation of nested knowledge bases and modal concepts in general is by addition to the knowledge base in the normal way; by being informed of the information. Thus, the statement 'it is written in holy scriptures that ...', when entered into CGP in the normal manner would add the appropriate modal concept in the same way as any other concept.

There is a second way in which nested knowledge bases can arise; by their generation in the system from an act of creativity, insight or intuition. When this occurs the result is a modal concept which represents some synthesis of knowledge within the global knowledge base. Such a synthesis might occur when pieces of evidence are put together to form a model to account for them (theory generation), or when an existing model is accounted for by showing that it could have been generated from previously isolated pieces of evidence (explanation). Since this section of the thesis is concerned with future work we are now discussing incompletely formulated material. For this reason we will not consider the specific case of explanation generation any further but will continue with some general considerations of theory generation. Appendix I contains an example of a solution to a 'real' problem using the approach to be described.

Theory Generation

In a sense, the generation of theories is allied to the generation of possible worlds. By this we mean that, from some current model of the world it is possible to combine knowledge in ways that are plausible but not necessarily true. This also means that there must be methods of generating such theories from a set of known facts and which would be part of an accessibility relation. Not only that but we also wish to be as general as possible and find an

accessibility relation which will generate all theories from a knowledge base in a domain independent way. This does not preclude the possibility that the accessibility relation will not have scope for including domain dependent aspects but it does mean that such domain dependent aspects will follow in some domain independent way from the knowledge base. If such an accessibility relation exists it will provide the most general possible means of generating theories.

From the point of view of the purely domain independent part of the accessibility relation we see that the generation of theories ought to proceed by a suitable combination of basic conceptual graphs operations. We also see that these operations will not be only those of Peirce logic since these can only construct graphs which are implied by a knowledge base and whose explicit statement therefore adds nothing new. We are therefore looking at other means of generating graphs and making assumptions that did not previously exist. Since all graphs fall into one of two categories, simple graphs and compound graphs, we will see that there will be two ways in which new graphs can come into existence:

* By combining simple graphs to give other simple graphs which contain relations that did not previously exist.

* By generating new rules by an inductive means.

The generation of new simple graphs involves the combination of graphs in a way that generates graphs which do not follow logically from their precursors. All graphs which do follow logically cannot be considered as new in any sense and will still follow logically in the theory. The rule within Conceptual Graphs theory that is capable of combining graphs in a non-logical manner is the rule of maximal join. We discussed this rule in Chapter 3 and adopted a slightly amended definition to allow the join to operate on single concepts in addition to relations. This amendment made the rule completely general with respect to the graphs that can be generated by it.

We also discussed in chapter 2 the semantics of the generic referent and concluded that we should adopt the view that the generic referent is a Skolem variable. With this semantics we are able to take a step which allows us to treat all generic referents at depth 0 as individual markers (in the Skolem sense) that

represent some other individual marker (in the sense of a # marker) but we do not know which one. In other words, a concept such as [PERSON : *] states that some PERSON exists but we do not yet know which # marker to assign to this PERSON.

When we take into account our semantics of the generic referent we feel compelled to adopt a view that states that all generic referents at depth 0 are searching for some sort of instantiation. The system is constantly attempting to find the correct # marker for each * marker. It may be that several * markers share the same # marker or it may be that the # marker for a particular * marker does not yet exist within the system (although the individual in the 'real' world must exist since there is a concept for it within the system). By combining graphs with the maximal join (which need not be truly maximal since it is not the case that only the maximal case is possible) we make assumptions about the true identity of generic markers within the system. By repeatedly joining graphs we generate a model which accounts for as many generic markers as possible, hopefully all of them. Should any fully joined model still contain generic markers then there exists in the real world further unheard-of individuals. Not only must instantiations of generic referents be found but also the possibility that two different individual markers turn out to represent the same real world individual must be considered. The maximal join operation does not take this into account at all and yet the realisation that where there was originally thought to be two different entities there is actually only one may be important in some situations. An example of this may be the murder case where a person has witnessed two murders apparently by different people but later realises that both murders were the same person in different guises. There will in general be several different ways of combining graphs, each being a theory.

The second means of generating new graphs, rule induction, again proceeds by making assumptions from the particular to the general. In this case we scan the knowledge base for instances of various kinds and if we see that certain things (say *p* and *q*) always (in our experience) occur together we may generate a rule that states that (*p* (*q*)). We may go so far as to say that *p* causes *q*. In doing this we generate a theory that relates instances of one type to instances of another. Nevertheless, such rules do not follow inevitably and may later be proved false. But the rule allows predictions to be made. As long as the rule stands up to testing it is accepted as a law.

Thus we have seen two ways in which theories can be generated from an existing world in a completely general, domain independent way. There is still more work to do on the results of the generation phase of theory building since some of the joins may result in graphs which can be shown to be canonically unsound or even empirically false. Thus each new simple graph generated by a maximal join must be tested for selectional constraint violations and each new simple graph and compound graph must be tested for logical consistency. New models that fail either of these tests are not theories with respect to the world from which they were generated and must be rejected.

We have just highlighted two problems. The first relates to what constitutes a canonical graph in some world. The question is 'do all theories that are accessible from some global world necessarily share the same canon?'. If the answer to this question is 'yes' then the problem vanishes but if it is 'no' then the tests for selectional constraint violations by a new graph must proceed by the use of the correct canon. The second question relates to the logical soundness of some theory in which the laws of logic are different. If the laws of logic are universal then there is not any problem but if they are not then how do we represent the alternative laws? Certainly it would not be possible to represent them by conceptual graphs with their ordinary semantics in any way that would make them actually work since the implementation of thus represented rules must be accomplished by the logic of the global world.

Inheritance

The previous paragraph highlighted a possible problem with the canonical basis used in the testing of a new theory for semantic correctness. The question becomes a more general question of what is inherited by a theory. Certainly it seems that a theory inherits individual markers and generic markers from its parent. If the canonical basis of a theory is the same as (or an augmentation of) that of the parent then it would seem that a canonical basis is also inherited. This is not to deny that it may subsequently be modified. The theory might also inherit its parent's type and relational hierarchies.

There are some problems with inheritance. If some modal concept describes a fictional story, science fiction for example, in which anything may happen then is it the case that its type hierarchy can contradict that of the

parent or that its canon may be different? These questions must be answered before a definitive treatment of this subject can be completed.

6.4.4.3 Reasoning With Modal Concepts

Having derived the means of generating theories it is necessary to be able to reason with the content of the theory in order to draw the conclusions that follow from it. At a more general level it is necessary to be able to reason with the contents of arbitrary modal concepts for the same reason. When this is possible the system may then work out the conclusions of different models and would be operating a realistic decision making procedure. Thus it would be possible for a system to reason with what it believed were the beliefs of others and, subject to a suitable means of recording what the system itself used to believe, with that which it used to believe itself. We have here the basic requirements of a properly founded non-monotonic system and a system that would be capable of changing its mind without needing to completely reorganise and verify the whole of its belief structure each time.

In order to bring this about it is necessary to develop the theory of Conceptual Graphs with respect to the representation of modal concepts and the operation of projection with respect to these. We touched upon this subject in Chapter 3.

6.4.4.4 Combining Knowledge Bases

The final problem with modal concepts is concerned with their joining. It is reasonable to suppose that when one creates some form of modal concept that it is not fixed for all time at that point. This is particularly the case when reading a story since the story may be put aside for a time and returned to later. In this case the remainder of the story is added to the previous version. Similarly, if a modal concept concerning belief about someone else's knowledge is generated and at some later stage some more insights into that person's knowledge are gained they must be added to the original modal concept.

If one possesses modal concepts about the beliefs of two different people and these concepts happen to be identical at some point then can they be joined? If they are then when at some later stage it is learned that one person believes one thing whilst the other believes its converse then the two modal

concepts must separated again.

6.4.5 Fuzzy Reasoning

The area of fuzzy logic is one which is gaining wider acceptance as being a more realistic approach to human-like reasoning and is becoming more prominent in the forms of neural nets and simulations of neural nets on digital machines. It is claimed that the capacity of fuzzy reasoning systems to learn and adapt is greater than that of other logics and that the similarity to human 'common sense' reasoning is marked.

If this is the case then the future for 'simple' two or three valued logics as vehicles for intelligent machines in the future is bleak. As it currently stands, conceptual graphs theory does not offer any possibility of either representing fuzzy concepts or computing with them. What is required is a consistent representation for the alethic modalities and an extension to the rules of inference over these. Peirce addressed these modalities in his system gamma by introducing the broken cut, which means 'possibly false'. Sowa introduced the monadic relations POSS and NECS for possible and necessary but these simply provide *ad hoc* means of expressing the alethic modalities without providing a proper semantics and logic for them.

6.5 FINAL CONCLUSION

In this thesis we have rationalised the notation and semantics of Sowa's conceptual graphs theory and have addressed the logical operations upon these graphs. As a result we have been able to describe an approach to automated computation within the conceptual graphs formalism. We have also gone on to show a realisation of the reconstruction in a prototype system written in Prolog and finally we have discussed areas where the theory of conceptual graphs requires further work and where its power must be increased if it is to compete with other forms of reasoning currently in use.

REFERENCES

[Andrews81] Andrews, P. B., "Theorem Proving *via* General Matings" (*JACM* 28 (2), pp 125 - 163, 1981).

[Aristotle] Aristotle, "The Categories, On Interpretation, Prior Analytics, Posterior Analytics, Topica" (Loed Classical Library, Harvard University Press, Cambridge, M.A.).

[Bakker87] Bakker, R. R., "Knowledge Graphs: Representation and Structuring of Scientific Knowledge" (PhD thesis, University of Twente, Enschede, Holland, 1987).

[Bibel76] Bibel, W., "A Syntactic Connection Between Proof Procedures And Refutation Procedures" (*Second Conference on Automated Deduction*, Oberwolfach, West Germany, 1976).

[Bibel82] Bibel, W., "A Comparative Study of Several Proof Procedures" (*Artificial Intelligence*, Vol 18, pp 269 - 293, 1982).

[Bibel83] Bibel, W., "Matings in Matrices" (*CACM* 26 (11), pp 844 - 852, 1983).

[Bokkers92] Bokkers, E. M., "A Critical Investigation Into The Conceptual Graphs Processor CGP" (Technical Report, Department of Computer Studies, Loughborough University, 1992).

[Butterworth84] Butterworth, J. P. G., "The Modal System T In Peirce's Existential Graphs" (MA thesis, University of Waterloo, Ontario, Canada, 1984).

[Campbell84] Campbell, J. A., "Implementations of Prolog" (Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England).

[Chan88] Chan, M. C., Garner, B. J., Tsui, E., "Recursive Modal Unification for Reasoning with Knowledge Using a Graph Representation" (*Knowledge Based Systems*, Vol 1, No. 2, March 1988).

[Chen76] Chen, P. P.-S., "The Entity-Relationship Model - Towards a Unified View of Data" (*ACM Transactions on Database Systems*, 1:1, pp 9-36).

[Doyle79] Doyle, J., "A Truth Maintenance System" (*Artificial Intelligence*, vol. 12, pp 231 - 272, 1979).

[Duda79] Duda, R., Gaschnig, J., Hart, P., "Model Design In The PROSPECTOR Consultant System For Mineral Exploration" (*Expert Systems In The Micro-Electronic Age*, Edinburgh University Press, 1979, pp 153 - 167).

[Ellis91] Ellis, G., ("Compiled Hierarchical Retrieval" (Proc Sixth Annual Workshop on Conceptual Graphs, 1991).

[Ellis93] Ellis, G., ("*Conceptual Graphs For Knowledge Representation*", Proc. First International Conference on Conceptual Structures, 1993, in "*Lecture Notes In Artificial Intelligence*" vol 699, Springer-Verlag, Berlin Heidelberg, 1993).

[Esch89] Esch, J. W., Nagle, T. E., Yim, M. L., Gerholz, L. L., "Resolving Polymorphism in the Theory of Conceptual Graphs" (Proc Fourth Annual Workshop on Conceptual Graphs, 1989).

[Fargues86] Fargues, J., Landau, M., Dogourd, A., Catach, L., "Conceptual Graphs for Semantics and Knowledge Processing" (*IBM Journal of Research and Development*, Vol 30, No. 1, January 1986).

[Fargues89] Fargues, J., "CG Information Retrieval Using Linear Resolution, Generalisation and Graph Splitting" (Proc Fourth Annual Workshop on Conceptual Graphs, 1989).

[Fetzer90] Fetzer, J. H., "Artificial Intelligence: Its Scope And Limits" (Dordrecht, The Netherlands: Kluwer Academic Publishers, 1990).

[Fitch52] Fitch, F. B., "Symbolic Logic: An Introduction" (New York: Ronald, 1952).

[Foo93] Foo, N., Pagnucco, M., "Inverting Resolution With Conceptual Graphs" ("*Conceptual Graphs For Knowledge Representation*", Proc. First International Conference on Conceptual Structures, 1993, in "*Lecture Notes In Artificial Intelligence*" vol 699, Springer-Verlag, Berlin Heidelberg, 1993).

[Frege79] Frege, G., "Begriffsschrift" (Cambridge, Mass: *From Frege to Godel, 1879-1931*", Harvard University Press, 1967, pp 1 - 82).

[Frost86] Frost, R. A., "Introduction To Knowledge Base Systems" (United Kingdom: William Collins Sons & Co. Ltd., *, Grafton Street, London, W1X 3LA, 1986).

[Gentzen34] Gentzen, G., "Untersuchungen Uber Das Logische Schliesses" (*Mathematische Zeitschrift,* Vol 39, pp 176 - 210 and 405 - 431, 1934).

[Heaton89] Heaton, J. E., "An Investigation into Knowledge Engineering using Conceptual Graphs" (Loughborough University, England: MSc Thesis 1989).

[Heaton91] Heaton, J. E., "Projection and Negation of Conceptual Graphs" (Proc Sixth Annual Workshop on Conceptual Graphs, 1991).

[Heaton92] Heaton, J. E., "CGP User Guide" (Technical Report, Department of Computer Studies, Loughborough University, 1992).

[Hendrix79] Hendrix, G. G., "Encoding Knowledge in Partitioned Networks" ("*Associative Networks: Representation and Use of Knowledge by Computers*", New York: Academic Press, 1979, pp 51 - 92).

[Hill74] Hill, R., "LUSH Resolution and its Completeness" (*DCS Memo No. 78*, School of Artificial Intelligence, Edinburgh University, 1974).

[Ho91] Ho, K. K., "Aspects of Conceptual Graphs Processor Design" (Proc Sixth Annual Workshop on Conceptual Graphs, 1991).

[James92] James, P., "Knowledge Graphs" (LIKE-workshop, Tilburg, 1992).

[Jaskowski34] Jaskowski, S., "On The Rules Of Supposition In Formal Logic" (*Studia Logica* 1).

[Kleene38] Kleene, S. C., "On Notation For Ordinal Numbers" (*Journal of Symbolic Logic*, vol. 2, pp 340 - 353 1938).

[Kleene62] Kleene, S. C., "Introduction to Metamathematics" (D. van Nostrand Company Inc., New York, 1962).

[Kocura90] Kocura, P., "Semantics For Conceptual Graphs" (Technical Report, Department of Computer Studies, Loughborough University, 1990).

[Kocura90b] Kocura, P., "Deep Knowledge Semantics For Conceptual Graphs" (Proc Fifth Annual Workshop on Conceptual Graphs, 1990).

[Kowalski71] Kowalski, R. A., Kuehner, D., "Linear Resolution with Selection Function" (*Artificial Intelligence*, Vol 2, pp 227 - 260).

[Loveland69] Loveland, D., "Theorem Provers Combining Model Elimination and Resolution" (*Machine Intelligence 4*, Meltzer B. & Michie D., New York & Elsevier, North Holland).

[Łukasiewicz20] Łukasiewicz, J., "On Three-Valued Logic" (Ruch Filozoficzny (Lwów), vol. 5, pp 169 - 171, 1920).

[Lukose92] Lukose, D., "Goal Interpretation As A Knowledge Acquisition Mechanism" (Deakin University, Australia: PhD Thesis, 1992).

[Lukose93] Lukose, D., "Critical Evaluation Of The Canonical Graphs Processor (Phases 1 and 2)" (Technical Reports, Department of Computer Studies, Loughborough University, 1993).

[Pfeiffer91] Pfeiffer, H. D., Hartley, R. T., "The Conceptual Programming Environment, CP: Time, Space and Heuristic Constraints" (Proc Sixth Annual Workshop on Conceptual Graphs, 1991).

[Prawitz76] Prawitz, D., "A Proof Procedure With Matrix Reduction" (*Lecture Notes in Mathematics 125*, Berlin and New York, Springer-Verlag, 1976).

[Reiter71] Reiter, R., "Two Results on Ordering for Resolution with Merging and Linear Format" (*JACM* Vol 18, pp 630 - 646, 1971).

[Rescher79] Rescher, N., Brandom, R., "The Logic Of Inconsistency" (Rowman And Littlefield, Totowa, New Jersey, 1979).

[Roberts73] Roberts, D. D., "The Existential Graphs of Charles S. Peirce" (Mouton, The Hague, 1973).

[Shapiro71] Shapiro, S. C., "A Net Structure For Semantic Information Storage, Deduction And Retrieval" (Proc IJCAI 1971, pp 512 - 523).

[Shapiro79] Shapiro, S. C., "The SNePS Semantic Network Processing System" ("*Associative Networks: Representation and Use of Knowledge by Computers*", New York: Academic Press, 1979, pp 179 - 203).

[Smith91] Smith, B., "CAMES - Expert System Administration of Money Market Services" (Proc Sixth Annual Workshop on Conceptual Graphs, 1991).

[Sowa84] Sowa, J. F., "Conceptual Structures: Information Processing in Mind and Machine" (Reading, Mass: Addison-Wesley Publishing Co., 1984).

[Sowa87] Sowa, J. F., "Semantic Networks" ("*Encyclopedia of Artificial Intelligence*", New York: John Wiley & Sons, 1987, pp 1011 - 1024).

[Sowa93] Sowa, J. F., "Relating Diagrams to Logic" ("*Conceptual Graphs For Knowledge Representation*", Proc. First International Conference on Conceptual Structures, 1993, in "*Lecture Notes In Artificial Intelligence*" vol 699, Springer-Verlag, Berlin Heidelberg, 1993).

[Tjan89] Tjan, B. S., Gardiner, D. A., Slagle, J. R., "Direct Inference Rules for Conceptual Graphs with Extended Notation" (Proc Fourth Annual Workshop on Conceptual Graphs, 1989).

[Tsui88] Tsui, E., "Modal Unification Using Conceptual Graphs" (Deakin University, Australia: PhD Theses, 1988).

[Webster88] Webster, D. E., "Mapping the Design Information Representation Terrain" (*Computer*, Dec 1988, pp 8 - 23).

[Wittgenstein21] Wittgenstein, L., "*Tractacus Logico-Philosophicus*" (Routledge & Kegan Paul, London, 1961).

[Wittgenstein53] Wittgenstein, L., "Philosophical Investigations" (Basil Blackwell, Oxford).

[Wuwongse93] Wuwongse, V., Manzano, M., "Fuzzy Conceptual Graphs" ("*Conceptual Graphs For Knowledge Representation*", Proc. First International Conference on Conceptual Structures, 1993, in "*Lecture Notes In Artificial Intelligence*" vol 699, Springer-Verlag, Berlin Heidelberg, 1993).

[Zadeh74] Zadeh, L., "Fuzzy Logic And Its Application To Approximate Reasoning" (*Information Processing 74*, North-Holland, Amsterdam, 1974, pp 591 - 594).

APPENDIX I

THEORY GENERATION

INTRODUCTION

This appendix shows the output produced by CGP when asked to solve a simple 'logic' puzzle from a popular puzzle magazine [*Solve It Logically Special*, issue number 5: Cottage Publishing Ltd., Abbey Lodge, Abbey Gardens, Chertsey, Surrey, KT16 8RQ, UK). It demonstrates the semantics of the generic referent as discussed in Chapter 2. To recapitulate, these semantics state that any graph which contains generic referents and which is enclosed at an even depth is a graph that contains a reference to some individual marker at the same depth (as opposed to the universal quantification of odd depths) but which one is not known. CGP contains a module that can solve some problems where information is of this kind by combining generic graphs with non-generic graphs in different ways to produce theories. The new theory is checked for consistency by the theorem prover since the original puzzle may contain sufficient information to show a new theory to be false. Each consistent theory is then a possible solution to the problem. In the example below there happens (presumably by design) to be only one theory and therefore only one solution but in general there may be more than one. The theory governing the results presented in this appendix is still in its infancy but it is nonetheless instructive to include this section as an indication of the direction of additional further work.

ACCESSIBILITY RELATION

Our semantics of the generic referent tell us that if a graph at depth 0 in some world *W* contains a generic referent then that graph is referring to some individual marker. Which one is not yet known, but it is possibly one of those already known to the system. Without this kind of semantics for the generic referent (and indeed the individual marker) it is impossible for conceptual graphs to solve any puzzle that requires the generation of a theory since Peirce

logic by itself will not generate anything that is not definitely true.

At the simplest level the accessibility relation to theories accessible from *W* is given by the following generate and test algorithm:

Algorithm Ii.1: Accessibility Relation

1) **FOR** each graph *g* in *W* which contains a generic referent **DO;**

2) Find a maximal join of *g* with some other graph such that some of the generic referents in *g* are replaced with individual referents;

3) **ENDFOR(1).**

The output of this algorithm is a new world *W'* which contains possible instantiations of the generic referents in *W*. There is not any guarantee that any of the maximal joins produce graphs which are true. For extra security we can modify the algorithm so that worlds which are palpably false are rejected. There are two ways in which we can do this:

* Test *W'* for selectional constraint violations.

* Test *W'* for logical consistency.

Any world *W'* which does not pass both of these tests is not a theory.

The procedure given here is a gross oversimplification of a complete system. In particular the following additional points must be dealt with in a complete system:

* There is no reason why the joins can only be maximal.

* Evenly enclosed graphs at depths greater than 0 may also be joined to produce new rules.

* Heuristics are required to remove the combinatorial nature of producing each possible theory from even a small set of premises.

Even though the prototype procedure is a simplification and the example problem to be given is a simple puzzle the results suggest that generation of theories (and explanations) by this kind of method is an area where further research would be valuable.

WORKING EXAMPLE

We turn our attention to an example. In this example there are four girls: Cheryl, Maddie, Isobel and Stella. Each of them wants to eat a different type of food, likes a different kind of music and wishes to participate in a different activity. We know the following facts:

* Isobel eats Chinese food.

* One of the girls wants to eat casserole.

* One of the girls both likes rock music and wants to eat burgers. This girl's name is not Stella.

* One of the girls both wishes to do some decorating and to eat vegetarian food.

* .Cheryl likes Country music

* One of the girls likes Heavy Metal music.

* One of the girls both wishes to repair her bicycle and listen to Classical music. This girl's name is not Isobel.

* One girl wishes to read.

* One girl wishes to do some dressmaking. This girl's name is not Maddie.

The nature of the puzzle is that each girl only likes one kind of music, only wishes to participate in one activity and only eats one kind of food. We therefore add this requirement to the list of known facts. The encoding into conceptual graphs may leave something to be desired in terms of the semantics of some of the relations used but the puzzle is not encoded in a way that ensures that the required answer will be generated.

Having encoded the puzzle we ask CGP to find the denotation of the statement 'a girl who likes some kind of music, wishes to participate in some activity and wants to eat some kind of food'. The output from this run does not include the entry of the knowledge into the system but the contents of the knowledge base are first displayed. Following that the query is posed and CGP finds one theory that explains all the facts. This theory produces a denotation consisting of four graphs. A manual solution of the puzzle conforms that they are correct.

The example script starts by loading a file which contains the graphs describing the facts of the puzzle. It is done this way to save space. Nevertheless, this file was created by entering the facts into CGP and allowing it to build the knowledge base itself. In this way it was also able to test the consistency of the puzzle. Following the loading the script prints all graphs and rules in the knowledge base to show that it is exactly what the puzzle states (plus the initial graphs of the lattices). Finally the query is made. The time taken to find all solutions (15 minutes) is more a reflection of the version of Prolog that was used than on the efficiency of the procedure. Nevertheless, even this small puzzle takes a relatively long time.

```
>> %girls.

Reading script girls...

girls:>> <girls2.

Loading girls2...

OK

girls:>> *graphs.

Graphs:

[girl:maddie].

[girl:stella].

[girl:isobel]-
        (eat)->[food_type:chinese].

[girl:*29]-
        (eat)->[food_type:casserole].
```

```
[girl:*33]-
        (like)->[music_type:rock]
        (eat)->[food_type:burgers].

[girl:*34]-
        (wish)->[activity_type:decorating]
        (eat)->[food_type:vegetarian].

[girl:cheryl]-
        (like)->[music_type:country].

[girl:*30]-
        (like)->[music_type:metal].

[girl:*32]-
        (wish)->[activity_type:repair_bicycle]
        (like)->[music_type:classical].

[girl:*31]-
        (wish)->[activity_type:reading].

[girl:*35]-
        (wish)->[activity_type:dressmaking].

(subtype)
        <-[type:food_type]
        <-[type:music_type]
        <-[type:music_type]
        <-[type:activity_type]
        <-[type:girl]
        <-[type:type]
        <-[type:rel]
        ->[type:absurd_type];
(subtype)
        <-[type:universal_type]
        ->[type:food_type]
        ->[type:music_type]
        ->[type:music_type]
        ->[type:activity_type]
        ->[type:girl]
        ->[type:absurd_type]
        ->[type:label]
        ->[type:type]
        ->[type:rel];
(subtype)
        <-[type:label]
        ->[type:absurd_type]
        ->[type:type]
        ->[type:rel].

(subrel)
        <-[rel:eat]
        <-[rel:like]
        <-[rel:wish]
        <-[rel:subtype]
        <-[rel:subrel]
        <-[rel:superlabel]
        <-[rel:supertype]
        <-[rel:superrel]
        ->[rel:absurd_rel];
```

```
(subrel)
        <-[rel:universal_rel]
        ->[rel:eat]
        ->[rel:like]
        ->[rel:wish]
        ->[rel:absurd_rel]
        ->[rel:sublabel]
        ->[rel:subtype]
        ->[rel:subrel]
        ->[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel];
(subrel)
        <-[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel];
(subrel)
        <-[rel:sublabel]
        ->[rel:absurd_rel]
        ->[rel:subtype]
        ->[rel:subrel].

>> *formulae.

Formulae:

(
     [type:*-1]-
             (subtype)->[type:*2];
     [type:*2]-
             (subtype)->[type:*1]
).

(
     [absurd_type:*-1]
).

(
     (absurd_rel)
).

(
     [girl:*26]-
             (*25)->[universal_type:*28]
             (*25)->[universal_type:*27]
).

(
     [girl:stella=*33]
).

(
     [girl:maddie=*35]
).

(
     [girl:isobel=*32]
).

girls:>> +time.
Fri,09 Jul 1993.22:23:07
```

```
girls:>> ?
[girl:*48]-
        (like)->[music_type:*51]
        (eat)->[food_type:*50]
        (wish)->[activity_type:*49].

Checking for selectional constraint violations in:
     [girl:*48]-
             (like)->[music_type:*51]
             (eat)->[food_type:*50]
             (wish)->[activity_type:*49].

Checking denotation of:
     [girl:*48]-
             (like)->[music_type:*51]
             (eat)->[food_type:*50]
             (wish)->[activity_type:*49].

The statement:
     [girl:*48]-
             (like)->[music_type:*51]
             (eat)->[food_type:*50]
             (wish)->[activity_type:*49].

is:

TRUE because:
   1/0.     proof of:
               [girl:*48]-
                       (wish)->[activity_type:*49];
               [girl:*48]-
                       (eat)->[food_type:*50];
               [girl:*48]-
                       (like)->[music_type:*51];
   2/0.     is:
   3/1.          possible theory:
                    [girl:stella]-
                            (eat)->[food_type:casserole]
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
                    [girl:maddie]-
                            (like)->[music_type:rock]
                            (eat)->[food_type:burgers]
                            (wish)->[activity_type:reading];
                    [girl:cheryl]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian]
                            (like)->[music_type:country];
                    [girl:isobel]-
                            (like)->[music_type:metal]
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
(
     [girl:isobel=stella]
);
(
     [girl:isobel=maddie]
);
(
     [girl:maddie=stella]
);
```

```
                    (
                        [girl:*26]-
                                (*25)->[universal_type:*28]
                                (*25)->[universal_type:*27]
                    );
4/1.            generated by:
5/1.            known facts combined:
                    [girl:*35]-
                            (wish)->[activity_type:dressmaking];
                    [girl:isobel]-
                            (eat)->[food_type:chinese];
6/1.            giving:
                    [girl:isobel]-
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
7/1.            known facts combined:
                    [girl:*31]-
                            (wish)->[activity_type:reading];
                    [girl:maddie];
8/1.            giving:
                    [girl:maddie]-
                            (wish)->[activity_type:reading];
9/1.            known facts combined:
                    [girl:*32]-
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
                    [girl:stella];
10/1.           giving:
                    [girl:stella]-
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
11/1.           known facts combined:
                    [girl:*30]-
                            (like)->[music_type:metal];
                    [girl:isobel]-
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
12/1.           giving:
                    [girl:isobel]-
                            (like)->[music_type:metal]
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
13/1.           known facts combined:
                    [girl:*34]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian];
                    [girl:cheryl]-
                            (like)->[music_type:country];
14/1.           giving:
                    [girl:cheryl]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian]
                            (like)->[music_type:country];
15/1.           known facts combined:
                    [girl:*33]-
                            (like)->[music_type:rock]
                            (eat)->[food_type:burgers];
                    [girl:maddie]-
                            (wish)->[activity_type:reading];
16/1.           giving:
```

```
                  [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
17/1.          known facts combined:
                  [girl:*29]-
                        (eat)->[food_type:casserole];
                  [girl:stella]-
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
18/1.          giving:
                  [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
19/1.             result:
                     [girl:isobel]-
                           (wish)->[activity_type:dressmaking];
20/1.             result:
                     [girl:isobel]-
                           (eat)->[food_type:chinese];
21/1.             result:
                     [girl:isobel]-
                           (like)->[music_type:metal];
22/0.       to give:
               {}.

TRUE because:
   1/0.     proof of:
               [girl:*48]-
                     (wish)->[activity_type:*49];
               [girl:*48]-
                     (eat)->[food_type:*50];
            [girl:*48]-
                     (like)->[music_type:*51];
2/0.        is:
3/1.           possible theory:
                  [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
                  [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
                  [girl:cheryl]-
                        (wish)->[activity_type:decorating]
                        (eat)->[food_type:vegetarian]
                        (like)->[music_type:country];
[girl:isobel]-
        (like)->[music_type:metal]
        (wish)->[activity_type:dressmaking]
        (eat)->[food_type:chinese];
(
     [girl:isobel=stella]
);
(
     [girl:isobel=maddie]
);
```

```
                    (
                         [girl:maddie=stella]
                    );
                    (
                         [girl:*26]-
                                 (*25)->[universal_type:*28]
                                 (*25)->[universal_type:*27]
                    );
4/1.           generated by:
5/1.           known facts combined:
                    [girl:*35]-
                            (wish)->[activity_type:dressmaking];
                    [girl:isobel]-
                            (eat)->[food_type:chinese];
6/1.           giving:
                    [girl:isobel]-
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
7/1.           known facts combined:
                    [girl:*31]-
                            (wish)->[activity_type:reading];
                    [girl:maddie];
8/1.           giving:
                    [girl:maddie]-
                            (wish)->[activity_type:reading];
9/1.           known facts combined:
                    [girl:*32]-
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
                    [girl:stella];
10/1.          giving:
                    [girl:stella]-
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
11/1.          known facts combined:
                    [girl:*30]-
                            (like)->[music_type:metal];
                    [girl:isobel]-
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
12/1.          giving:
                    [girl:isobel]-
                            (like)->[music_type:metal]
                            (wish)->[activity_type:dressmaking]
                            (eat)->[food_type:chinese];
13/1.          known facts combined:
                    [girl:*34]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian];
                    [girl:cheryl]-
                            (like)->[music_type:country];
14/1.          giving:
                    [girl:cheryl]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian]
                            (like)->[music_type:country];
15/1.          known facts combined:
                    [girl:*33]-
                            (like)->[music_type:rock]
                            (eat)->[food_type:burgers];
```

```
                  [girl:maddie]-
                        (wish)->[activity_type:reading];
16/1.          giving:
                  [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
17/1.          known facts combined:
                  [girl:*29]-
                        (eat)->[food_type:casserole];
                  [girl:stella]-
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
18/1.          giving:
                  [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
19/1.          result:
                  [girl:cheryl]-
                        (wish)->[activity_type:decorating];
20/1.          result:
                  [girl:cheryl]-
                        (eat)->[food_type:vegetarian];
21/1.          result:
                  [girl:cheryl]-
                        (like)->[music_type:country];
22/0.     to give:
             {}.

TRUE because:
  1/0.     proof of:
             [girl:*48]-
                   (wish)->[activity_type:*49];
             [girl:*48]-
                   (eat)->[food_type:*50];
             [girl:*48]-
                   (like)->[music_type:*51];
2/0.     is:
3/1.          possible theory:
                 [girl:stella]-
                       (eat)->[food_type:casserole]
                       (wish)->[activity_type:repair_bicycle]
                       (like)->[music_type:classical];
                 [girl:maddie]-
                       (like)->[music_type:rock]
                       (eat)->[food_type:burgers]
                       (wish)->[activity_type:reading];
                 [girl:cheryl]-
                       (wish)->[activity_type:decorating]
                       (eat)->[food_type:vegetarian]
                       (like)->[music_type:country];
[girl:isobel]-
       (like)->[music_type:metal]
       (wish)->[activity_type:dressmaking]
       (eat)->[food_type:chinese];
(
     [girl:isobel=stella]
);
```

```
                    (
                         [girl:isobel=maddie]
                    );
                    (
                         [girl:maddie=stella]
                    );
                    (
                         [girl:*26]-
                                 (*25)->[universal_type:*28]
                                 (*25)->[universal_type:*27]
                    );
4/1.           generated by:
5/1.           known facts combined:
                    [girl:*35]-
                                 (wish)->[activity_type:dressmaking];
                    [girl:isobel]-
                                 (eat)->[food_type:chinese];
6/1.           giving:
                    [girl:isobel]-
                                 (wish)->[activity_type:dressmaking]
                                 (eat)->[food_type:chinese];
7/1.           known facts combined:
                    [girl:*31]-
                                 (wish)->[activity_type:reading];
                    [girl:maddie];
8/1.           giving:
                    [girl:maddie]-
                                 (wish)->[activity_type:reading];
9/1.           known facts combined:
                    [girl:*32]-
                                 (wish)->[activity_type:repair_bicycle]
                                 (like)->[music_type:classical];
                    [girl:stella];
10/1.          giving:
                    [girl:stella]-
                                 (wish)->[activity_type:repair_bicycle]
                                 (like)->[music_type:classical];
11/1.          known facts combined:
                    [girl:*30]-
                                 (like)->[music_type:metal];
                    [girl:isobel]-
                                 (wish)->[activity_type:dressmaking]
                                 (eat)->[food_type:chinese];
12/1.          giving:
                    [girl:isobel]-
                                 (like)->[music_type:metal]
                                 (wish)->[activity_type:dressmaking]
                                 (eat)->[food_type:chinese];
13/1.          known facts combined:
                    [girl:*34]-
                                 (wish)->[activity_type:decorating]
                                 (eat)->[food_type:vegetarian];
                    [girl:cheryl]-
                                 (like)->[music_type:country];
14/1.          giving:
                    [girl:cheryl]-
                                 (wish)->[activity_type:decorating]
                                 (eat)->[food_type:vegetarian]
                                 (like)->[music_type:country];
15/1.          known facts combined:
```

```
                [girl:*33]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers];
                [girl:maddie]-
                        (wish)->[activity_type:reading];
16/1.       giving:
                [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
17/1.       known facts combined:
                [girl:*29]-
                        (eat)->[food_type:casserole];
                [girl:stella]-
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
18/1.       giving:
                [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
19/1.       result:
                [girl:maddie]-
                        (wish)->[activity_type:reading];
20/1.       result:
                [girl:maddie]-
                        (eat)->[food_type:burgers];
21/1.       result:
                [girl:maddie]-
                        (like)->[music_type:rock];
22/0.   to give:
            {}.

TRUE because:
   1/0.     proof of:
                [girl:*48]-
                        (wish)->[activity_type:*49];
                [girl:*48]-
                        (eat)->[food_type:*50];
                [girl:*48]-
                        (like)->[music_type:*51];
   2/0.     is:
   3/1.       possible theory:
                [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
                [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
[girl:cheryl]-
                        (wish)->[activity_type:decorating]
                        (eat)->[food_type:vegetarian]
                        (like)->[music_type:country];
[girl:isobel]-
                        (like)->[music_type:metal]
                        (wish)->[activity_type:dressmaking]
                        (eat)->[food_type:chinese];
```

```
(
     [girl:isobel=stella]
);
(
     [girl:isobel=maddie]
);
(
     [girl:maddie=stella]
);
(
     [girl:*26]-
               (*25)->[universal_type:*28]
               (*25)->[universal_type:*27]
);
4/1.       generated by:
5/1.       known facts combined:
              [girl:*35]-
                       (wish)->[activity_type:dressmaking];
              [girl:isobel]-
                       (eat)->[food_type:chinese];
6/1.       giving:
              [girl:isobel]-
                       (wish)->[activity_type:dressmaking]
                       (eat)->[food_type:chinese];
7/1.       known facts combined:
              [girl:*31]-
                       (wish)->[activity_type:reading];
              [girl:maddie];
8/1.       giving:
              [girl:maddie]-
                       (wish)->[activity_type:reading];
9/1.       known facts combined:
              [girl:*32]-
                       (wish)->[activity_type:repair_bicycle]
                       (like)->[music_type:classical];
              [girl:stella];
10/1.      giving:
              [girl:stella]-
                       (wish)->[activity_type:repair_bicycle]
                       (like)->[music_type:classical];
11/1.      known facts combined:
              [girl:*30]-
                       (like)->[music_type:metal];
              [girl:isobel]-
                       (wish)->[activity_type:dressmaking]
                       (eat)->[food_type:chinese];
12/1.      giving:
              [girl:isobel]-
                       (like)->[music_type:metal]
                       (wish)->[activity_type:dressmaking]
                       (eat)->[food_type:chinese];
13/1.      known facts combined:
              [girl:*34]-
                       (wish)->[activity_type:decorating]
                       (eat)->[food_type:vegetarian];
              [girl:cheryl]-
                       (like)->[music_type:country];
14/1.      giving:
```

```
                    [girl:cheryl]-
                            (wish)->[activity_type:decorating]
                            (eat)->[food_type:vegetarian]
                            (like)->[music_type:country];
15/1.           known facts combined:
                    [girl:*33]-
                            (like)->[music_type:rock]
                            (eat)->[food_type:burgers];
                    [girl:maddie]-
                            (wish)->[activity_type:reading];
16/1.           giving:
                    [girl:maddie]-
                            (like)->[music_type:rock]
                            (eat)->[food_type:burgers]
                            (wish)->[activity_type:reading];
17/1.           known facts combined:
                    [girl:*29]-
                            (eat)->[food_type:casserole];
                    [girl:stella]-
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
18/1.           giving:
                    [girl:stella]-
                            (eat)->[food_type:casserole]
                            (wish)->[activity_type:repair_bicycle]
                            (like)->[music_type:classical];
19/1.           result:
                    [girl:stella]-
                            (wish)->[activity_type:repair_bicycle];
20/1.           result:
                    [girl:stella]-
                            (eat)->[food_type:casserole];
21/1.           result:
                    [girl:stella]-
                            (like)->[music_type:classical];
22/0.       to give:
                    {}.

girls:>> +time.
Fri,09 Jul 1993.22:38:42

girls:>> end_of_file.
OK

>>
```

APPENDIX II

CGP USER GUIDE

INTRODUCTION

This appendix is made up of the complete User Guide for CGP. As such it documents features of the program that are not reported in the body of the thesis but which are nevertheless extant for experimental reasons.

In addition there are many example proofs within the User Guide and these can be used to augment those given in the main text.

CGP

USER GUIDE

John E. Heaton
Department Of Computer Studies
Loughborough University Of Technology
England

August 1993

CGP

USER GUIDE

CONTENTS

General Information 1
- Introduction 1.1
- Running The Program 1.2
- CGP'S World Model 1.3
  - Truth Values 1.3.1
  - Denotation 1.3.2
  - Semantic Net 1.3.3
  - Nested Models 1.3.4
- Metaknowledge 1.4
- Supplementary Information 1.5

Conceptual Graph Syntax And Semantics 2
- Concepts 2.1
- Relations 2.2
- CGP Objects 2.3
  - Generic - * 2.3.1
  - Coreferent Generic - **var* 2.3.2
  - Individual - *#marker* 2.3.3
  - Name - *string* 2.3.4
  - Prototype - *&* 2.3.5
  - Measure - *@* 2.3.6
  - Context - *graphs* 2.3.7
  - Generic Context - *{ * }* 2.3.8
  - Equality Symbol - = 2.3.9
- Simple Graphs 2.4
- Contexts 2.5
  - Positive Contexts 2.5.1
  - Negated Contexts 2.5.2
  - Separating Graphs Within Contexts 2.5.3
  - Lambda Expressions 2.5.4
- Other Data 2.6
  - The Type And Relational Lattices 2.6.1
  - Conformity Relations 2.6.2
  - Definitions 2.6.3
    - Type Definitions 2.6.3.1
    - Relational Definitions 2.6.3.2
  - The Canon 2.6.4
  - The Lexicon 2.6.5
- Formatting Graphs 2.7
- Text Files And Scripts 2.8
  - Simple Text Files 2.8.1
  - Scripts 2.8.2
- Metasyntax 2.9
- CGP's Built-In Canonical Basis 2.10
  - Built-In Type Lattice 2.10.1
  - Built-In Relational Lattice 2.10.2
  - Built-In Canonical Graphs 2.10.3

CGP Commands 3
The < Command 3.1
The > Command 3.2
The . Command 3.3
The \ Command 3.4
The % Command 3.5
The * Command 3.6
The ~ Command 3.7
The + Command 3.8
The ^ Command 3.9
The $ Command 3.10
The ! Command 3.11
The ? Command 3.12
The & Command 3.13
The s Command 3.14
The v Command 3.15
The p Command 3.16
The i Command 3.17
The c Command 3.18
The o Command 3.19
The z Command 3.20
The f Command 3.21
The g Command 3.22
The n Command 3.23
The l Command 3.24
The d Command 3.25
The t Command 3.26
The u Command 3.27
The b Command 3.28
The m Command 3.29
The @ Command 3.30

Using The Program 4
Adding Simple Knowledge 4.1
Simple Queries 4.2
Canonicity Checking 4.3
Placing Of Newly Encountered Types Within The Lattices 4.3.1
Adding Subtypes Of Known Types 4.3.2
Adding New Relations 4.3.3
Selectional Constraint Violations 4.3.4
Definition Inconsistencies 4.3.5
Some Simple False Facts 4.4
Simple Deduction 4.5
Simple Rules 4.5.1
A More Subtle Example 4.5.2
More Complex Deduction 4.6
Deduction Requiring Several Facts 4.6.1
Deduction Requiring A Chain Of Rules 4.6.2
Deduction With Recursive Use Of A Rule 4.6.3
Modus Tollens 4.6.4
Deductions Requiring The Use Of Conditional Rules 4.6.5
Deductions Requiring Combinations Of Rules 4.6.6
Higher Order Reasoning 4.6.7
Theorem Proving 4.7
Removal Of Double Negations 4.7.1
Delteration 4.7.2
Elimination Of Disjunctions 4.7.3
Iteration 4.7.4
Restriction 4.7.5
Some Examples 4.7.6
Resolution 4.7.7
The Role Of The Theorem Prover In Deduction 4.8

Other Forms Of Reasoning 4.9
Possible Worlds 4.9.1
More About The CGP Interface 4.10
Evaluation Of New Graphs By The Interface 4.10.1
Inferences Made By The Interface 4.10.2
CGP'S Operational Modes 4.11
Speed 4.11.1
Reports 4.11.2
Interaction 4.11.3
Evaluation 4.11.4
Check With User 4.11.5
Iteration 4.11.6

Appendix I - CGP Messages

Appendix II - Peirce Logic

Appendix III - Internal representation

Appendix IV - Known problems

Index

SECTION 1 - GENERAL INFORMATION

1.1 Introduction

CGP is designed to be a fully automated, interactive, goal-driven reasoning program based on conceptual graphs representation and operations. It exists in order to demonstrate that automated reasoning using conceptual graphs is possible, and as such, CGP is very much an experimental system. Version 4 is an implementation of an enhanced Prolog-like deduction mechanism augmented with a general theorem prover; it can accept arbitrary facts and logical constructs (all refered to by the term *formulae* or *graphs*) and answer queries by using combinations of the facts, and certain fundamental logical truths. An explanation facility provides a description of the reasoning behind CGP's conclusions.

In addition to performing proofs of queries, all input into the system is checked for selectional constraint violations. In order for this to take place to the maximum effect, an interactive mechanism for placing newly encountered types and relational labels into the correct place in the appropriate lattice is provided. Included within this mechanism is a means of entering definitions for new labels. Also, for newly encountered relational labels, an opportunity to provide the most general unambiguous use for the label is given. This usage is then used to check whether the actual usage is correct.

The basic model that is supported by CGP is that of the *Open World* and the evaluation of simple queries is carried out by the *Open World Evaluation Game*. On its own, the Open World Evaluation Game is not strong enough to perform all of logic. It is for this reason that CGP is augmented with a Prolog-like deduction mechanism and a theorem prover.

As the program is of an experimental nature, it is possible that there are situations in which it will fail to find answers to queries. There are three reasons for this. The first is that certain features of conceptual graph theory are not yet implemented. The second is that the novel nature of the program means that new theory is included and some of this theory is not yet complete. The third reason is that there may be 'bugs' present in the code or its specification that prevent proofs from proceeding under certain conditions. Whilst every attempt has been made to avoid simple coding errors these do come to light occasionally. However, the program will either correctly answer a query or it will respond with 'do not know' and can therefore be classified a *safe*. The author would appreciate any such failures being notified to him.

Throughout this guide, interactions with the computer are shown as:

```
terminal output
```

CGP is written in Prolog, compiled using the SICStus compiler and is available on the Suna.

1.2 Running The Program

The program is activated by entering the command:

```
cgp
```

after which there is a short delay whilst it is loaded. Once loaded, a message similar to the following is displayed:

```
Welcome to CGP version 4.0.
For help or latest information, enter ?? followed by a full stop
at the prompt (>>).

>>
```

An online help facility is available and is accessed by use of the ??. command as suggested above. This is self-explanatory, with screen menus, and will not be described further.

The CGP command set is very limited as far as evaluation of queries is concerned. This is intentional as one important aspect of the operation of CGP is that it is automated as much as possible. The majority of the commands are used for housekeeping tasks such as loading or saving files and for configuring the system. There are only three commands that actually cause CGP to perform any real work. Two of these

commands are commands to inform CGP of some knowledge and the third command is a command to query CGP about its knowledge. Each command will be described in detail later.

The following sections describe the syntax and basic purpose of each command and a later section goes into more detail about the detailed uses, and in doing so, provides something of a simple tutorial. Also, a summary of the linear form syntax for conceptual graphs that CGP accepts is given in the next section.

1.3 CGP'S World Model

1.3.1 Truth Values

CGP uses the *open world* model. In an open world model there are three logic values. These are:

TRUE	a fact is known to the system to be true,
FALSE	a fact is known to the system to be false,
UNKNOWN	the system does not know whether a fact is TRUE or FALSE.

The last truth value, UNKNOWN, exists in two forms, so in reality, an open world model can distinguish four different truth values. In one case a formula may be UNKNOWN if there is simply no statement in the knowledge base stating its truth. In the other case, a formula may be provable as both TRUE and FALSE, in which case the knowledge base is inconsistent. CGP recognises these two different forms of the UNKNOWN truth value, therefore, whenever UNKNOWN arises during a proof then some fact is truly unknown, but if the knowledge base is inconsistent then the value INCONSISTENT is generated. Under some circumstances a formula which is INCONSISTENT may be reported as TRUE and then FALSE. This depends on how the system is configured.

The truth value FALSE is also used when any formula fails the selectional constraint violation tests. In this situation, FALSE actually means IMPLAUSIBLE or MEANINGLESS.

1.3.2 Denotation

The *denotation* of a simple formula consists of two parts: the truth value and the set of all known cases for which the formula has a model in the knowledge base (the formula's *extension*). A compound formula has only a truth value.

1.3.3 Semantic Net

All new simple graphs that enter the system are joined on identical concepts into as few large graphs as possible, forming a kind of *semantic net*, which is known as the *true set*. This is in accordance with the requirement of an open world model that the set of true graphs should contain each individual marker only once. The joining of graphs also has an effect upon whether a query can be proved or not.

An open world has a second set of graphs; the *false set*. CGP records these as being true statements about the falsity of a proposition. Therefore, if *p* is FALSE then it is TRUE that *p* is FALSE. False propositions cannot be joined to the semantic net because graphs cannot be joined across a context boundary, and so CGP keeps these separate. Once again this is in accordance with the open world model in that individual markers may occur many times within the set of false graphs. These false graphs include the laws of the domain.

Defaults, schemata and prototypes are the final components of an open world model. These are not handled in any way by the current version of CGP.

1.3.4 Nested Models

It is theoretically possible to nest models within other models. In this situation one model would be the 'global model', which would be considered as the model of the 'real world' by CGP and other models would be possible worlds. Such possible worlds would contain their own canons and laws, but would also inherit from their parent world in cases where there was no canon or law local to the nested model. This situation is exactly what occurs when people read or create stories. All stories describe possible worlds and

where the story proposes laws that either conflict with the real world are which are completely conjectural (as in Science Fiction) then the nested model that represents the story is augmented with the extra information.

CGP does not yet fully support nested models, and so each knowledge base must be self contained. In the future it is envisaged that CGP will be able to support nested models and will have facilities to measure their plausibility and similarity to the global model.

1.4 Metaknowledge

Metaknowledge is knowledge about knowledge. As an example, if it is known that Clyde is an elephant then that fact is knowledge. Also, if it is known that an elephant is a mammal then that also is knowledge. If, however it is known that elephant is a type, mammal is a type and that elephant is a subtype of mammal then these facts are often considered to be metaknowledge.

Any system must recognise metaknowledge as special cases of knowledge. CGP incorporates the basis of a metaknowledge processing system but this system is as yet incomplete. Section 2.9 describes all the metaknowledge of which CGP is currently aware.

1.5 Supplementary Information

As CGP is a development prototype, its specification and functionality will change from time to time. In order to keep the documentation up to date, a supplement to this guide will be released as necessary. Information about this will be provided in the banner message that is issued when CGP is first run.

SECTION 2 - CONCEPTUAL GRAPH SYNTAX AND SEMANTICS

On the whole, the syntax that CGP understands is similar to the linear form of Sowa with certain omissions. There is, however, a small number or minor changes that allow the parser to be simplified slightly and keep Prolog happy. In addition, certain minor changes have been made for theoretical reasons.

The following sections describe the syntax and semantics of each object, and examples of their use are given later. Certain important theoretical points will be made in what follows, but this section is not intended to act as a conceptual graphs tutorial.

2.1 Concepts

Concept nodes are shown as being enclosed within the usual square brackets ([ ]) and contain a type field and a referent field separated by a colon (:). The general form of a concept is therefore:

[*type* : *referent*]

Each of these fields can contain any one of a set of *objects*, so that *type* is an object and *referent* is an object. These objects correspond to type labels and referents. Each object will be described later.

As an option, the referent and the colon can be omitted if the referent is the generic object. In this case, CGP will allocate a system-generated referent. The type field can never be omitted.

All generic concepts are treated by CGP as being *existentially* quantified. This interpretation of concepts is vital to the reading of any graph that CGP produces and CGP will interpret all graphs in this way. Existential quantification is one of the four logical primitives of conceptual graphs.

2.2 Relations

A CGP relational node is shown as in the standard linear form; enclosed in parentheses (( )). Once again, the relation can contain any of the CGP objects to be described later. Version 4 of CGP supports relations of any *arity* (number of attached arrows), and these are directed relations with zero or more arrows pointing towards the relational node and zero or more pointing away. In general, any number of concepts can be joined to each relation, with syntax of the general form:

(*relation*)-
->[*concept2*]
<-[*concept1*]

The arrows that join concepts and relations are made up of a single hyphen and either < or > as appropriate and there is a single hyphen following the relational node. The reading of all relations of this form is '*concept1* has a *relation* which is *concept2*.

The above relation could also be typed in reverse to give:

(*relation*)-
<-[*concept1*]
->[*concept2*]

in which case it is read as '*concept2* is a *relation* of *concept1*'. This form, with the relation node and hyphen placed first followed by a list of attached concepts, is that which must be used for *n*-adic relations in general. Each such relation within any one graph must be entered as a separate graph with suitable coreference links to the remaining parts. This is a restriction imposed by the current version of the parser. Another example might be:

(*between*)-
<-[*brick* : *1*]
<-[*brick* : *2*]
->[*brick* : *3*]
->[*brick* : *4*]

in which 'brick 1 AND brick 2 have between them brick 3 AND brick 4'. The concepts attached to the relation can be listed in any order but each must be preceded by an arrow pointing in the correct direction otherwise the parser is unable to determine the correct structure.

As a special case, the use of diadic relations does not suffer from any restrictions and the above diadic relations could be entered as:

[*concept1*]->(*relation*)->[*concept2*]

and any graph, or section of a graph, that contains diadic relations only can be entered as one complete graph in the normal way.

The directionality of all relations must not be neglected. By always reading relations in this standard way, the problem of remembering which way a relation should point is avoided, and anyone who tries to read a graph will know what it says. Those who are familiar with *entity-relation* theory are particularly susceptible to making errors in the direction of relations and extra care is required. Some relations are bidirectional (symmetrical). To enter a bidirectional relation, a graph must contain the relation twice, once in each direction. For example, the *sibling* relation is bidirectional because if *x* is a sibling of *y* then *y* is a sibling of *x*:

[*person* :**x*]-
(*sibling*)->[*person* : **y*]
(*sibling*)<-[*person* : **y*].

On the other hand, the *brother* relation is not bidirectional:

[*girl : Jane*]->(*brother*)->[*boy : John*].

Note that the entity to which the relation points is the one that is playing the *role* described by the relation.

2.3 CGP Objects

The following is a list of the objects that can be placed in any referent, type or relational field. Objects such as control marks for actors are not available (as not indeed are actors).

2.3.1 Generic - *.

The generic object is the conceptual graphs variable. It can stand for any other object (except contexts - see 2.3.7). The CGP parser will allocate a unique coreference marker to each generic object (this modification of the object does not alter its semantics). In the case where the * object is optionally omitted from the referent field CGP will again allocate a system-generated generic object with a unique marker.

It is important to note that the two concepts:

[*type*]

[*type*]

when they occur as separate concepts within one graph, are not regarded by CGP as necessarily being the same concept. This is so because of the absence of a line of identity between them.

2.3.2 Coreferent Generic - **var*

The coreferent generic object is similar to the generic object except that it has the additional property of linking different concepts or relations together. It provides the means of representing graphs that cannot be drawn in linear form without typing some concepts more than once, linking different fields together or linking graphs across context boundaries. In order to ensure the uniqueness of the coreference marker in each graph within the knowledge base, CGP replaces each marker in each new graph with a unique, system-generated one in a way that does not alter the semantics of the graph.

2.3.3 Individual - *#marker.*

Individual objects are represented in the standard way, as a # symbol followed by a unique character string, which is normally a number. Thus, the marker *#53246* is a reference to the individual whose reference number is 53246.

Since the system allocates a # marker to all name referents and other labels it is not safe to use them without first being sure that they have already been allocated by CGP.

2.3.4 Name - *String.*

Name objects are arbitrary character strings that must be unique to the entity to which they are assigned. They can contain arbitrary alpha-numeric characters and, in addition if they are surrounded by single quotes, can contain spaces and most of the characters which form part of conceptual graph syntax except full stops. Type labels and relation labels are normally objects of this kind.

CGP converts all string objects to an equivalent system generated individual object. This allows the unification algorithm to be simplified and thus the system runs more quickly. It also means that string objects must be unique to the individual to whom they are allocated. Thus, when referring to two different people who happen to be called John then it is the user's responsibility to ensure that the string objects for each are different, *e.g.* John_1 and John_2.

Sometimes it is desirable to refer to some entity within the universe of discourse even though it does not have a known name. Thus when one says that 'there is someone at the door' the someone is a definite person but it is not known who. In this case it is not possible to refer to the person by name or individual marker and it would be unsafe to allocate one's own new individual marker in case the system had already used it. To alleviate this problem, the special string objects *some* and *some_...* are available. Within any one input context the label *some* refers to the same individual wherever it appears. When this label is parsed the system allocates a new individual marker to it and informs the user so that this can be safely used at a later time. The *some_...* string can be followed any other string to allow a context with more than one such unknown individual to be represented. Thus the labels *some_one*, *some_body*, *some_body_else*, *etc.* can be used. The use of the _ character allows the use of words that happen to start with *some* to be used as expected. Thus the word *somersault* would be used in the same way as the word John.

2.3.5 Prototype - *&.*

A prototype object is similar to an individual object in that it represents some individual of some type. It is used to refer to a typical individual of that type and, as such, need not have any real existence. It simply stands for some imagined entity that can be talked about as if it existed.

2.3.6 Measure - *@.*

A measure object is an object that represents any form of generalised quantifier or enumeration. There are two basic forms. The first is used to specify non-dimensional values such as degree (much, not very *etc*) or colour or sound pitch. In this role, the measure referent takes the form of the @ symbol followed by the dimensionless value. An example of this first form is *@red*, which represents the colour red.

The second form is more complex and is used to represent any quantity and its associated units, which can be as complex as necessary. The value, which will generally be a number but need not be, is followed by the unit which is represented as an arithmetical formula. The grammar is:

S :- @ term

term :- (term)

term :- term * term

term :- term / term

term :- value [[*] term]

term :- unit

The tokens of type 'value' and 'unit' are the terminal symbols of the grammar. Values are any generalised quantifier such as numbers and units are any one dimensional unit such as foot, second or hectare. Examples of valid measure referents are:

@ 2 ft	2 feet	distance
@ 2 ft * ft	2 square feet	area
@ 70 mile / hour	70 miles per hour	speed
@ 10 m / (s * s)	10 meters per second squared	acceleration
@ (2 ft) * (2 ft)	2 feet squared	area

Note the use of parentheses in the final two examples to show the order of precedence. CGP adopts the conventional left associativity where parentheses are omitted, but they should be included where there may be ambiguity.

A special form of the @ object is available to indicate *many*. Thus the concept:

[*person* : *@many*]

refers to many people.

The basic @ notation can be augmented by adding a comparison operator. These are: *lt*, *le*, *ne*, *ge*, *gt*. The *eq* operator is the default assumed by the basic @ notation and is not available separately. Thus it is possible to make a query of the system about people whose age is between 21 and 30 inclusive by including the concept:

[*age* : *@ge 21 = @ le 30*]

The use of the = sign before the second *2* is simply as a separator (see section 2.3.9).

2.3.7 Context

Any set of graphs can be used as an object. Such a set of graphs, called a *context*, will contain further objects which can also be graphs. Graphs can therefore be nested to any depth within any field. Included within this type of object is the *lambda abstraction*. The syntax for a set of graphs is given in the section on contexts (section 2.5).

2.3.8 Generic Context

A generic context is any context that contains an unspecified set of graphs. It is shown as a * and optional coreference marker surrounded by braces ({ }). By using this device it is possible to refer to whole contexts in a general way. Thus we might state that:

((*necs*)->[*proposition* : { **x* }] ((*necs*)->[*proposition* : { (*necs*)->[*proposition* : { **x* }] }]))

which is one way to state that if the proposition *x is necessarily true then the proposition *x is necessarily necessarily true. The use of { **x* } instead of simply **x* reflects the subtle difference between a marker which is a surrogate for an entity in the real world and a context which is a description of an entity in the real world.

In addition to their use within the referent field (or any other field), generic contexts can also be used on their own. Thus we may state the following:

({ **x* } ([*proposition* : { **x* }]))

which states that any set of graphs is a proposition. The use of the parentheses is explained in section 2.5.

2.3.9 Equality Symbol - =

In order to allow multiple labels within any field (*i.e.* to express the fact that the two labels represent the same individual within the model) the labels are placed within the same field and separated by the equals sign (=). Any field can contain as many objects as required, joined in this way. The provision of this facility allows equality to be represented and computed. Equality is the second logical primitive of conceptual graphs.

2.4 Simple Graphs

A *simple* conceptual graph is made up of either a single concept, a single relation or a connected set of relations. Any graph that consists of a simple chain of concepts and relations can be typed on one line, but most real graphs have branches and cycles. CGP adopts the standard linear form method of showing coreference links between coreferent generic concepts, by the use of coreferent generic objects. Branches are terminated by the usual commas and complete graphs terminated by full stops.

The following are all valid simple graphs:

* [*type*].

* [*type1*]->(*rel*)->[*type2*].

* [*type1*]->(*rel*)->[*type2*]->(*rel*)->[*type3*].

* [*type2*]-
  (*rel*)->[*type3*]
  (*rel*)<-[*type1*].

* [*type1*]-
  (*rel*)->[*type2*]-
  (*rel*)->[*type3*],.

With reference to *n*-adic relations and the restrictions placed upon their representation in CGP the following is not allowed:

(*between*)-
<-[*brick* : *1*]->(*attr*)->[*colour* : *green*]
<-[*brick* : *2*]
->[*brick* : *3*]->(*attr*)->[*colour* : *red*]
->[*brick* : *4*].

whereas its equivalent form:

(*between*)-
<-[*brick* : *1*]
<-[*brick* : *2*]
->[*brick* : *3*]
->[*brick* : *4*]
[*brick* : *1*]->(*attr*)->[*colour* : *green*]
[*brick* : *3*]->(*attr*)->[*colour* : *red*].

is. Note that only the final graph is terminated with a full stop.

2.5 Contexts

Any context falls under the category of *compound graph* or *complex graph*. There are two kinds.

2.5.1 Positive Contexts

A *context* is any set of graphs. The logical connective between each graph in the set is AND. Contexts come in two forms: *positive* and *negated*. A positive context can be represented in either of two ways; the graphs can be enclosed within a set of braces ({ }) or the braces may be omitted. If *p*, *q* and *r*

are arbitrary graphs then the following are equivalent positive contexts:

{ *p q r* }

p q r

p { *q r* }

CGP ignores the braces. The logical connective between all three graphs is AND. AND is the third logical primitive of conceptual graphs.

Any positive context may contain any number of simple graphs and also any number of negated contexts in any combination.

2.5.2 Negated Contexts

A negated context is one in which the graphs within the context are of the opposite truth value to those outside it and is represented in CGP by enclosing the graphs within parentheses, which are compulsory. Negated contexts can be nested to produce complex logical formulae. Once again, the logical connective between the enclosed graphs is AND. Negation is the fourth logical primitive of conceptual graphs.

The following examples illustrate negated contexts:

(*p*)	the proposition made by graph *p* is FALSE,
(*p q*)	it is FALSE that both propositions *p* AND *q* are TRUE,
((*p*) (*q*))	it is FALSE that proposition *p* is FALSE AND proposition *q* is FALSE,
(*p* (*q*))	it is FALSE that proposition *p* is TRUE AND proposition *q* is FALSE.

These example illustrate the versatility of negated contexts in representing all of propositional logic. The third example is a common case, *disjunction*, which states in normal language that either *p* OR *q* is TRUE (since they are not both false). They may both be true, unless as in these examples there is also a statement that they are not both true. The fourth example is also a common one, that of *implication*. The importance of implication to CGP will become apparent later. Section 2.6 uses it to replace the traditional means of expressing definitions. Appendix II gives a list of some common logical formulae expressed in this form.

A negated context may contain any number of nested simple graphs or negated contexts.

2.5.3 Separating Graphs Within Contexts

The graphs within a context are not individually terminated with a full stop, because this would confuse Prolog. Instead, the whole context is terminated with a full stop and the nested graphs are terminated with a semicolon (;). In many cases, the semicolon is optional, but where a concept, or a branch of a graph terminated with commas, is followed by the opening parenthesis of a negated context, the semicolon is compulsory to prevent CGP from thinking that the opening parenthesis is a relation which is part of the previous graph.

Contexts may be placed within type, relation or referent fields. In this case, the context is not terminated by a full stop, because it is only a part of the greater context which contains the graph in which it is nested.

2.5.4 Lambda Expressions

Where a lambda expression is required it has the following syntax:

[*lambda(x) <context> : referent*]

where the parameter of the lambda expression must be a generic object marker that also occurs within the

context (lambda expressions within relational nodes must have at least two parameters, the first ones being the obsject that occur within the concepts which point towards the relational node and the final ones being the other ones). Note that, as stated in section 2.5.3, the graphs within the lambda expression are not terminated by a full stop.

2.6 Other Data

There are very few other forms of data that CGP accepts. Definitions are entered in a form that is logically equivalent to Sowa's definitions, and these are described in section 2.6.3. The remaining data structures that CGP can accept directly are *type lattice* relations and *relational lattice* relations. Additionally, during the use of the system, a set of canonical graphs that give the most general unambiguous usage of each relation will be built up. These graphs are requested by the system as required and cannot be entered directly at the prompt. Also, a lexicon which associates words with internal markers will be created. Again, these are generated by the system during run-time. Both the canon and the lexicon can be perused. Further data include some items recorded for efficiency reasons and one other important item *episodes*. An episode is a literal record of each formula submitted to CGP in the order in which they were submitted.

2.6.1 The Type And Relational Lattices

Type lattice and relational lattice relations are both of the form:

type1 << type2.

relation1 << relation2.

where *type1* and *relation1* are the subtype and subrelation of *type2* and *relation2* respectively. The use of the double arrow instead of Sowa's single arrow is simply to enable the parser to distinguish this arrow from those in graphs.

CGP generates these relations and maintains the lattices automatically from definitions and so there is normally no requirement to use this means of input. However, for small applications it is often convenient that this form be used. It should be noted, however, that for the system to add such new relations to the lattice, then *type2* or *relation2* must already exist within the appropriate lattice. If this were not the case then the system would not know to which lattice each new relation is to be added.

At the top of each lattice is the universal type or the universal relation. These are known to the system as *universal_type* and *universal_rel* respectively. In addition, there are the absurd type and the absurd relation attached to the bottom of the appropriate lattice. These are known to the system as *absurd_type* and *absurd_rel* respectively. There are some additional built-in types and relations which are included within the lattices. These are discussed in sections 2.9 and 2.10.

2.6.2 Conformity Relations

Conformity relations cannot be entered directly into CGP but must instead be entered in the form of individual concepts from which CGP extracts the conformity relation. Once extracted it is added to a conformity relations table.

This approach is reasonable in view of the semantics of Sowa's *::* operator. The *::* operator states that something conforms to a given type. For this to be the case then the entity must exist and that implies the existence of an individual concept.

2.6.3 Definitions

Definitions, such as type definitions and relational definitions are not implemented as such in CGP. Instead, they must be presented to CGP in a logically equivalent and computationally more tractable form; that of implications.

2.6.3.1 Type Definitions

Any type definition has the following Sowa form:

type TYPE(x) is

graph

where *graph* contains a concept with the variable as shown in the definition's heading in its referent field. This notation for a type definition is equivalent to the following form:

([*TYPE : *x*] (*graph*)).

(*graph* ([*TYPE : *x*])).

such that *graph* has a concept which is coreferent with [*TYPE : *x*]. Additionally, *graph* must either be bigger than a single concept or there must be additional negated simple graphs in conjunction with it.

All type definitions have the general form of a logical equivalence between a single concept and a bigger graph which contains a concept which is coreferent with the single concept. CGP will, however, recognise either implication when added some time after the addition of the other member of the pair and extract the type lattice relation from it automatically, whilst updating the type lattice as required. Also, whilst it extracts the type lattice relation, CGP also stores an equivalent lambda expression for the definition. This is used by the knowledge base maintenance system when new graphs are added.

2.6.3.2 Relational Definitions

Relational definitions also have an equivalent logical form. In this case, a graph containing a single relation, the relation being defined, implies some bigger graph such that each concept in the relation being defined is coreferent with a concept in the larger graph. As an example:

([*T1 : *x*]->(*R1*)->[*T2 : *y*]
(...[*T1 : *x*]->(*R2*)...(*Rn*)->[*T2 : *y*]...)).

(...[*T1 : *x*]->(*R2*)...(*Rn*)->[*T2 : *y*]...
([*T1 : *x*]->(*R1*)->[*T2 : *y*])).

This is a simple relational definition of the form described by Sowa.

In addition, it is possible to define a relational hierarchy, in which one relation is a subrelation of another. A simple example will suffice to demonstrate that such relations exist:

father is a kind of parent.

In this example, the *father* relation is a subrelation of the *parent* relation. To show this as a double implication, it is necessary to recognise that the concepts in the single relation graph will be coreferent with concepts on either side of a single relation in the larger graph. Thus the general form is:

([*T1 : *x*]->(*R1*)->[*T2 : *y*]
(... [*T1 : *x*]->(*R2*)->[*T2 : *y*] ...)).

(... [*T1 : *x*]->(*R2*)->[*T2 : *y*] ...
([*T1 : *x*]->(*R1*)->[*T2 : *y*])).

This defines the relation *R1* as a subrelation of *R2*. Therefore:

([*person : *x*]->(*father*)->[*person : *y*]
([*person : *x*]->(*parent*)->[*person : *y*]->(*gender*)->[*male*])).

([*person : *x*]->(*parent*)->[*person : *y*]->(*gender*)->[*male*]
([*person : *x*]->(*father*)->[*person : *y*])).

When CGP encounters either of these forms it automatically extracts the relational hierarchy relation.

There is an additional form that such definitions can take. This has the general form:

```
( [ T1 : *x ]->( R1 )->[ T2 : *y ]
        ( [ T3 : *x ]->( R2 )->[ T4 : *y ] ) ).

( [ T3 : *x ]->( R2 )->[ T4 : *y ]
        ( [ T1 : *x ]->( R1 )->[ T2 : *y ] ) ).
```

In this form, *R1* is a subrelation of *R2* if either or both of *T3* and *T4* are subtypes of *T1* and *T2* respectively. This is the case because if either *T3* or *T4* were expanded until the types of the concepts that contain them were *T1* and *T2* respectively then the graph containing *R2* would be of exactly the same general form as before. An additional variation on this theme is given by the following case:

```
( [ T1 : *x ]->( R1 )->[ T2 : *y ]
        ( [ T1 : *x ]->( R2 )->[ T2 : *y ] ) ).

( [ T1 : *x ]->( R2 )->[ T2 : *y ]
        ( [ T1 : *x ]->( R1 )->[ T2 : *y ] ) ).
```

In this case *R1* is a subrelation of *R2* if the graph containing *R1* is the canonical graph for *R1* and the graph containing *R2* is a specialisation of the canonical graph for *R2* (see 2.6.4).

With each form of relational definition, once both implications are known to the system, CGP will generate and store a lambda expression for the definition. This is used by the knowledge base maintenance system when new graphs are added.

2.6.4 The Canon

The canon is a catalogue of graphs that gives the most general acceptable use of each relation. These graphs are requested by the system during run-time and cannot be entered directly. Graphs within this catalogue are not existentially quantified, but provide models for acceptable formats of each relational label known to the system. Section 2.10 describes CGP's built-in canonical basis.

Canonical graphs come in two forms depending upon the mode of operation of the system. In one mode each relation that is known to the system has associated with it a single canonical graph which models the most general unambiguous use and the arity of the relation. In the other mode there is an additional graph which specifies the order of the relation. Each mode reflects a different view of the meaning of arity and order.

In the standard mode each relation has a single canonical graph corresponding to it. For any case of that relation within an input formula to be valid it must be specialisation of the canonical graph and have the same arity. As an example, the canonical graph:

```
[ person ]->( parent )->[ person ]
```

allows the following graph to be considered valid:

```
[ man ]->( parent )->[ man ]
```

In the extended mode each relation has in addition to the canonical graph that it has in the standard mode (the *upper bound)* an optional second graph which models the maximum arity allowed for that relation and in addition the maximum number of concepts of any type that can take part in the relation (the *lower bound).* The following would be the lower bound for the *parent* relation:

```
( parent )-
        <-[ person : @many ]
        ->[ man ]
        ->[ woman ]
```

With this graph the system would disallow the following:

(*parent*)-
<-[*person* : *John*]
->[*man* : *Albert*]
->[*man* : *Brian*]

2.6.5 The Lexicon

The lexicon is a catalogue of relations which bind words in the world being modelled to the internal markers used by the system. As such, all these relations are allocated by the system at run-time and cannot be specified by the user. The user can, however, peruse the lexicon, in which case the lexical relations are displayed in the form:

```
word -> #marker
```

Section 2.10 describes CGP's built-in canonical basis.

2.7 Formatting Graphs

Graphs may be formatted as they are entered at the keyboard or when they are typed into a text file. The advantages of formatting graphs, especially complex negated contexts, is that errors can be more easily seen and corrected. CGP does not attach any relevance to the format of any graph.

Graphs are formatted by the addition of extra spaces, tabs and carriage returns. These characters are only significant when included in labels that are surrounded by single quotes, and so such labels cannot be formatted. All other such characters are stripped when a graph is parsed.

It is recommended that lists of relations attached to a concept should be indented with tab characters and that each branch of a graph be terminated with a carriage return. Also, graphs nested within negated contexts should be similarly listed and indented to show the depth of nesting.

Some machines put an upper limit on the number of characters that can be accepted before RETURN is pressed. If this is the case, then to continue entering a long graph, RETURN should be pressed once the limit is reached. Entry of a graph is only terminated by a full stop.

2.8 Text Files And Scripts

CGP can read data graphs from two sources: the keyboard and text files. A text file is a file produced by CGP when saving data. It contains graphs that are formatted in CGP's internal form. How to submit text files and scripts to CGP is described in section 3.

2.8.1 Simple Text Files

A simple text file is a file that contains graphs, lattice relations, conformity relations, canonical graphs, lexical relations and other system data in any combination and in CGP internal form. In addition, it can contain comments, which are preceded by the % character. An example of a text file might be:

```
% Semantic Net

graph([rel([#(2)],cons([#(0)],[#(1)]),cons([#(0)],[#(3)])),rel([#(6)],cons([#(0)],[#(5)]),
cons([#(7)],[#(8)])),rel([#(4)],cons([#(0)],[#(3)]),cons([#(0)],[#(5)]))]).

% Rules

% Lattice

lattice(4,-2).
lattice(7,-1).
lattice(2,-2).
lattice(0,-1).
lattice(6,-2).
lattice(-4,4).
lattice(-4,6).
```

lattice(-3,7).
lattice(-4,2).
lattice(-3,0).
lattice(-3,-1).
lattice(-5,-1).
lattice(-6,-1).
lattice(-7,-1).
lattice(-3,-5).
lattice(-6,-5).
lattice(-7,-5).
lattice(-3,-6).
lattice(-3,-7).
lattice(-4,-2).
lattice(-8,-2).
lattice(-9,-2).
lattice(-10,-2).
lattice(-11,-2).
lattice(-12,-2).
lattice(-13,-2).
lattice(-4,-8).
lattice(-9,-8).
lattice(-10,-8).
lattice(-4,-9).
lattice(-4,-10).
lattice(-4,-11).
lattice(-12,-11).
lattice(-13,-11).
lattice(-4,-12).
lattice(-4,-13).

% Conformity Relations Table

conf(7,8).
conf(4,&(4)).
conf(7,&(2)).
conf(0,3).
conf(2,&(1)).
conf(0,&(0)).
conf(0,1).
conf(6,&(3)).
conf(0,5).

% Canon

canon(graph([rel([#(-8)],cons([#(-5)],[(-1)]),cons([#(-5)],[*(-2)])])]).*
canon(graph([rel([#(-11)],cons([#(-5)],[(-1)]),cons([#(-5)],[*(-2)])])]).*
canon(graph([rel([#(-9)],cons([#(-6)],[(-1)]),cons([#(-6)],[*(-2)])])]).*
canon(graph([rel([#(-12)],cons([#(-6)],[(-1)]),cons([#(-6)],[*(-2)])])]).*
canon(graph([rel([#(-10)],cons([#(-7)],[(-1)]),cons([#(-7)],[*(-2)])])]).*
canon(graph([rel([#(-13)],cons([#(-7)],[(-1)]),cons([#(-7)],[*(-2)])])]).*

% Lexicon

lex([universal_type],-1).
lex([universal_rel],-2).
lex([absurd_type],-3).
lex([absurd_rel],-4).
lex([label],-5).
lex([type],-6).
lex([rel],-7).
lex([sublabel],-8).
lex([subtype],-9).

```
lex([subrel],-10).
lex([superlabel],-11).
lex([supertype],-12).
lex([superrel],-13).
lex([person],0).
lex(['Sue'],1).
lex([brother],2).
lex(['John'],3).
lex([parent],4).
lex(['Edward'],5).
lex([gender],6).
lex([sex],7).
lex([male],8).

% Aliases

alias(-1,[-1]).
alias(-2,[-2]).
alias(-3,[-3]).
alias(-4,[-4]).
alias(-5,[-5]).
alias(-6,[-6]).
alias(-7,[-7]).
alias(-8,[-8]).
alias(-9,[-9]).
alias(-10,[-10]).
alias(-11,[-11]).
alias(-12,[-12]).
alias(-13,[-13]).
alias(0,[0]).
alias(1,[1]).
alias(2,[2]).
alias(3,[3]).
alias(4,[4]).
alias(5,[5]).
alias(6,[6]).
alias(7,[7]).
alias(8,[8]).

% Lambda Abstractions

% Internal Markers

imarker(9).
vmarker(0).
gmarker(5).
emarker(2).

% Episodes

episode(0,[graph([rel([#(2)],cons([#(0)],[#(1)]),cons([#(0)],[#(3)]))])]).
episode(1,[graph([rel([#(6)],cons([#(0)],[#(5)]),cons([#(7)],[#(8)])),rel([#(4)],cons([#(0)],[#(3)]),cons([
        #(0)],[#(5)]))])]).

end_of_file.
```

When CGP loads a file such as this there is no attempt to check the consistency of the data. It is intended that this facility should be used to reload data when CGP is restarted after a previous session in which the text file was produced by the CGP save command (>).

Such a text file can be modified at will before reloading. Note that the data contained within the text

file replace the data that were previously held within the system and so the - command need not be used before loading a new knowledge base if the knowledge within in it is to replace the previous knowledge base. The internal form is described in detail in Appendix III.

2.8.2 Scripts

A CGP script is similar to a text file except that a script contains valid CGP commands which are always executed. Thus, any data that are presented to CGP in this way are evaluated and any other commands are fully executed. This facility provides a batch processing capability, in which large graphs can be built up with a text editor and checked for errors before being submitted to CGP. Work is also saved because CGP can be left to extract all lattice relations and conformity relations. The loading of data in this way is slower than the loading of data from a text file.

An example of a script is:

\ *Example Script.*

! [*person* : *Sue*]->(*brother*)->[*person* : *John*].
! [*person* : *John*]->(*parent*)->[*person* : *Edward*]->(*gender*)->[*sex* : *male*].

end_of_file.

The first line is a comment (see the \ command in section 3.4) and the next two lines are assertions (section 3.11). The last line is a CGP command to indicate the end of a script. When submitted to CGP, this script would generate the same knowledge base as the previous example text file. Section 3 describes the meanings of the other available commands.

The last line of all scripts must be:

end_of_file.

With command scripts it may be necessary to split long graphs into sections with extra carriage returns otherwise any machine imposed limit on the number of characters that can be entered at one time may be exceeded and may cause a problem.

There are two special 'scripts' that CGP recognises. These are the scripts called *user* and *build_model* respectively. These are described fully in section 3.5. Because the system recognises these script names, it is not useful to use them as the names of user-defined scripts.

2.9 Metasyntax

CGP is capable of a degree of processing of metaknowledge. Metaknowledge is knowledge about knowledge and, as such, includes knowledge about basic relationships such as type/subtype relationships for example. Syntactically, metaknowledge is expressed as graphs instead of the traditional Sowa representation. This allows the system to treat assertions and queries about the information in the system's database in exactly the same way as the information itself.

The current system implements a selection of metaknowledge as metasyntax. Some of this has already been met in section 2.6 with the discussion about definitions, but there is also one extra area that is currently implemented. This area of metaknowledge concerns the type lattice. One way to state that type p is a subtype of type q (without giving a definition) is to state that p << q as described earlier. This is similar to the traditional Sowa syntax and CGP will accept this. Additionally, CGP will accept any relation of the form:

[*type* : *T1*]->(*subtype*)->[*type* : *T2*]

or

[*type* : *T2*]->(*supertype*)->[*type* : *T1*]

These are equivalent statements. Note that *type*, *subtype* and *supertype* as used in these graphs are actual labels, like *elephant*. When CGP encounters either of these forms it attempts to prove the equivalent

relationship:

T2 << T1

With this relationship expressed as a graph, CGP is able to communicate with the user entirely in conceptual graphs. In the future, CGP may support additional meta forms such:

[*type* : *t*]->(*definition*)->[*graph* : *<graph>*]

[*relation* : *r*]->(*canonical_graph*)->[*graph* : *<graph>*]

but as yet these are not supported.

Along with the above metatypes and metarelations, CGP understands other built_in types and relations. The complete list is:

Types:

universal_type
label
type
rel
absurd_type

Relations:

universal_rel
sublabel
superlabel
subtype
supertype
subrel
superrel
absurd_rel

2.10 CGP'S Built-In Canonical Basis

CGP contains a built-in *canonical basis.* A canonical basis is a set of graphs and other structures that form a basis on which a knowledge base can be built and which can be used as the standard by which the semantic correctness of input graphs can be judged. It consists of a type lattice, a relational lattice and a set of canonical graphs for the relations within the relational lattice.

The canonical basis of CGP is under constant review and may well differ from that below. The current supplement to this guide will give additional information as necessary.

2.10.1 Built-in Type Lattice

The built-in type lattice consists of the following relations:

label << universal_type
type << label
rel << label
type << universal_type
rel << universal_type
absurd_type << universal_type
absurd_type << label
absurd_type << type
absurd_type << rel

The type *label* is a metatype that is the type to which all type labels and relational labels conform. The subtypes of *label* are *type* and *rel.* All type labels conform to *type* and all relational labels conform to *rel.*

CGP maintains a lattice in which all lattice relations are included explicitly. This reduces the search time for lattice relations between types that are not directly linked and is therefore an efficiency measure.

2.10.2 Built-in Relational Lattice

The built-in relational lattice consists of the following relations:

sublabel << universal_rel
subtype << universal_rel
subrel << universal_rel
superlabel << universal_rel
supertype << universal_rel
superrel << universal_rel
subtype << sublabel
subrel << sublabel
supertype << superlabel
superrel << superlabel
absurd_rel << universal_rel
absurd_rel << sublabel
absurd_rel << subtype
absurd_rel << supertype
absurd_rel << superlabel
absurd_rel << supertype
absurd_rel << superrel

The relation *sublabel* is the metarelation which is the relation of which all relations that specify a superrelation/subrelation relation are a subrelation. The relation *superlabel* is the metarelation which is the relation of which all relations that specify a subrelation/superrelation relation are a subrelation.

As with the type lattice, the relational lattice contains all lattice relations explicitly.

2.10.3 Built-in Canonical Graphs

The built-in canonical graphs define the correct usages of the built-in relations. The catalogue is:

[*type*]->(*subtype*)->[*type*]

[*type*]->(*supertype*)->[*type*]

[*rel*]->(*subrel*)->[*rel*]

[*rel*]->(*superrel*)->[*rel*]

[*label*]->(*sublabel*)->[*label*]

[*label*]->(*superlabel*)->[*label*]

Each relation has only one canonical graph as there is no limit to the number of concepts that can be attached to either side of the relation.

SECTION 3 - CGP COMMANDS

The format of all CGP commands is very simple. They are all a single character followed by either a set of conceptual graphs (which is a positive context to be treated as a whole), or some other parameter. All commands must end with a full stop, and none are allowed to contain any other full stops.

3.1 The < Command

Syntax:

<filename.

Description:

The < command is used to load data from a text file stored in the file *filename.* Such a file contains graphs, type lattice relations and conformity relations and other data in internal form, and is prepared either by the > command (3.2) or with a standard text editor. Data entered in this manner are added directly to the knowledge base and are not checked for consistency in any way.

The main purpose of this command is to reload a knowledge base that has previously been saved. It should be noted that this command starts by resetting the system to its initial state and that the data within the file replace those already in the system. A more sound way of entering new data is by the use of the % command, however, the < command is useful for bypassing the data checking mechanism in cases where this is slow.

Caveat:

The data within the text file completely replace that which was already within the system. Important data must therefore be saved before using this command.

Associated Commands:

>, %.

3.2 The > Command

Syntax:

>filename.

Description:

The > command writes the contents of the knowledge base out to the file *filename.* The file is a simple text file and is of exactly the form necessary for reloading with the < command (3.1). Such files can be edited with a standard text editor.

Associated Commands:

<, %.

3.3 The . Command

Syntax:

.

Description:

The . command exits CGP and returns the system to the Prolog prompt. CGP can be resumed by typing:

cgp.

This may occasionally be necessary if certain host system commands cause CGP's + command (3.8) to fail. Once the Prolog system prompt has appeared, the following goal can be evaluated to issue arbitrary host system commands:

```
unix(system('command')).
```

This goal only applies to CGP run on a Unix machine. Other Prologs will have similar goals.

Associated Commands:

^.

Restrictions:

The . command is not available to users of the PC version of CGP.

3.4 The \ Command

Syntax:

text.

Description:

The \ command causes CGP to ignore the rest of the command (up to the next full stop), which is thus treated as a comment. It is only of use within command scripts (3.5).

Associated Commands:

>, <, %.

3.5 The % Command

Syntax:

%*filename*.

Description:

The % command executes the file *filename* as a set of commands. Each line in the file, or *script*, must therefore be a valid CGP command. All CGP commands are available from within a script, including the % command. If the filename *user* is used then control is returned to the user at the keyboard. The last line of a script must be:

end_of_file.

When CGP is executing a script, the prompt changes from

>>

to

filename:>>

to reflect the fact. If the command *%user* had been used then the prompt is

user:>>.

When a command script is being run, all the interactive parts of the system are disabled (selectional constraint violation checking and the Help system). This is necessary because the whole point of command

scripts is that the system should be left to itself. In addition, the use of the *%user* command allows for the interactive parts of the system to be disabled whilst still retaining terminal control. This can be convenient when the interactive parts of the system become tedious.

The special script name build_model is reserved for the rebuilding of the model from the set if instances recorded in the episodic memory. When the command *%build_model* is issued, the parts of the knowledge base that depend upon the logic of the statements made to CGP are deleted and the *episodes* are used as new statements from which a new model is built. The episodes are read in reverse order to that in which CGP received them originally, therefore the model is rebuilt with the most recent version of the facts. This allows any previously entered fact to be contradicted at a later stage and the model to be changed in a consistent way to reflect this.

If a command within a script causes an error then the error menu is summoned and control is handed to the user for the duration of the error condition. This facility is not available to users of the PC version of CGP.

Associated Commands:

>, <.

3.6 The * Command

Syntax:

*.

Description:

The * command displays the contents of the knowledge base. The format is identical to that used by the > command (2).

It is possible to list parts of the knowledge base independently by attaching one of the folowing parameters to the * command:

graphs,
rules,
lattice,
conf,
canon,
lex,
lambda,
frames,
episodes,
factors,
scales,
heads.

In each case only the indicated part of the knowledge base is displayed. This is convenient for any knowledge base that is more than trivial in size.

A further option is that of including a search string parameter. In this case the syntax is:

**\string*

The \ character is interpreted as as escape character to indicate that the search string *<string>* is intended as such. When the * command is used in this way then each structure that contains this string is displayed. If there are no matches the message *search string <string> not found* is issued. Structures that might be listed are: graphs, negated contexts, lattice relationships, conformity relations, canonical graphs, lambda abstractions and episodes.

Associated Commands:

None.

3.7 The - Command

Syntax:

Description:

If the - command is used without supplying any parameter then the whole knowledge base is erased and reset to its initial state, including the options set by the *&*, *s*, *v*, *p* and *i* commands. Optionally, the - command can take a numeric parameter, in which case this parameter specifies the number of an episode to be erased.

Associated Commands:

*, <, >, %, !, o, &, s, v, p, i, g, n, l, d, t, u, b.

3.8 The + Command

Syntax:

+command.

Description:

The + command allows commands to be passed to the host operating system, with restrictions. The restrictions concern the use of full stops within commands, which is not allowed. Otherwise, any valid host command can be executed in this way. Note that the full stop that is required by the + command is stripped before the command is passed to the host.

Associated Commands:

3.9 The ^ Command

Syntax:

Description:

The ^ command causes CGP to terminate, with the loss of all data. All data should therefore be saved before using this command.

Associated Commands:

3.10 The $ Command

Syntax:

$graphs.

Description:

The *$* command allows data to be added to the system without being evaluated first. This should be used in conjunction with the *%build_model* command to provide a quick method of adding a new, contradictory graph and rebuilding a consistent model that takes this graph into account. Data added in this way are added to the list of episodes.

Associated Commands:

!, ?.

3.11 The ! Command

Syntax:

!*graphs*.

Description:

The *!* command is used to tell CGP of some new fact. The parameter *graphs* can be any valid positive context or any valid type/relational lattice relation. In addition, all of these data structures can be mixed in the same command, in which case, each must be separated with the semicolon, the final one being terminated by a full stop.

Each graph entered is passed to the evaluation mechanism in order to test whether CGP already knows about it or not. There are four possible outcomes to an attempt to tell CGP something. These are:

The graph is TRUE and no further action is taken,

The graph is FALSE and no further action is taken,

The graph is INCONSISTENT and no further action is taken,

The graph is UNKNOWN and is added to the knowledge base.

If several graphs are included in the command then they are evaluated as a complete context, with possible instantiations of generic referents being propagated to coreferent concepts and relations throughout the context.

Associated Commands:

$, ?.

3.12 The ? Command

Syntax:

?*param*.

Description:

The *?* command can take two forms of parameter. This can be either *?* or any valid context. If the parameter is *?* then the help menu is summoned. The operation of the help system is self evident and will not be discussed further.

The *?* command in which *param* is a context is used to ask questions of CGP. The questions can be either simple ones consisting of single graphs or more complex ones consisting of arbitrarily complex contexts. There are four possible outcomes to a query:

The query is TRUE and any instantiations of generic referents are given,

The query is FALSE,

The query is UNKNOWN,

The query is inconsistent within the knowledge base.

For the first two cases, several different results will be given if the query contains any generic referents and there is more than one way to instantiate them. Any query must be only one of TRUE, FALSE, UNKNOWN or INCONSISTENT therefore a mixture of answers is not possible unless the knowledge base is inconsistent.

An explanation of the reasoning is given following the reporting of the truth value of the query. The messages that are produced are given in Appendix I.

Note that the behaviour of this command is affected by the the *&* command.

Associated Commands:

-, !, $, &.

3.13 The & Command

Syntax:

&.

Description:

The *&* command toggles between different modes of interaction with CGP. There are two modes: *findall* and *single solution*. When the *&* command is used CGP responds with either:

```
interaction = findall
```

or:

```
interaction = single solution
```

as appropriate.

In findall mode, CGP searches for each solution to a query before reporting any proofs. CGP then reports each proof, which, if there are many, will take some time. In cases where each solution is required, findall mode is the mode of choice.

In single solution mode, CGP reports each solution as it finds it and then prompts the user if any more solutions are required. If more solutions are required then the response must be 'y.'. The response must be in lower case and consist of a single y.

Note that the % command (3.5) always changes to findall mode. If this were not done then it would be impossible to run any script for which the number of proofs were not previously known. After the % command has terminated, the mode is restored to its previous state.

Default Value:

single

Associated Commands:

-, ?, s, v, p, i, c, o, g, n, l, d, t, u, b.

3.14 The s Command

Syntax:

s.

Description:

The *s* command toggles between two speeds of operation of the system. In *slow* mode the system produces each proof of each result of a query whereas in *quick* mode the system will find each result but only one proof for each. When the *s* command is used the system responds with either:

```
speed = slow
```

or

```
speed = quick
```

The use of quick mode can significantly reduce the times of proofs, therefore, but has the effect that only one proof for each answer is given. As there is no means of knowing which proof the system will find first there is no guarantee that the proof offered will be the simplest or most intuitive.

Default Value:

quick

Associated Commands:

-, !, ?, &, v, p, i, c, o, g, n, l, d, t, u, b.

3.15 The v Command

Syntax:

v.

Description:

The *v* command toggles the system between two levels of verbosity of it explanations. In *verbose* mode the system gives full explanations of its reasoning whilst in *brief* mode only the facts that proved the query, and hence any instantiations of variables, are reported. When the *v* command is used the system responds with either:

```
report = brief
```

or

```
report = verbose
```

Default Value:

verbose

Associated Commands:

-, !, ?, &, s, p, i. c, o, g, n, l, d, t, u, b.

3.16 The p Command

Syntax:

p.

Description:

The *p* command toggles the system between two levels of sophistication. In *full* mode the system uses all of its reasoning powers whilst in *Prolog* mode only the Prolog-like deduction mechanism is available. This allows reasoning by *modus ponens* and *modus tollens* to be used. When the *p* command is used the system responds with either:

```
evaluation = full
```

or

```
evaluation = partial
```

Caveat:

When used in conjunction with the *l* command this command may allow inconsistent knowledge to be entered into the knowledge base.

Default Value:

full

Associated Commands:

-, !, ?, &, s, v, i, c, o, g, n, l, d, t, u, b.

3.17 The i Command

Syntax:

i.

Description:

The *i* command allows the theorem prover's rule of iteration to be disabled. This facility may be useful because the current system may suffer from unduly reduced efficiency when several related rules are required to prove a theorem. When the *i* command is used the system responds with either:

```
iteration on
```

or

```
iteration off
```

Caveat:

When used in conjunction with the *l* command this command may allow inconsistent knowledge to be entered into the knowledge base.

Default Value:

on

Associated Commands:

-, !, ?, &, s, v, p, c, o, g, n, l, d, t, u, b.

3.18 The c Command

Syntax:

c.

Description:

The *c* command toggles the system between a mode in which it asks the user if any graph that it cannot itself prove is known to the user and a mode in which this does not occur. In this way, it is possible that the system can continue with a proof that would have otherwise been impossible. When the *c* command is issued the system responds with either:

```
check with user on
```

or

```
check with user off
```

Default Value:

off

Associated Commands:

-, !, ?, &, s, v, p, i, o, g, n, l, d, t, u, b.

3.19 The O Command

Syntax:

o.

Description:

The *o* command displays the current settings of the operational modes. An example of its output might be:

```
Speed                = quick (toggle: s.)
Report               = verbose (toggle: v.)
Interaction          = single solution (toggle: &.)
Iteration            = on (toggle: i.)
Evaluation           = full theorem prover (toggle: p.)
Evaluation           = deductive (toggle: l.)
Direction            = goal driven (toggle: d.)
Update after query = off (toggle: u.)
Bounds equal         = on (toggle: n.)
Proof by game        = off (toggle: g.)
Theory generation    = off (toggle: t.)
Timeout limit        = 100 (alter: @n.)
Check with user      = off (toggle: c.)
```

Associated Commands:

-, ?, &, s, v, p, i, c, g, n, l, d, t, u, b.

3.20 The z Command

Syntax:

z<parameter>.

Description:

The *z* command is an undefined command, possibly taking a parameter, which is used for the temporary implementation of features under development. At any one time there may be any number of functions associated with this command. In general, it is best to ignore this command.

From time to time, the supplement to the user guide will contain information about the *z* command.

Associated Commands:

undefined.

3.21 The f Command

Syntax:

f.

Description:

Frames are used by CGP to simplify the entry of information with a standard format. The *f* command allows the user to either add a new frame or to instantiate an existing frame. When the *f* command is used a submenu is presented from which either of two options, *a* or *i*, can be selected.

The *a* option allows the addition of a new frame. A graph which represents the frame is requested and then the referent of the concept whose type label is to be used to identify the frame is requested. Before being accepted, the frame is evaluated to ensure its semantic and logical correctness.

The *i* option allows the user to select a frame for instantiation and addition to the knowledge base. After asking for the name of the frame (there may be more than one frame for any one type) the system asks for individual referents for each concept attached to each relation in turn. When finished the frame is evaluated and added to the knowledge base if appropriate.

Associated Commands:

!.

3.22 The g Command

Syntax:

g.

Description:

The *g* command toggles the theorem prover between two modes of operation in which whole negated contexts are proved by direct projection into the knowledge base or not. This may speed up some proofs slightly.

Default Value:

off

Associated Commands:

-, !, ?, &, s, v, p, l, c, o, n, l, d, t, u, b.

3.23 The n Command

Syntax:

n.

Description:

The *n* command toggles the semantic constraint checking mechanism between the state where it uses a single canonical graph to test relations for the presence of the correct concepts only and where it

also uses another canonical graph to test the relations for arity violations.

Default Value:

on (simple semantic constraint test)

Associated Commands:

-, ?, &, s, v, p, i, c, o, g, l, d, t, u, b.

3.24 The l Command

Syntax:

l.

Description:

The *l* command toggles the system between a state in which the lookup procedure is the only form of proof of simple graphs and a state in which the deductive mechanism is also active. When lookup only is used the system will run much faster but is not guaranteed to find all solutions.

Default Value:

off

Associated Commands:

-, !, ?, &, s, v, p, i, c, o, g, n, d, t, u, b.

3.25 The d Command

Syntax:

d.

Description:

The *d* command toggles the system between a state in which it is purely goal driven and one in which it is data driven. In the goal driven state the system only makes deductions when asked a query and only makes those deduction which are relevant. In the data driven mode it makes all deductions each time new information is added. Querying will be faster in data driven mode but data entry will be faster in goal driven mode.

Caveat:

Data driven mode may enter an infinite loop in situations where the product of the generation of data results in the production of more data of the same kind.

Default Value:

goal

Associated Commands:

-, !, ?, &, s, v, p, i, c, o, g, n, l, t, u, b.

3.26 The t Command

Syntax:

t.

Description:

The *t* command allows the activation of the theory generation mechanism. When active, this mechanism will generate possible worlds from its knowledge base and will attempt to answer queries by using the possible worlds produced.

Default Value:

off

Associated Commands:

-, !, ?, &, s, v, p, i, c, o, g, n, l, d, u, b.

3.27 The u Command

Syntax:

u.

Description:

By default, after completing a proof, the system throws away any information that it generates during the course of the proof. Any such information that may be needed by a later proof must be regenerated each time. The *u* command switches the system into a mode in which such information is not thrown away but is added to the knowledge base and is therefore available for lookup at a later stage. Some proofs will run more quickly as a result of the extra information being already available but if the deductive mechanism is invoked such information will still be regenerated and will result in duplicate solutions.

Default Value:

off

Associated Commands:

-, !, ?, &, s, v, p, i, c, o, g, n, l, d, t, b.

3.28 The b Command

Syntax:

b.

Description:

The *b* command toggles the unification algorithm between a state in which it allows two different generic objects to unify with the same individual object and a state in which it does not.

Default Value:

off

Associated Commands:

-, ?, &, s, v, p, i, c, o, g, n, l, d, t, u.

3.29 The m Command

Syntax:

m.

Description:

The *m* command invokes a submenu that allows the addition of 'scales' and 'factors'. A scale is a set of values which can be used as values within the measure object. A factor allows a value of one unidimensional unit to be equated with an equivalent value of another unidimensional unit.

Scales must be entered as values separated by commas and at most one space, terminated by a full stop. Factors are entered in the form:

value1 unit1 = value2 unit2.

Where all spaces are single and are significant.

Associated Commands:

None.

3.30 The @ Command

Syntax:

@n.

Description:

The @ command sets the number of major proof steps *n* that the system will attempt before giving up a proof. Such steps include the initiation of the proof of a simple graph, each loop around the theorem prover and the generation of theories. The value of *n* can be anything but only numerical values >=0 will specify a valid limit; other values turn the feature off and all proofs will continue until they terminate naturally.

When used from within a script, including *user* and *build_model*, the effect of reaching the specified limit is to cause immediate termination. When used from without a script the system pauses with a request to continue or abandon the current proof. If the user replies with *y* then a further period of *n* steps will be executed.

Default Value:

100

Associated Commands:

!, ?.

SECTION 4 - USING THE PROGRAM

This section provides a tutorial for users new to the program. It assumes that the reader has a basic understanding of conceptual graph theory and that the previous sections of this guide have been read. The discussions assume that CGP is running and is in its initial state, with *single solution*, *verbose*, *full* and *quick* modes set. This being the case, the screen should be displaying a banner message similar to:

```
Welcome to CGP version 4.0.

For help or latest information, enter ?? followed by a full stop
at the prompt (>>).

>>
```

It is worth trying out the built-in help facility at this point. This is summoned with the *??* command as stated n the banner. When this command is issued a help menu appears and a new prompt is shown. Try one or two options and then select option *e* to return to the main prompt.

Many of the examples given in this section include CGP's explanatory output. Generally, only one example of this is given here, but some of the examples will actually produce more than one proof. Also, the form of the explanations in this section may differ slightly to that of actual output from a CGP run. A full list of of CGP's messages is given in Appendix I.

At this point there is no knowledge at all in CGP's knowledge base except for the basic type lattices and canonical basis. This can be verified by issuing the * command to display the contents of the knowledge base. If this is done with CGP in its initial state, the following will be displayed:

```
>> *.

Graphs:

(subtype)
        <-[type:type]
        <-[type:rel]
        ->[type:absurd_type];
(subtype)
        <-[type:label]
        ->[type:absurd_type]
        ->[type:type]
        ->[type:rel];
(subtype)
        <-[type:universal_type]
        ->[type:absurd_type]
        ->[type:label]
        ->[type:type]
        ->[type:rel].

(subrel)
        <-[rel:subtype]
        <-[rel:subrel]
        <-[rel:superlabel]
        <-[rel:supertype]
        <-[rel:superrel]
        ->[rel:absurd_rel];
(subrel)
        <-[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel];
```

```
(subrel)
        <-[rel:sublabel]
        ->[rel:absurd_rel]
        ->[rel:subtype]
        ->[rel:subrel];
(subrel)
        <-[rel:universal_rel]
        ->[rel:absurd_rel]
        ->[rel:sublabel]
        ->[rel:subtype]
        ->[rel:subrel]
        ->[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel].
```

```
Rules:

(
     [type:*-1]-
             (subtype)->[type:*-2];
     [type:*-2]-
             (subtype)->[type:*-1]
).

(
     [absurd_type:*-1]
).

(
     (absurd_rel)
).
```

```
Type lattice:

absurd_type << universal_type.
label << universal_type.
type << universal_type.
rel << universal_type.
absurd_type << label.
type << label.
rel << label.
absurd_type << type.
absurd_type << rel.

Relational lattice:

absurd_rel << universal_rel.
sublabel << universal_rel.
subtype << universal_rel.
subrel << universal_rel.
superlabel << universal_rel.
supertype << universal_rel.
superrel << universal_rel.
absurd_rel << sublabel.
subtype << sublabel.
subrel << sublabel.
absurd_rel << subtype.
absurd_rel << subrel.
absurd_rel << superlabel.
supertype << superlabel.
superrel << superlabel.
absurd_rel << supertype.
```

```
absurd_rel << superrel.

Conformity relations:

Canon:

sublabel:
[label:*-1]-
        (sublabel)->[label:*-2].

superlabel:
[label:*-1]-
        (superlabel)->[label:*-2].

subtype:
[type:*-1]-
        (subtype)->[type:*-2].

supertype:
[type:*-1]-
        (supertype)->[type:*-2].

subrel:
[rel:*-1]-
        (subrel)->[rel:*-2].

superrel:
[rel:*-1]-
        (superrel)->[rel:*-2].

Lexicon:

universal_type -> #-1.
universal_rel -> #-2.
absurd_type -> #-3.
absurd_rel -> #-4.
label -> #-5.
type -> #-6.
rel -> #-7.
sublabel -> #-8.
subtype -> #-9.
subrel -> #-10.
superlabel -> #-11.
supertype -> #-12.
superrel -> #-13.

Lambda abstractions:

Frames:

Episodes:

Factors:

Scales:

Heads:

>>
```

During the following sections, a small knowledge base containing family history information about the Lattice family will be built up. Because one section leads on to the subsequent sections it is necessary to follow each through completely, otherwise knowledge will be missing from the system when the later stages are tried.

4.1 Adding Simple Knowledge

The first section shows how to add simple graphs to the knowledge base, and how CGP stores these graphs in its semantic net. Before following the examples in this section the command:

```
>> %user.
```

should be issued to set the system so that the checking of the canonicity of input graphs is disabled. This is necessary at this stage to simplify the explanations that follow.

Simple graphs are added to the knowledge base by the use of the *!* command. Try the following:

```
user:>> ![person:John]->(parent)->[person:Edward].
```

There may be a short delay whilst two things happen. These are:

* the query is parsed,
* the graph is evaluated.

The evaluation consists of two parts: checking that the graph does not violate selectional constraints and checking that the graph does not contradict the existing knowledge base, but, as stated, the full checking for selectional constraint violations is disabled by the *%user* command (although messages are issued). After entry of the above command CGP produces the following output:

```
conformity relation added:
     person :: &.

lattice relation added:
     person << universal_type.

lattice relation added:
     absurd_type << person.

conformity relation added:
     parent :: &.

lattice relation added:
     parent << universal_rel.

lattice relation added:
     absurd_rel << parent.

Checking for selectional constraint violations in:
     [person:John]-
             (parent)->[person:Edward].

Checking denotation of:
     [person:John]-
             (parent)->[person:Edward].

I did not know that:
     [person:John]-
             (parent)->[person:Edward].
```

```
Tidying up...

conformity relation added:
     person :: John.

conformity relation added:
     person :: Edward.

user:>>
```

It is not necessary at this stage to study this output in depth, but it is sufficient to say that CGP produces a record of all the data that it is storing.

The following command should now be entered:

```
user:>> ![person:John]->(parent)->[person:Rita].
```

CGP responds with:

```
checking for selectional constraint violations in:
     [person:John]-
             (parent)->[person:Rita].

Checking denotation of:
     [person:John]-
             (parent)->[person:Rita].

I did not know that:
     [person:John]-
             (parent)->[person:Rita].

Tidying up...

conformity relation added:
     person :: Rita.

user:>>
```

Once again, CGP responds with a comprehensive description of its actions. The use of the * command to list the contents of the knowledge base results in something like:

```
Graphs:

[rel:parent]-
        (subrel)->[rel:absurd_rel];
[rel:universal_rel]-
        (subrel)->[rel:parent];
(subrel)
        <-[rel:subtype]
        <-[rel:subrel]
        <-[rel:superlabel]
        <-[rel:supertype]
        <-[rel:superrel]
        ->[rel:absurd_rel];
(subrel)
        <-[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel];
(subrel)
        <-[rel:sublabel]
        ->[rel:absurd_rel]
        ->[rel:subtype]
        ->[rel:subrel];
```

```
(subrel)
        <-[rel:universal_rel]
        ->[rel:absurd_rel]
        ->[rel:sublabel]
        ->[rel:subtype]
        ->[rel:subrel]
        ->[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel].

[type:person]-
        (subtype)->[type:absurd_type];
[type:universal_type]-
        (subtype)->[type:person];
(subtype)
        <-[type:type]
        <-[type:rel]
        ->[type:absurd_type];
(subtype)
        <-[type:label]
        ->[type:absurd_type]
        ->[type:type]
        ->[type:rel];
(subtype)
        <-[type:universal_type]
        ->[type:absurd_type]
        ->[type:label]
        ->[type:type]
        ->[type:rel].

[person:John]-
        (parent)->[person:Rita]
        (parent)->[person:Edward].

Rules:

(
     [type:*-1]-
             (subtype)->[type:*-2];
     [type:*-2]-
             (subtype)->[type:*-1]
).

(
     [absurd_type:*-1]
).

(
     (absurd_rel)
).

Type lattice:

person << universal_type.
absurd_type << person.
absurd_type << universal_type.
label << universal_type.
type << universal_type.
rel << universal_type.
absurd_type << label.
type << label.
rel << label.
```

```
absurd_type << type.
absurd_type << rel.

Relational lattice:

parent << universal_rel.
absurd_rel << parent.
absurd_rel << universal_rel.
sublabel << universal_rel.
subtype << universal_rel.
subrel << universal_rel.
superlabel << universal_rel.
supertype << universal_rel.
superrel << universal_rel.
absurd_rel << sublabel.
subtype << sublabel.
subrel << sublabel.
absurd_rel << subtype.
absurd_rel << subrel.
absurd_rel << superlabel.
supertype << superlabel.
superrel << superlabel.
absurd_rel << supertype.
absurd_rel << superrel.

Conformity relations:

person :: Edward.
person :: John.
person :: &.
parent :: &.
person :: Rita.

Canon:

sublabel:
[label:*-1]-
        (sublabel)->[label:*-2].

superlabel:
[label:*-1]-
        (superlabel)->[label:*-2].

subtype:
[type:*-1]-
        (subtype)->[type:*-2].

supertype:
[type:*-1]-
        (supertype)->[type:*-2].

subrel:
[rel:*-1]-
        (subrel)->[rel:*-2].

superrel:
[rel:*-1]-
        (superrel)->[rel:*-2].
```

```
Lexicon:

universal_type -> #-1.
universal_rel -> #-2.
absurd_type -> #-3.
absurd_rel -> #-4.
label -> #-5.
type -> #-6.
rel -> #-7.
sublabel -> #-8.
subtype -> #-9.
subrel -> #-10.
superlabel -> #-11.
supertype -> #-12.
superrel -> #-13.
person -> #1.
John -> #2.
parent -> #3.
Edward -> #4.
Rita -> #5.

Lambda abstractions:

Frames:

Episodes:
0
[person:John]-
        (parent)->[person:Edward].

1
[person:John]-
        (parent)->[person:Rita].

Factors:

Scales:

Heads:


user:>>
```

An important point to note is that CGP always re-displays the input graph at each stage of its processing. This allows the graph to be checked to see if CGP's parse is equivalent to that typed.

4.2 Simple Queries

The knowledge base now contains enough knowledge to allow some simple queries to be made. Queries are made by using the *?* command followed by a graph that is required to be proved. The result of the proof of a graph is that CGP responds with the graph's denotation. This consists of the truth value of the graph, the instances for which the graph is TRUE and a list of other facts that CGP used in the proof.

An example will make this clear. Try the following query:

```
user:>> ?[person:John]->(parent)->[person:Edward].
```

CGP responds with:

```
Checking for selectional constraint violations in:
     [person:John]-
             (parent)->[person:Edward].
```

```
Checking denotation of:
     [person:John]-
             (parent)->[person:Edward].

The statement:
     [person:John]-
             (parent)->[person:Edward].

is:

TRUE because:
   1/0.      proof of:
                [person:John]-
                        (parent)->[person:Edward];
   2/0.      is:
   3/1.           fact:
                     [person:John]-
                             (parent)->[person:Edward].

user:>>
```

This is the simplest kind of query that can be made of any system; a query that exactly matches some fact in the knowledge base. Basically, this query was answered by *projecting* the query into the semantic net. As long as the projection operation succeeded the query was true and the projection provided the explanation. At the end of the proof CGP reports its reasoning. The numbers down the left give an indication of the depth of nesting of each part of the proof. By this is meant that depth 0 represents the original query whilst other numbers represent complete proofs of subparts of the original.

The next example deals with a case that is often seen at first as giving the wrong answer. The query is:

```
>> ?[person:Edward]->(parent)->[person:John].
```

The answer to this is, according to CGP:

```
Checking for selectional constraint violations in:
     [person:Edward]-
             (parent)->[person:John].

Checking denotation of:
     [person:Edward]-
             (parent)->[person:John].

The statement:
     [person:Edward]-
             (parent)->[person:John].
is:

UNKNOWN because of unknown graphs:

     [person:Edward]-
             (parent)->[person:John].

user:>>
```

The first reaction of many people to this is that the answer should be FALSE because if some person *x* is a parent of some person *y* then person *y* cannot be a parent of person *x*. Humans know this to be true, but CGP does not. This is not a fault in CGP or with logic itself, but a common problem with people. People always assume certain knowledge when answering queries and do not always appreciate that other people may not have that knowledge. In the case of this query, the missing knowledge is the rule that states that no person can be the parent of their own parent, or that the *parent* relation is *non-reflexive*. Until CGP is taught this fact it will always answer UNKNOWN to this and similar queries. People are the same.

It is possible to ask CGP some more general questions. If it is necessary to find all the parents of John then the following query is required:

```
user:>> ?[person:John]->(parent)->[person].
```

When CGP replies, the following is seen:

```
Checking for selectional constraint violations in:
     [person:John]-
            (parent)->[person:*1].

Checking denotation of:
     [person:John]-
            (parent)->[person:*1].

The statement:
     [person:John]-
            (parent)->[person:*1].
is:

TRUE because:
   1/0.     proof of:
               [person:John]-
                     (parent)->[person:*1];
   2/0.     is:
   3/1.          fact:
                    [person:John]-
                           (parent)->[person:Rita].

TRUE because:
   1/0.     proof of:
               [person:John]-
                     (parent)->[person:*1];
   2/0.     is:
   3/1.          fact:
                    [person:John]-
                           (parent)->[person:Edward].

user:>>
```

Each fact that makes the query true is reported. The implied generic referent has been allowed to pick up each possible individual referent within the semantic net that matches the query. The query:

```
user:>> ?[person]->(parent)->[person].
```

could also be made. In this case, within the current knowledge base, the reply from CGP would be the same as the previous example.

It would be possible to make a query such as:

```
user:>> ?[person:*x]->(parent)->[person:*x].
```

In this case the reply from CGP is UNKNOWN because the knowledge base knows nothing about people who are their own parents. The two **x* referents represent the same individual, so that once the projection operation has found a value for one of them the other acquires the same value. However, this stops the projection from proceeding. Once again, CGP knows nothing of any rule that prevents a person from being their own parent and so the answer FALSE is not produced.

It is possible to ask CGP what is the relationship between John and Edward. To do this, the query is:

```
user:>> ?[person:John]->(*relation)->[person:Edward].
```

This query has a variable in the relation node. After CGP has evaluated this graph it replies with:

```
Checking for selectional constraint violations in:
     [person:John]-
             (*2)->[person:Edward].

Checking denotation of:
     [person:John]-
             (*2)->[person:Edward].

The statement:
     [person:John]-
             (*2)->[person:Edward].
is:

TRUE because:
   1/0.      proof of:
                [person:John]-
                        (*2)->[person:Edward];
   2/0.      is:
   3/1.           fact:
                     [person:John]-
                             (parent)->[person:Edward].

user:>> end_of_file.

OK

>>
```

In other words, CGP has found out that the relationship between John and Edward is that Edward is John's parent. The ability to include variables within the relation field is very powerful. Variables can also be placed within the type field. More is said about this in section 4.6.6.

4.3 Canonicity Checking

Following the simple examples given in the previous section it is now necessary to unleash the full power of CGP on our input graphs. To do this it is necessary to enable the checking of relations within the input for violations of selectional constraints. At the same time, CGP updates its lattices and catalogues with any relevant information by means of an interactive conversational mechanism. To enable this awesome power to be used, and assuming that the system is still under the influence of the *%user* command, the command:

```
user:>> end_of_file.
```

should be given. The prompt returns to >>. Before attempting the following examples, reset the knowledge base with the - command.

4.3.1 Placing Of Newly Encountered Types Within The Lattices

The first stage of this process involves the placing of newly encountered types and relations within the lattices. For each unknown type or relational label within the graph, CGP follows a question and answer procedure to locate the exact position of each. This question and answer procedure is intelligent in that it is only invoked when it is not obvious where the new type or relation belongs and when it is used it attempts to minimise the number of questions asked. Therefore, if the statement:

```
>> ![person:John]
```

is submitted to CGP the following output is generated:

```
Checking for selectional constraint violations in:
     [person:John].
```

```
conformity relation added:
     person :: &.

lattice relation added:
     person << universal_type.

lattice relation added:
     absurd_type << person.
```

Notice that there are many things that CGP can work out for itself. Because the new type in the input graph is the only type that CGP knows about (apart from the primitive types) there is only one possible place for it within the hierarchy and the question and answer procedure is omitted in this case.

Following the checking of the type label, CGP continues with:

```
Checking denotation of:
     [person:John].

I did not know that:
     [person:John].

Tidying up...

conformity relation added:
     person :: John.

>>
```

The fact the CGP did not already know what it was told is not very surprising. Following the result, CGP displays a list of additional inferences that the knowledge base maintenance system has made. Try the * command again to find out how the knowledge base has changed. Something similar to the following should be displayed:

```
>> *.

Graphs:

[person:John].

(subrel)
        <-[rel:subtype]
        <-[rel:subrel]
        <-[rel:superlabel]
        <-[rel:supertype]
        <-[rel:superrel]
        ->[rel:absurd_rel];
(subrel)
        <-[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel];
(subrel)
        <-[rel:sublabel]
        ->[rel:absurd_rel]
        ->[rel:subtype]
        ->[rel:subrel];
```

```
(subrel)
        <-[rel:universal_rel]
        ->[rel:absurd_rel]
        ->[rel:sublabel]
        ->[rel:subtype]
        ->[rel:subrel]
        ->[rel:superlabel]
        ->[rel:supertype]
        ->[rel:superrel].

[type:person]-
        (subtype)->[type:absurd_type];
[type:universal_type]-
        (subtype)->[type:person];
(subtype)
        <-[type:type]
        <-[type:rel]
        ->[type:absurd_type];
(subtype)
        <-[type:label]
        ->[type:absurd_type]
        ->[type:type]
        ->[type:rel];
(subtype)
        <-[type:universal_type]
        ->[type:absurd_type]
        ->[type:label]
        ->[type:type]
        ->[type:rel].

Rules:

(
     [type:*-1]-
             (subtype)->[type:*-2];
     [type:*-2]-
             (subtype)->[type:*-1]
).

(
     [absurd_type:*-1]
).

(
     (absurd_rel)
).

Type lattice:

person << universal_type.
absurd_type << person.
absurd_type << universal_type.
label << universal_type.
type << universal_type.
rel << universal_type.
absurd_type << label.
type << label.
rel << label.
absurd_type << type.
absurd_type << rel.
```

```
Relational lattice:

absurd_rel << universal_rel.
sublabel << universal_rel.
subtype << universal_rel.
subrel << universal_rel.
superlabel << universal_rel.
supertype << universal_rel.
superrel << universal_rel.
absurd_rel << sublabel.
subtype << sublabel.
subrel << sublabel.
absurd_rel << subtype.
absurd_rel << subrel.
absurd_rel << superlabel.
supertype << superlabel.
superrel << superlabel.
absurd_rel << supertype.
absurd_rel << superrel.

Conformity relations:

person :: &.
person :: John.

Canon:

sublabel:
[label:*-1]-
        (sublabel)->[label:*-2].

superlabel:
[label:*-1]-
        (superlabel)->[label:*-2].

subtype:
[type:*-1]-
        (subtype)->[type:*-2].

supertype:
[type:*-1]-
        (supertype)->[type:*-2].

subrel:
[rel:*-1]-
        (subrel)->[rel:*-2].

superrel:
[rel:*-1]-
        (superrel)->[rel:*-2].

Lexicon:

universal_type -> #-1.
universal_rel -> #-2.
absurd_type -> #-3.
absurd_rel -> #-4.
label -> #-5.
type -> #-6.
rel -> #-7.
sublabel -> #-8.
subtype -> #-9.
```

```
subrel -> #-10.
superlabel -> #-11.
supertype -> #-12.
superrel -> #-13.
person -> #1.
John -> #2.

Lambda abstractions:

Frames:

Episodes:
0
[person:John].

Factors:

Scales:

Heads:


>>
```

This shows that the graph has been added to the knowledge base, but also that certain other data have also been added. Each new type that is used is extracted from the new graph and added to the appropriate lattice. As no type/subtype relationships are implicit in the knowledge base so far the only new type/subtype relations that can be added for new types are those that insert the new types into the lattice under the *universal* type. Each new individual marker is added to the conformity relations table and a new prototype marker is added for each new type. Other information gives the internal markers that have been allocated to each label.

4.3.2 Adding Subtypes Of Known Types

If CGP is now informed that John is a man with the command:

```
![man:John]
```

the interaction is:

```
>> ![man:John].

Is man a kind of person? y.

conformity relation added:
     man :: &.
lattice relation added:
     man << person.

lattice relation added:
     man << universal_type.

lattice relation added:
     absurd_type << man.

Enter definition for man in terms of person:
.

Checking for selectional constraint violations in:
     [man:John].
```

```
Checking denotation of:
     [man:John].

I did not know that:
     [man:John].

Tidying up...

conformity relation added:
     man :: John.

concept specialised:
     [person:John];
to:
     [man:John]
>>
```

When CGP encounters an unknown type for which it cannot automatically find the correct place within the attice, it asks a series of questions to find the immediate supertypes of the new type. Once a supertype has been located CGP then goes on to check if any of the subtypes of that supertype are also subtypes of the new type.

Notice that the request for a definition of man in terms of person is made by CGP. In this case the request is bypassed by entering the null response, a full stop. The reason for doing this at this point is that the definition would be evaluated in full and cause a recursive invocation of the facilities being explained in this section and would therefore complicate the explanation. However, the ability to bypass the definition request also allows the user to omit the entry of a definition at any time. For each new type/subtype relationship that CGP finds, a definition is requested. The reply to this request is given in the form of the *differentiae* of the type definition of the type. Thus, if it had been decided to enter a definition of a man as a person who is male then the following output would have been obtained (as can be verified by using the - command and re-entering the graphs [person:John] and [man:John]):

```
>> ![person:John]
...
>> ![man:John].

Is man a kind of person? y.

conformity relation added:
     man :: &.

lattice relation added:
     man << person.

lattice relation added:
     man << universal_type.

lattice relation added:
     absurd_type << man.
```

```
Enter definition for man in terms of person:
[person]->(attr)->[gender:male].

(
     [man:*1];
     (
          [person:*1]-
                  (attr)->[gender:male]
     )
);
(
     (
          [man:*1]
     );
     [person:*1]-
               (attr)->[gender:male]
).

Is this OK? y.

Checking for selectional constraint violations in:
     (
          [man:*1];
          (
               [person:*1]-
                       (attr)->[gender:male]
          )
     );
     (
          (
               [man:*1]
          );
          [person:*1]-
                    (attr)->[gender:male]
     ).

Is gender a kind of person? n.

conformity relation added:
     gender :: &.

lattice relation added:
     gender << universal_type.

lattice relation added:
     absurd_type << gender.

Is person a kind of gender? n.

conformity relation added:
     attr :: &.

lattice relation added:
     attr << universal_rel.

lattice relation added:
     absurd_rel << attr.
```

```
Is the general relation:

     [person:*2]-
             (attr)->[gender:*3].

the most general unambiguous usage? y.

Checking denotation of:
     (
          [man:*1];
          (
               [person:*1]-
                       (attr)->[gender:male]
          )
     );
     (
          (
               [man:*1]
          );
          [person:*1]-
                  (attr)->[gender:male]
     ).

I did not know that:
     (
          [man:*1];
          (
               [person:*1]-
                       (attr)->[gender:male]
          )
     );
     (
          (
               [man:*1]
          );
          [person:*1]-
                  (attr)->[gender:male]
     ).

Tidying up...

Checking for selectional constraint violations in:
     [man:John].

Checking denotation of:
     [man:John].

I did not know that:
     [man:John].

Tidying up...

conformity relation added:
     man :: John.

concept specialised:
     [person:John];
to:
     [man:John].

>>
```

The *differentiae* are checked as fully as any input graph, but the form that is used for this checking is that of :he double implication, which is generated automatically by CGP. After generating the double implication :he system ascertains if it has constructed the correct one. If it has not then either it will offer another ɔossibility or will request another definition altogether.

For each newly encountered relation, the system asks if the usage given is the most general ɹnambiguous usage for that relation. If not, and *n.* is given in response, a graph that gives its most general ɹniversally quantified use is requested. If the system is already aware of a usage of a relation and a new ɹsage is given then by answering 'y' to the question about whether that usage is allowed, CGP will be ɒlaced in a position to make its own generalisation of the known usage and the new usage. For simplicity, :he example above assumes that the usage of the *attr* relation is the most general unambiguous usage.

4.3.3 Adding New Relations

In the previous section the *attr* relation was added. As there were no other known relations the *attr* relation was added to the type lattice under *universal_rel*. The following example (which should be tried after issuing the - command) describes the addition of a subrelation to the system.

If the following graph is entered:

```
>> ![person:john]->(spouse)->[person:elizabeth].
```

:he following output is produced:

```
conformity relation added:
     person :: &.

lattice relation added:
     person << universal_type.

lattice relation added:
     absurd_type << person.

conformity relation added:
     spouse :: &.

lattice relation added:
     spouse << universal_rel.

lattice relation added:
     absurd_rel << spouse.

Checking for selectional constraint violations in:
     [person:john]-
             (spouse)->[person:elizabeth].

Is the general relation:
     [person:*0]-
             (spouse)->[person:*1].

the most general unambiguous usage? y.

Checking denotation of:
     [person:john]-
             (spouse)->[person:elizabeth].

I did not know that:
     [person:john]-
             (spouse)->[person:elizabeth].

Tidying up...
```

```
conformity relation added:
     person :: john.

conformity relation added:
     person :: elizabeth.

>>
```

As before, the system does as much of the work as it can without user intervention. It is now appropriate to use the *wife* relation and define it as a female spouse. To do this we will inform CGP that:

```
>> ![person:john]->(wife)->[woman:elizabeth].

Checking for selectional constraint violations in:
     [person:john]-
             (wife)->[woman:elizabeth].

Is woman a kind of person? y.

conformity relation added:
     woman :: &.

lattice relation added:
     woman << person.

lattice relation added:
     woman << universal_type.

lattice relation added:
     absurd_type << woman.

Enter definition for woman in terms of person:
[person]->(gender)->[sex:female].

Checking for selectional constraint violations in:
     (
          [woman:*3];
          (
               [person:*3]-
                       (gender)->[sex:female]
          )
     );
     (
          (
               [woman:*3]
          );
          [person:*3]-
                  (gender)->[sex:female]
     ).

Is sex a kind of person? n.

conformity relation added:
     sex :: &.

lattice relation added:
     sex << universal_type.

lattice relation added:
     absurd_type << sex.

Is person a kind of sex? n.
```

```
Is gender a kind of spouse? n.

conformity relation added:
     gender :: &.

lattice relation added:
     gender << universal_rel.
lattice relation added:
     absurd_rel << gender.

Is spouse a kind of gender? n.

Is the general relation:

     [person:*4]-
          (gender)->[sex:*5].

the most general unambiguous usage? y.

Checking denotation of:
     (
          [woman:*3];
          (
               [person:*3]-
                    (gender)->[sex:female]
          )
     );
     (
          (
               [woman:*3]
          );
          [person:*3]-
               (gender)->[sex:female]
     ).

I did not know that:

     (
          [woman:*3];
          (
               [person:*3]-
                    (gender)->[sex:female]
          )
     );
     (
          (
               [woman:*3]
          );
          [person:*3]-
               (gender)->[sex:female]
     ).

Tidying up...

Is wife a kind of gender? n.

Is wife a kind of spouse? y.

conformity relation added:
     wife :: &.
```

```
lattice relation added:
     wife << spouse.

lattice relation added:
     wife << universal_rel.

lattice relation added:
     absurd_rel << wife.
Enter definition for wife in terms of spouse:
[person]->(spouse)->[woman].

(
     [person:*8]-
             (wife)->[woman:*9];
     (
          [person:*8]-
                  (spouse)->[woman:*9]
     )
);
(
     (
          [person:*8]-
                  (wife)->[woman:*9]
     );
     [person:*8]-
             (spouse)->[woman:*9]
).

Is this OK? y.

Checking for selectional constraint violations in:
     (
          [person:*8]-
                  (wife)->[woman:*9];
          (
               [person:*8]-
                       (spouse)->[woman:*9]
          )
     );
     (
          (
               [person:*8]-
                       (wife)->[woman:*9]
          );
          [person:*8]-
                  (spouse)->[woman:*9]
     ).

Is the general relation:

     [person:*10]-
             (wife)->[woman:*11].

the most general unambiguous usage? y.
```

```
Checking denotation of:
     (
          [person:*8]-
                  (wife)->[woman:*9];
          (
               [person:*8]-
                       (spouse)->[woman:*9]
          )
     );
     (
          (
               [person:*8]-
                       (wife)->[woman:*9]
          );
          [person:*8]-
                  (spouse)->[woman:*9]
     ).

I did not know that:
     (
          [person:*8]-
                  (wife)->[woman:*9];
          (
               [person:*8]-
                       (spouse)->[woman:*9]
          )
     );
     (
          (
               [person:*8]-
                       (wife)->[woman:*9]
          );
          [person:*8]-
                  (spouse)->[woman:*9]
     ).

Tidying up...

Checking denotation of:
     [person:john]-
             (wife)->[woman:elizabeth].

I did not know that:
     [person:john]-
             (wife)->[woman:elizabeth].

Tidying up...

conformity relation added:
     woman :: elizabeth.
concept specialised:
     [person:elizabeth];
to:
     [woman:elizabeth]

>>
```

Notice how CGP can find out all the information that may be relevant to the proof of the original statement. The salient point in this example is the request for a definition of wife in terms of spouse. As with type definitions, only the *differentiae* are given and CGP compiles the rest automatically. It is, however, possible

for more than one form of the definition to be generated by CGP. For this reason the system asks if the form that it has generated is OK. If it is then answer *y.* as in this example. If not then answer *n.* and another form will be offered. The possible variations in the form of the definition are concerned with the placing of lines of identity and it is these that should be checked with each form offered.

4.3.4 Selectional Constraint Violations

Having taught CGP about what is allowed then an attempt to use a graph that contains a selectional constraint violation will result in the finding of that graph to be FALSE. With the examples of the previous section in the system, an attempt to inform CGP that Elizabeth has a wife called John results in:

```
>> ![woman:elizabeth]->(wife)->[person:john].

Checking for selectional constraint violations in:

     [woman:elizabeth]-
          (wife)->[person:john].

I think the opposite is true to:
     [woman:elizabeth]-
          (wife)->[person:john].

because:

1/0    relations violating selectional constraints:
          [woman:elizabeth]-
               (wife)->[person:john].

>>
```

The graph that is offered by CGP as a possible canonical graph for the *wife* relation is generated by finding a common generalisation between the known usage of the relation and the newly encountered one. This is offered as a possible replacement canonical graph for the relation. However, if the correct most general graph has already been entered then the answer to this question will always be *n*..

4.3.5 Definition Inconsistencies

In some views of the nature of type definitions t is possible that the type definitions of a type and one of its subtypes are contradictory. An example is the case where a bird is defined as a flying animal and then an ostrich is defined as a nonflying bird. In a truly Aristotelian hierarchy this does not make sense because either all birds fly or they don't. If they all fly by definition and we want an ostrich to be defined as a bird then it must also fly. The truth is that flight is not a necessary condition for any animal to be a bird and should not form part of its type definition.

Having said all this the following is a type definition for a bird:

([*bird* : *x] ([*animal* : *x]<-(*agent*)<-[*flight*]))
([*animal* : *x]<-(*agent*)<-[*flight*] ([*bird* : *x]))

and the following is that for an ostrich:

([*ostrich* : *x] (([*bird* : *x]<-(*agent*)<-[*flight*])))
(([*bird* : *x]<-(*agent*)<-[*flight*]) ([*ostrich* : *x]))

The first part reduces to:

([*ostrich* : *x] [*bird* : *x]<-(*agent*)<-[*flight*])

and together, both parts say that *x* is an ostrich that is not a flying bird and *x* is either an ostrich or it can fly. In short, anything that is an ostrich is both a flying bird and it cannot fly. Any attempt to inform CGP of the existence of an ostrich will result in the rejection of the graph on the grounds that it is contradicted by these two rules and the type definition given for a bird.

4.4 Some Simple False Facts

Before going on to the main substance of this section it is necessary to generate the required knowledge base. To do so, ensure that the prompt is of the *user:>>* form. This will simplify the interactions and make these instructions easier to follow. Finally, re_enter the two graphs:

```
[person:John]->(parent)->[person:Edward]

[person:John]->(parent)->[person:Rita].
```

The interaction with CGP is now:

```
user:>> ![person:John]->(parent)->[person:Edward].

conformity relation added:
     person :: &.

lattice relation added:
     person << universal_type.

lattice relation added:
     absurd_type << person.

Checking for selectional constraint violations in:
     [person:John]-
             (parent)->[person:Edward].

conformity relation added:
     parent :: &.

lattice relation added:
     parent << universal_rel.

lattice relation added:
     absurd_rel << parent.

Checking denotation of:
     [person:John]-
             (parent)->[person:Edward].

I did not know that:
     [person:John]-
             (parent)->[person:Edward].

Tidying up...

conformity relation added:
     person :: John.

conformity relation added:
     person :: Edward.

user:>> ![person:John]->(parent)->[person:Rita].

Checking for selectional constraint violations in:
     [person:John]-
             (parent)->[person:Rita].

Checking denotation of:
     [person:John]-
             (parent)->[person:Rita].
```

```
I did not know that:
     [person:John]-
               (parent)->[person:Rita].

Tidying up...

conformity relation added:
     person :: Rita.

user:>>
```

In this section, some simple negated contexts will be added. These are added in the same way as ordinary facts but the effect on the knowledge base is slightly different because they are not added to the semantic net. Suppose that Edward is bald and we choose to tell CGP that Edward does not have any hair. The way to state this is 'it is false that Edward has some hair'. The command to tell CGP is:

```
user:>> !([person:Edward]<-(possessor)<-[hair]).
```

CGP responds with:

```
conformity relation added:
     hair :: &.

lattice relation added:
     hair << universal_type.

lattice relation added:
     absurd_type << hair.

Checking for selectional constraint violations in:
     (
          [hair:*1]-
                    (possessor)->[person:Edward]
     ).

conformity relation added:
     possessor :: &.

lattice relation added:
     possessor << universal_rel.

lattice relation added:
     absurd_rel << possessor.

Checking denotation of:
     (
          [hair:*1]-
                    (possessor)->[person:Edward]
     ).

I did not know that:
     (
          [hair:*1]-
                    (possessor)->[person:Edward]
     ).

Tidying up...

user:>>
```

Note that the nested graph has been printed in the reverse order to that in which it was entered, which shows that CGP has parsed it correctly but has stored it in the reverse but equivalent form. Also note that

CGP was told that the *possessor* relation exists between a person and hair. This was done for simplicity to avoid a long chain of explanations about what hair is.

This form of negated context is the simplest form of false fact that can be added. There are no nested negated contexts, only a simple negated fact. It is now possible to ask CGP if Edward has any hair:

```
user:>> ?[person:Edward]<-(possessor)<-[hair].
```

The reply is:

```
Checking for selectional constraint violations in:
     [hair:*3]-
            (possessor)->[person:Edward].

Checking denotation of:
     [hair:*3]-
            (possessor)->[person:Edward].

The statement:
     [hair:*3]-
            (possessor)->[person:Edward].

is:

FALSE because:
   1/0.     proof of:
               [hair:*3]-
                      (possessor)->[person:Edward];
   2/0.     is:
   3/1.          fact:
                    (
                         [hair:*1]-
                                (possessor)->[person:Edward]
                    ).

user:>>
```

CGP has correctly proved that Edward does not have any hair. It is also possible to ask CGP if Edward has any particular hair rather than any hair at all. Such a query might be:

```
user:>> ?[person:Edward]<-(possessor)<-[hair:#53246].
```

The reply is:

```
Checking for selectional constraint violations in:
     [hair:#53246]-
            (possessor)->[person:Edward].

Checking denotation of:
     [hair:#53246]-
            (possessor)->[person:Edward].

The statement:
     [hair:#53246]-
            (possessor)->[person:Edward].

is:

FALSE because:
```

```
1/0.      proof of:
                [hair:#53246]-
                        (possessor)->[person:Edward];
   2/0.      is:
   3/1.           fact:
                     (
                          [hair:*1]-
                                  (possessor)->[person:Edward]
                     ).

user:>>
```

CGP has correctly shown that if Edward does not have any hair at all then he does not have any particular hair either.

4.5 Simple Deduction

So far, all the examples have centered around queries of knowledge that is explicit within the knowledge base. In order to extend the power of the evaluation procedure it is necessary to use logical deduction. The deductive mechanism that CGP uses is based upon *Peirce logic*, which is the logical basis for the whole of conceptual graphs, and some familiarity with this formalism would be an advantage. It is also assumed that the reader has some familiarity with standard logic, so that terms such as *conjunction*, *disjunction*, *implication*, '*modus ponens*' and '*modus tollens*' are familiar. Appendix II contains a list of the most common logical relationships and their equivalent Peirce forms.

Two examples will be considered in this section. The first will deal with the deduction of new knowledge by the use of *IF-THEN* rules and the second will relate to a previous example.

4.5.1 Simple Rules

To prepare the way for the first example of deduction it is necessary to add some information to the knowledge base. This information is a statement that Edward is male:

```
user:>> ![person:Edward]->(gender)->[sex:male].
```

CGP accepts this as a new fact in the normal way. Now, we would like to say that 'if some person *x* has a parent who is male then that parent is the father of person *x*'. To help with the formulation of such statements into the correct graph the statement should be worded in this form, with the use of words such as *some*, *exists*, *true*, *false*, *if* and *then*. When this is done the mapping into conceptual graphs becomes much easier and reliable.

This is a typical IF-THEN rule in which the IF part is *some person x has a parent who is male* and the THEN part is *that parent is the father of person x*. The general form required for this rule is that of an implication, which, symbolically, is:

(*p* (*q*))

which states that *if p then q*. In this graph, *p* is the *antecedent* and *q* is the *consequent*. Therefore the correct graphical representation of the rule of fatherhood is:

([*person* : **x*]->(*parent*)->[*person* : **y*]->(*gender*)->[sex : *male*]
([*person* : **x*]->(*father*)->[*person* : **y*])).

It is a good idea to put the consequent of this rule on a separate line and indent it slightly as above. When this rule is submitted to CGP it is evaluated and then added to the knowledge base:

```
user:>>!([person:*x]->(parent)->[person:*y]->(gender)->[sex:male]
            ([person:*x]->(father)->[person:*y])).
```

It is now possible to ask CGP if Edward is the father of John:

```
user:>> ?[person:John]->(father)->[person:Edward].
```

CGP replies with:

```
Checking for selectional constraint violations in:
     [person:John]-
            (father)->[person:Edward].

Checking denotation of:
     [person:John]-
            (father)->[person:Edward].

The statement:
     [person:John]-
            (father)->[person:Edward].

is:

TRUE because:
   1/0.     proof of:
               [person:John]-
                     (father)->[person:Edward];
   2/0.     is:
   3/1.          deduction:
   4/1.          known rule:
                    (
                         [person:*4]-
                                (gender)->[sex:male];
                         [person:*5]-
                                (parent)->[person:*4];
                         (
                              [person:*5]-
                                     (father)->[person:*4]
                         )
                    );
   5/1.          so:
                    (
                         (
                              [person:John]-
                                     (father)->[person:Edward]
                         );
                         [person:Edward]-
                                (gender)->[sex:male];
                         [person:John]-
                                (parent)->[person:Edward]
                    );
   6/1.          proving antecedent:
                    [person:Edward]-
                           (gender)->[sex:male];
   7/1.          proof of:
                    [person:Edward]-
                           (gender)->[sex:male];
   8/1.          is:
   9/2.               fact:
                         [person:Edward]-
                                (gender)->[sex:male];
  10/1.          proving antecedent:
                    [person:John]-
                           (parent)->[person:Edward];
  11/1.          proof of:
                    [person:John]-
                           (parent)->[person:Edward];
  12/1.          is:
```

```
13/2.                fact:
                        [person:John]-
                                (parent)->[person:Edward];
14/1.          result:
                  [person:John]-
                          (father)->[person:Edward];
15/0.     to give:
             {}.

user:>>
```

CGP has derived the query from a fact and the rule. This is an example of *modus ponens*. CGP achieved the deduction by following a simple procedure: the necessary rule was located, updated with referents from the query and then the nested graphs were individually proved; any true ones were removed by *deiteration* and the residue, after removal of double negations, was taken to be the deduction.

4.5.2 A More Subtle Example

The second example of deduction concerns the earlier query about whether John is Edward's parent. It will be recalled that CGP responded with UNKNOWN when asked this query. We are now in a position to tell CGP that if some person *x* is a parent of some person *y* then it is false that person *y* is a parent of person *x*.

The general form required for the rule about parentage is that of NAND (NOT AND). This states that one or other of the propositions may be true but that at least one is definitely false. The general form is therefore:

(*p q*)

which, when applied to the rule about parentage becomes:

([*person* : **x*]->(*parent*)->[*person* : **y*]
  [*person* : **y*]->(*parent*)->[*person* : **x*])

This fact should now be submitted to CGP:

```
user:>> !([person:*x]->(parent)->[person:*y]
             [person:*y]->(parent)->[person:*x]).
```

As usual, CGP evaluates the rule to see if it is contradicted by its knowledge. In this case the rule is acceptable and is added to the knowledge base in the usual way. It is now possible to ask the query:

```
user:>>?[person:Edward]->(parent)->[person:John].
```

This time, the response is quite different to the previous time when UNKNOWN was generated. CGP produces something similar to:

```
Checking for selectional constraint violations in:
     [person:Edward]-
             (parent)->[person:John].

Checking denotation of:
     [person:Edward]-
             (parent)->[person:John].

The statement:
     [person:Edward]-
             (parent)->[person:John].

is:

FALSE because:
```

```
1/0.      proof of:
             [person:Edward]-
                    (parent)->[person:John];
2/0.      is:
3/1.           deduction:
4/1.           known rule:
                  (
                       [person:*6]-
                              (parent)->[person:*7];
                       [person:*7]-
                              (parent)->[person:*6]
                  );
5/1.           so:
                  (
                       [person:John]-
                              (parent)->[person:Edward];
                       [person:Edward]-
                              (parent)->[person:John]
                  );
 6/1.          proving antecedent:
                  [person:John]-
                         (parent)->[person:Edward];
 7/1.          proof of:
                  [person:John]-
                         (parent)->[person:Edward];
 8/1.          is:
 9/2.            fact:
                    [person:John]-
                           (parent)->[person:Edward];
10/1.          result:
                  (
                       [person:Edward]-
                              (parent)->[person:John]
                  );
   11/0.     to give:
                  ().

user:>>
```

Without going into all the details of the deduction algorithm, the proof of the query was accomplished by finding an appropriate rule and making it applicable to the query by projecting one of the nested graphs onto the query. This allowed the instantiation of variables and specialisation of types. Having done that, the nested graph that could be proved directly (the one stating that Edward is John's parent) could be removed by the Peirce rule of deiteration and the result found.

This deductive mechanism will be seen to be very similar to that for the example given in 4.5.1. This leads to an important principle of CGP. That is that CGP does not distinguish between antecedents and consequents of rules. Any nested graph can take on the role of antecedent but the *consequence* of a rule need not be the classical consequent of a classical implication. This will be illustrated again later.

4.6 More Complex Deduction

So far, we have considered deduction that requires the use of a single rule and a single fact. Other forms also exist: deduction requiring several facts, deduction requiring a chain of rules, deduction requiring recursive use of the same rule and *modus tollens*. There is also a further example in which a deduction can only proceed when a rule that is required for a deduction must itself be deduced. To illustrate these cases, simplified knowledge bases will be used. These will be of a more symbolic nature than the previous ones because the use of such knowledge bases will save typing and demonstrate that the principles apply to arbitrary objects. However, real world examples will also be given in some cases. This approach will also help the novice in logic to see how Peirce logic relates to ordinary symbolic logic and to statements made in natural language.

Before trying the examples in each of the following sections it is advantageous to use the - command to clear the knowledge base. The following examples also assume that the command *%user* has been issued.

4.6.1 Deduction Requiring Several Facts

An example of a deduction that requires the proof of more than one graph is:

p q (p q(r)).

which could be interpreted as stating that *p* is true, *q* is true and the truth of both *p* and *q* implies the truth of *r*. So that CGP will not find a trivial solution based on *type contraction*, the graph *r* will have a relation attached to it. In conceptual graphs this could be formulated as:

```
user:>> ![p:1] [q:1] ([p:*x][q:*x]([r:*x]->(rel)->[s])).
```

It is now possible to make the query:

```
user:>> ?[r].
```

CGP can perform the necessary deduction by exactly the same procedure as before. A rule appropriate to the query is found and is updated to contain the referents contained in the query. All the true graphs are removed from the rule and the residue is, after removal of double negations, taken to be the deduction. In the case of the current example, CGP responds with something like:

```
Checking for selectional constraint violations in:
     [r:*4].

Checking denotation of:
     [r:*4].

The statement:
     [r:*4].

is:

TRUE because:
   1/0.      proof of:
                [r:*4];
   2/0.      is:
   3/1.           deduction:
   4/1.           known rule:
                     (
                          [p:*1];
                          [q:*1];
                          (
                               [r:*1]-
                                       (rel)->[s:*2]
                          )
                     );
   5/1.           so:
                     (
                          (
                               [r:*4]-
                                       (rel)->[s:*2]
                          );
                          [p:*4];
                          [q:*4]
                     );
6/1.            proving antecedent:
                     [p:*4];
```

```
 7/1.          proof of:
                  [p:*4];
 8/1.          is:
 9/2.               fact:
                       [q_p:1];
10/1.          proving antecedent:
                  [q_p:1];
11/1.          proof of:
                  [q_p:1];
12/1.          is:
13/2.               fact:
                       [q_p:1];
14/1.          result:
                  [r:1];
15/0.     to give:
             {}.

          user:>>
```

Note that this example demonstrates some of the housekeeping that CGP performs. If it is known that some entity *x* is of types *p* and *q* then it follows that there a a common subtype of *p* and *q* to which *x* conforms. That subtype is generated automatically by CGP and is used wherever entity *x* is referred to.

A (crudely represented) real world example might be the rule 'if the sun is shining and some person is on the beach then that person will become sunburnt':

([*person* : **x*]->(*location*)->[*beach*]
  [*sun* : *sol*]<-(*agent*)<-[*shine*]
        ([*person* : **x*]<-(*recipient*)<-[*sunburn*])).

This will not be shown here and may by tried as an exercise.

4.6.2 Deduction Requiring A Chain Of Rules

Sometimes the antecedent of an implication may only be provable by further deduction. In this case it is necessary for CGP to be able to use the full deductive mechanism on each antecedent. The symbolic example is:

p (*p*(*q*)) (*q*(*r*))

which can be coded into conceptual graphs (after using the - command) as:

```
user:>> ![p:1] ([p:*x]([q:*x])) ([q:*x]([r:*x])).
```

Now, it is possible to ask CGP if there exists an entity of type *r*:

```
user:>> ?[r].
```

CGP responds with:

```
Checking for selectional constraint violations in:
     [r:*2].

Checking denotation of:
     [r:*2].

The statement:
     [r:*2].

is:

TRUE because:
```

```
1/0.      proof of:
             [r:*2];
2/0.      is:
3/1.           deduction:
4/1.           known rule:
                  (
                       [q:*0];
                       (
                            [r:*0]
                       )
                  );
5/1.           so:
                  (
                       (
                            [r:*2]
                       );
                       [q:*2]
                  );
 6/1.          proving antecedent:
                  [q:*2];
 7/1.          proof of:
                  [q:*2];
 8/1.          is:
 9/2.               deduction:
10/2.               known rule:
                       (
                            [p:*0];
                            (
                                 [q:*0]
                            )
                       );
11/2.               so:
                       (
                            (
                                 [q:*2]
                            );
                            [p:*2]
                       );
12/2.               proving antecedent:
                       [p:*2];
13/2.               proof of:
                       [p:*2];
14/2.               is:
15/3.                    fact:
                            [p:1];
16/2.               result:
                       [q:1];
17/1.          to give:
                  {};
18/1.          result:
                  [r:1];
19/0.     to give:
             {}.

user:>>
```

Note that the explanation for the proof of [*q* : **3*] is nested within that for the original query, as shown by the nested *proof of antecedent ... result* messages, the numbering and the indentation. This means that complex explanations may require careful reading in order to follow them properly, but it also demonstrates the naturalness of the deductive mechanism. In practice, complete nested explanations are bracketed either by:

```
.
is:
.
.
fact:

is:
.
.
result:

is:
.
.
axiom:

is:
.
.
to give:
```

or

```
is:
.
.
to give:
```

and can be deciphered quite easily, especially when the numbering of each stage is taken into account.

A real world example might be *if a person is born in Britain then that person is British, and if a person is British then that person is a connoisseur of fish and chips.* A (crude) conversion into conceptual graphs gives:

```
user:>> ![person:*x]->(birthplace)->[place:Britain]
            ([person:*x]->(nationality)->[British])).

user:>> ![person:*x]->(nationality)->[British]
            ([person:*x]<-(connoisseur)<-[fish_and_chips])).

user:>> ![person:John]->(birthplace)->[place:Britain].
```

The query 'is John a connoisseur of fish and chips' is:

```
user:>> ?[person:John]<-(connoisseur)<-[fish_and_chips].
```

produces the following output:

```
Checking for selectional constraint violations in:
     [fish_and_chips:*9]-
          (connoisseur)->[person:John].

Checking denotation of:
     [fish_and_chips:*9]-
          (connoisseur)->[person:John].

The statement:
     [fish_and_chips:*9]-
          (connoisseur)->[person:John].

is:

TRUE because:
```

```
1/0.    proof of:
           [fish_and_chips:*9]-
                  (connoisseur)->[person:John];
2/0.    is:
3/1.         deduction:
4/1.         known rule:
                (
                     [person:*5]-
                            (nationality)->[British:*6];
                     (
                          [fish_and_chips:*7]-
                                 (connoisseur)->[person:*5]
                     )
                );
5/1.         so:
                (
                     (
                          [fish_and_chips:*9]-
                                 (connoisseur)->[person:John]
                     );
                     [person:John]-
                            (nationality)->[British:*6]
                );
6/1.         proving antecedent:
                [person:John]-
                       (nationality)->[British:*6];
7/1.         proof of:
                [person:John]-
                       (nationality)->[British:*6];
8/1.         is:
9/2.              deduction:
10/2.             known rule:
                     (
                          [person:*1]-
                                 (birthplace)->[place:Britain];
                          (
                               [person:*1]-
                                      (nationality)->[British:*2]
                          )
                     );
11/2.             so:
                     (
                          (
                               [person:John]-
                                      (nationality)->[British:*6]
                          );
                          [person:John]-
                                 (birthplace)->[place:Britain]
                     );
12/2.             proving antecedent:
                     [person:John]-
                            (birthplace)->[place:Britain];
13/2.             proof of:
                     [person:John]-
                            (birthplace)->[place:Britain];
14/2.             is:
15/3.                  fact:
                          [person:John]-
                                 (birthplace)->[place:Britain];
16/2.             result:
                     [person:John]-
                            (nationality)->[British:*6];
```

```
17/1.          to give:
                  {};
18/1.          result:
                  [fish_and_chips:*9]-
                        (connoisseur)->[person:John];
19/0.     to give:
             {}.

user:>>
```

This section has described the chaining of rules but has not mentioned the terms *forward chaining* or *backward chaining*. This is deliberate because CGP does not distinguish between the two. More will be said about this in the section on *modus tollens*.

4.6.3 Deduction With Recursive Use Of A Rule

There are cases in which a proof may require repeated use of the same rule. This is the case with *transitive* relations and *transitive closure* is obtained by applying the same rule recursively. An example of transitive closure is:

([*p* : **x*]->(*r*)->[*p*]->(*r*)->[*p* : **y*]
([*p* : **x*]->(*r*)->[*p* : **y*])).

This states that the relation *r* which exists between two entities *p*:**x* and *p*:**y* with a third *p* in between implies that the relation *r* exists directly between *p*:**x* and *p*:**y*. It is appropriate to give a real world example at this point: *if a person x has an ancestor which is person y who has an ancestor which is person z then person x has an ancestor which is person z*. This real world example will be used in this section instead of the symbolic form. To demonstrate this facility, enter the - command and then the following information:

```
user:>>![person:John]-
          (ancestor)->[person:Edward]-
                (ancestor)->[person:Hugh]-
                      (ancestor)->[person:Lambert]-
                            (ancestor)->[person:Waldef],,,.

user:>>
!([person:*x]->(ancestor)->[person]->(ancestor)->[person:*y]
          ([person:*x]->(ancestor)->[person:*y])).
```

Now, the question can be posed 'is Waldef an ancestor of John?'. The proof of this requires three stages, each using the same rule. It also shows another powerful feature of CGP which is necessary for reasoning of this type. The output will now be given and then discussed in more detail to illustrate this feature.

```
user:>> ?[person:John]->(ancestor)->[person:Waldef].

Checking for selectional constraint violations in:
     [person:John]-
          (ancestor)->[person:Waldef].

Checking denotation of:
     [person:John]-
          (ancestor)->[person:Waldef].

The statement:
     [person:John]-
          (ancestor)->[person:Waldef].

is:

TRUE because:
   1/0.     proof of:
               [person:John]-
                     (ancestor)->[person:Waldef];
```

```
2/0.     is:
3/1.          deduction:
4/1.          known rule:
                 (
                      [person:*1]-
                           (ancestor)->[person:*2];
                      [person:*3]-
                           (ancestor)->[person:*1];
                      (
                           [person:*3]-
                                (ancestor)->[person:*2]
                      )
                 );
5/1.          so:
                 (
                      (
                           [person:John]-
                                (ancestor)->[person:Waldef]
                      );
                      [person:*1]-
                           (ancestor)->[person:Waldef];
                      [person:John]-
                           (ancestor)->[person:*1]
                 );
6/1.          proving antecedent:
                 [person:*1]-
                      (ancestor)->[person:Waldef];
7/1.          proof of:
                 [person:*1]-
                      (ancestor)->[person:Waldef];
8/1.          is:
9/2.               fact:
                      [person:Lambert]-
                           (ancestor)->[person:Waldef];
10/1.         proving antecedent:
                 [person:John]-
                      (ancestor)->[person:Lambert];
11/1.         proof of:
                 [person:John]-
                      (ancestor)->[person:Lambert];
12/1.         is:
13/2.              deduction:
14/2.              known rule:
                      (
                           [person:*1]-
                                (ancestor)->[person:*2];
                           [person:*3]-
                                (ancestor)->[person:*1];
                           (
                                [person:*3]-
                                     (ancestor)->[person:*2]
                           )
                      );
```

```
15/2.                so:
                        (
                              (
                                    [person:John]-
                                          (ancestor)->[person:Lambert]
                              );
                              [person:*1]-
                                    (ancestor)->[person:Lambert];
                              [person:John]-
                                    (ancestor)->[person:*1]
                        );
16/2.                proving antecedent:
                        [person:*1]-
                              (ancestor)->[person:Lambert];
17/2.                proof of:
                        [person:*1]-
                              (ancestor)->[person:Lambert];
18/2.                is:
19/3.                      fact:
                              [person:Hugh]-
                                    (ancestor)->[person:Lambert];
20/2.                proving antecedent:
                        [person:John]-
                              (ancestor)->[person:Hugh];
21/2.                proof of:
                        [person:John]-
                              (ancestor)->[person:Hugh];
22/2.                is:
23/3.                      deduction:
24/3.                      known rule:
                              (
                                    [person:*1]-
                                          (ancestor)->[person:*2];
                                    [person:*3]-
                                          (ancestor)->[person:*1];
                                    (
                                          [person:*3]-
                                                (ancestor)->[person:*2]
                                    )
                              );
25/3.                so:
                              (
                                    (
                                          [person:John]-
                                                (ancestor)->[person:Hugh]
                                    );
                                    [person:*1]-
                                          (ancestor)->[person:Hugh];
                                    [person:John]-
                                          (ancestor)->[person:*1]
                              );
26/3.                proving antecedent:
                        [person:*1]-
                              (ancestor)->[person:Hugh];
27/3.                proof of:
                        [person:*1]-
                              (ancestor)->[person:Hugh];
28/3.                is:
29/4.                      fact:
                              [person:Edward]-
                                    (ancestor)->[person:Hugh];
```

```
30/3.                    proving antecedent:
                            [person:John]-
                                    (ancestor)->[person:Edward];
31/3.                    proof of:
                            [person:John]-
                                    (ancestor)->[person:Edward];
32/3.                    is:
33/4.                         fact:
                                 [person:John]-
                                         (ancestor)->[person:Edward];
34/3.                    result:
                            [person:John]-
                                    (ancestor)->[person:Hugh];
35/2.                 to give:
                         {};
36/2.                    result:
                            [person:John]-
                                    (ancestor)->[person:Lambert];
37/1.              to give:
                      {};
38/1.              result:
                      [person:John]-
                              (ancestor)->[person:Waldef];
39/0.           to give:
                   {}.

      user:>>
```

This example shows that it is possible for a particular rule to be used several times within the same proof, but it also shows another very important point, the powerful feature referred to earlier. When the rule is extracted the first time and updated to reflect the original query, the antecedent is:

```
[person:*1]-
(ancestor)<-[person:John]
(ancestor)->[person:Waldef];
```

This graph cannot be proved directly and so the rule is required again. But, the rule will only generate new graphs that are made up of a single ancestor relation and not two as this graph requires. The only way to prove this antecedent therefore is to split it into two separate graphs, one for each relation, and prove these two as separate antecedents. In this way, as the explanations show, CGP was able to prove one of the relations directly:

```
6/1.           proving antecedent:
                  [person:*1]-
                          (ancestor)->[person:Waldef];
7/1.           proof of:
                  [person:*1]-
                          (ancestor)->[person:Waldef];
8/1.           is:
9/2.                fact:
                       [person:Lambert]-
                               (ancestor)->[person:Waldef];
```

This finds a referent for the middle person of the unsplit graph and generates the graph:

```
[person:John]-
        (ancestor)->[person:Lambert];
```

which is then proved by the second incarnation of the rule. This graph must also be proved in separate stages because there is no direct link between the two concepts and so the rule will be invoked for a third time at some point during the proof.

This example is worth careful study because it illustrates the nesting of the explanations that CGP gives.

4.6.4 *Modus Tollens*

Modus tollens is the inverse of *modus ponens*. It states that *if p implies q and q is false then p is false*. Stated symbolically:

$$((p \Rightarrow q) \wedge \neg q) \Rightarrow \neg p)$$

and in conceptual graphs:

```
(
        ([p:*x]([q:*y]))
        ([q:*y])
        (([p:*x]))
).
```

An example will illustrate the point. We wish to state that *if x is a mammal then x is a warm-blooded animal, but it is false that the crocodile Spike is a warm blooded animal.* After using the - command, this is represented as:

```
user:>> !([mammal:*x]
          ([animal:*x]->(attr)->[warm_blooded])).

user:>> !([crocodile:Spike]->(attr)->[warm_blooded]).

user:>> ![crocodile:Spike].
```

One extra piece of information is required, because the *%user* command has disabled the part of the system which would have dealt with it:

```
user:>> crocodile << animal.
```

It is now possible to ask CGP if Spike is a mammal:

```
user:>> ?[mammal:Spike].

Checking for selectional constraint violations in:
     [mammal:Spike].

Checking denotation of:
     [mammal:Spike].

The statement:
     [mammal:Spike].

is:

FALSE because:
   1/0.     proof of:
                [mammal:Spike];
   2/0.     is:
   3/1.          deduction:
   4/1.          known rule:
                    (
                         [mammal:*1];
                         (
                              [animal:*1]-
                                   (attr)->[warm_blooded:*2]
                         )
                    );
```

```
5/1.          so:
                   (
                        (
                             [crocodile:Spike]-
                                  (attr)->[warm_blooded:*2]
                        );
                        [mammal:Spike]
                   );
   6/1.       proving antecedent:
                   (
                        [crocodile:Spike]-
                             (attr)->[warm_blooded:*2]
                   );
   7/1.       proof of:
                   (
                        [crocodile:Spike]-
                             (attr)->[warm_blooded:*2]
                   );
    8/1.       is:
    9/2.            fact:
                        (
                             [crocodile:Spike]-
                                  (attr)->[warm_blooded:*4]
                        );
   10/1.       result:
                        (
                        [mammal:Spike]
                        );
   11/0.    to give:
                   ().

user:>>
```

Notice that the basic mechanism for *modus tollens* is identical to that for all the other examples given earlier. CGP therefore does not treat either *modus ponens* and *modus tollens* as special cases, each with their own code. The deiteration of true graphs from a rule automatically generates these two rules of inference.

A second example of *modus tollens* is appropriate at this point. In the previous example, the solution was found by simply proving and deiterating the inner negated context of the rule. Suppose that the knowledge base contains the family relations graphs in section 4.6.3 along with the extra graph which denies that any person can be an ancestor of themselves:

([person : *x]->(ancestor)->[person : *x])

Now, the query:

[person : Edward]->(ancestor)->[person : John]

is proved as follows:

```
user:>> ?[person:Edward]->(ancestor)->[person:John].

Checking for selectional constraint violations in:
     [person:Edward]-
          (ancestor)->[person:John].

Checking denotation of:
     [person:Edward]-
          (ancestor)->[person:John].
```

```
The statement:
     [person:Edward]-
            (ancestor)->[person:John].

is:

FALSE because:
   1/0.     proof of:
               [person:Edward]-
                     (ancestor)->[person:John];
   2/0.     is:
   3/1.          deduction:
   4/1.          known rule:
                    (
                         [person:*1]-
                               (ancestor)->[person:*2];
                    [person:*3]-
                               (ancestor)->[person:*1];
                    (
                         [person:*3]-
                                   (ancestor)->[person:*2]
                    )
                    );
   5/1.          so:
                    (
                    [person:John]-
                               (ancestor)->[person:*2];
                    [person:Edward]-
                               (ancestor)->[person:John];
                    (
                         [person:Edward]-
                                   (ancestor)->[person:*2]
                    )
                    );
   6/1.          proving antecedent:
                    [person:John]-
                               (ancestor)->[person:*2];
   7/1.          proof of:
                    [person:John]-
                               (ancestor)->[person:*2];
   8/1.          is:
   9/2.               fact:
                         [person:John]-
                                    (ancestor)->[person:Edward];
  10/1.          proving antecedent:
                    (
                         [person:Edward]-
                                   (ancestor)->[person:Edward]
                    );
  11/1.          proof of:
                    (
                         [person:Edward]-
                                   (ancestor)->[person:Edward]
                    );
  12/1.          is:
  13/2.               fact:
                         (
                              [person:*4]-
                                        (ancestor)->[person:*4]
                         );
```

```
14/1.          result:
                    (
                         [person:Edward]-
                              (ancestor)->[person:John]
                    );
  15/0.     to give:
               ().

user:>>
```

This example is interesting because the proof can only succeed when not only has the nested negated context been proved and deiterated but so has part of the 'antecedent'. This demonstrates the need to prove each part of a graph as a separate entity because without this feature CGP would be unable to prove this example.

4.6.5 Deductions Requiring The Use Of Conditional Rules

Consider the following statement: *If it is raining then the pavement will get wet and if I go out without wearing my waterproof shoes then my feet will get wet.* This is a complex rule which contains as one of its consequences the nested rule: *If I go out without wearing my waterproof shoes then my feet will get wet.* If this rule is known and it is also known that it is raining and that I have not got my waterproof shoes on then I am going to get wet feet.

CGP will evaluate any nested rules of this form if they are appropriate to a query. Put symbolically, the general form of the rule above, in conceptual graphs, is:

([*p*]([*q*]([*r*]([*s*])))).

Following the use of the - command and being given this knowledge and the knowledge about the weather and the shoes:

```
user:>> !([p:*x]([q:*x]([r:*x]([s:*x])))).

user:>> ![p:1] [r:1].
```

CGP can now prove that an instance of [*s*] exists:

```
user:>> ?[s].
Checking for selectional constraint violations in:
     [s:*2].

Checking denotation of:
     [s:*2].

The statement:
     [s:*2].

is:

TRUE because:
   1/0.     proof of:
                [s:*2];
   2/0.     is:
   3/1.          deduction:
```

```
4/1.          known rule:
                    (
                         [p:*0];
                         (
                              [q:*0];
                              (
                                   [r:*0];
                                   (
                                        [s:*0]
                                   )
                              )
                         )
                    );
   5/1.          so:
                    (
                         (
                              (
                                   (
                                        [s:*2]
                                   );
                                   [r:*2]
                              );
                              [q:*2]
                         );
                         [p:*2]
                    );
   6/1.          proving antecedent:
                    [p:*2];
   7/1.          proof of:
                    [p:*2];
   8/1.          is:
   9/2.               fact:
                         [r_p:1];
   10/1.         proving antecedent:
                    [q:1];
   11/1.         proof of:
                    [q:1];
   12/1.         is:
   13/2.              deduction:
   14/2.              known rule:
                         (
                              [p:*0];
                              (
                                   [q:*0];
                                   (
                                        [r:*0];
                                        (
                                             [s:*0]
                                        )
                                   )
                              )
                         );
   15/2.         so:
                    (
                         (
                              [q:1]
                         );
                         [r_p:1]
                    );
   16/2.         proving antecedent:
                    [r_p:1];
```

```
17/2.              proof of:
                        [r_p:1];
  18/2.               is:
  19/3.                    fact:
                              [r_p:1];
  20/2.               result:
                         [q:1];
  21/1.          to give:
                    {};
  22/1.          proving antecedent:
                    [r_p:1];
  23/1.          proof of:
                    [r_p:1];
  24/1.          is:
  25/2.               fact:
                         [r_p:1];
  26/1.               result:
                         [s:1];
  27/0.          to give:
                    {}.

user:>>
```

Note that, once again, the subtype of *p* and *r*, *p_r*, has been created.

4.6.6 Deductions Requiring Combinations Of Rules

There are occasions when the answer to a query is contained within the the rules even if no specific knowledge is present. To give a simple example, consider the following two rules: *If Clyde is a cat then Clyde is an elephant* and *It is false that Clyde is a cat and Clyde is an elephant.* Without knowing anything about Clyde it is possible to work out that the second rule states that Clyde is not both a cat and an elephant (although he may be one or the other). Given this, then we know that, from the first rule, if Clyde is a cat then he must also also an elephant. Therefore, because of the second rule, the first rule can only be true if Clyde is not a cat. We cannot say, however, whether Clyde is actually an elephant or not.

In other words, we can prove that Clyde is not a cat just from these two rules. Symbolically, this can be shown and proved as follows (see section 4.7 for descriptions of the rules of inference which CGP uses):

given the rules:

$(p\,(q))$

$(p\,q)$

iteration of the second rule into the nested context of the first rule:

$(p\,(q\,(p\,q)))$

deiteration of *p*

$(p\,(q\,(q)))$

deiteration of *q*

$(p\,(q\,()))$

elimination of disjunctions:

$(p\,(()))$

removal of double negations:

(p)

In other words, it is possible to derive the graph that proves that p is false. There are other ways of achieving the same result. The above example can be tried in conceptual graphs, and seeing a variation of this proof, by telling CGP the two rules:

```
user:>> !([p:*x][q:*x]).

user:>> !([p:*x]([q:*x])).
```

followed by the query:

```
user:>> ?[p].

Checking for selectional constraint violations in:
     [p:*3].

Checking denotation of:
     [p:*3].

The statement:
-    [p:*3].

is:

FALSE because:
   1/0.     proof of:
               [p:*3];
   2/0.     is:
   3/1.          iteration of:
                    (
                         [p:*0];
                         [q:*0]
                    );
   4/1.          giving:
                    (
                         (
                              (
                                   [p:*3];
                                   [q:*3]
                              );
                              [p:*3]
                         )
                    );
   5/1.          deiteration of:
                    [p:*3];
   6/1.          giving:
                    (
                         (
                              (
                                   [q:*3]
                              );
                              [p:*3]
                         )
                    );
   7/1.          removal of double negations:
   8/1.          giving:
                    (
                         [q:*3]
                    );
                    [p:*3];
```

```
9/1.        iteration of:
                (
                    [p:*1];
                    (
                        [q:*1]
                    )
                );
10/1.       giving:
                (
                    (
                        (
                            [p:*3];
                            (
                                [q:*3]
                            )
                        );
                        [p:*3];
                        (
                            [q:*3]
                        )
                    )
                );
11/1.       deiteration of:
                [p:*3];
12/1.       giving:
                (
                    (
                        (
                            (
                                [q:*3]
                            )
                        );
                        [p:*3];
                        (
                            [q:*3]
                        )
                    )
                );
13/1.       deiteration of:
                (
                    [q:*3]
                );
14/1.       giving:
                (
                    (
                        (
                        );
                        [p:*3];
                        (
                            [q:*3]
                        )
                    )
                );
15/1.       removal of disjunctions:
16/1.       giving:
                (
                    (
                        (
                        )
                    )
                );
```

```
        17/1.           removal of double negations:
        18/1.           giving:
                           (
                           );
        19/0.     to give:
                     ().

    user:>>
```

4.6.7 Higher Order Reasoning

This section describes the facilities that CGP offers to allow higher order reasoning. Higher order reasoning is reasoning at a higher level than that of simple entities and relations; rather than being reasoning *with* types and relations it is reasoning *about* types and relations as if they were themselves individual entities. CGP provides a simple facility to allow such reasoning in a natural way. The means to do this were suggested in sections 2.1 and 2.3. Before giving some examples, it will be instructive to discuss the two kinds of hierarchy that exist.

The first of these hierarchies is the familiar *type hierarchy*. In a type hierarchy, the objects are arranged in a *partial order* that reflects the relationships between them in terms of type/subtype relations. Thus, a person is a type of mammal and a mammal is a type of animal. The important point to notice is that the type/subtype relation is transitive: if a person is a type of mammal and a mammal is a type of animal then a person is a type of animal. This applies to all types, and also to relations: if an uncle by marriage is an uncle and an uncle is a relative then an uncle by marriage is a relative.

The second kind of hierarchy is a *membership hierarchy*. in a membership hierarchy some entities are members of a type and that type is also an entity which is a member of some type. Thus, Clyde is a member of the type *elephant* and elephant is a member of the type *species*. This example illustrates the essence of one of Aristotel's fallacies: the membership relation is not transitive, therefore it is not correct to say that if Clyde is an elephant and elephant is a species then Clyde is a species. The so-called *isa* relation exists in two, often confused and completely different incarnations.

The representation of type hierarchy relations has been given earlier. Such relations are given either in the form:

type1 << type2.

or in the form of type definitions. The representation of membership hierarchy relations has also been given. Such relations are given either in the form:

type :: referent.

or in the form of single concepts. The concept [*elephant* : *Clyde*] states that Clyde exists and is a member of the type *elephant*. It therefore follows that *elephant* Is a member of the type *species*. Thus:

species :: elephant.

and:

[*species* : *elephant*].

The type *species* is said to be a higher order type than is the type *elephant*. The important point to observe here is that the objects that occur in type fields can also have a meaningful existence in the referent field. Sowa's notation does not allow this. Instead he states that:

[*elephant* : *Clyde*] [*species* : *'Elephant'*].

where *elephant* and *'Elephant'* represent the same entity, but that in some sense they are different. CGP recognises that it is important to be able to refer to types as individuals with the same status and so the two concepts above would be given as:

[*elephant* : *Clyde*] [*species* : *elephant*].

Indeed, *elephant* and *'Elephant'* would be recognised by CGP as distinctly different objects. CGP will accept any valid CGP object in both the type field and the referent field. Therefore, variables can be used in the type field and can be made coreferent to some other referent field. This allows:

```
user:>> -.

Resetting...

All data retracted,
Operational modes reset.

user:>> ![species:elephant]  [elephant:clyde].

conformity relation added:
     species :: &.

lattice relation added:
     species << universal_type.

lattice relation added:
     absurd_type << species.

conformity relation added:
     elephant :: &.

lattice relation added:
     elephant << universal_type.

lattice relation added:
     absurd_type << elephant.

Checking for selectional constraint violations in:
     [species:elephant];
     [elephant:clyde].

Checking denotation of:
     [species:elephant];
     [elephant:clyde].

I did not know that:
     [species:elephant];
     [elephant:clyde].

Tidying up...

conformity relation added:
     species :: elephant.

conformity relation added:
     elephant :: clyde.

user:>> ?[species:*x]  [*x:clyde].

Checking for selectional constraint violations in:
     [species:*0];
     [*0:clyde].

Checking denotation of:
     [species:*0];
     [*0:clyde].
```

```
The statement:
     [species:*0];
     [*0:clyde].

is:

TRUE because:
   1/0.     proof of:
               [species:*0];
               [*0:clyde];
   2/0.     is:
   3/1.          proving nested structure:
                    [species:*0];
   4/1.          proof of:
                    [species:*0];
   5/1.          is:
    6/2.              fact:
                         [species:elephant];
    7/1.          giving:
                     [elephant:clyde];
    8/1.          proving nested structure:
                     [elephant:clyde];
    9/1.          proof of:
                     [elephant:clyde];
   10/1.          is:
   11/2.               fact:
                          [elephant:clyde];
   12/1.          giving:
                     {};
   13/0.     to give:
                {}.

user:>>
```

This query asks for a species of which Clyde is a member. Without variables that can be placed in both the type field and the referent field, this query is impossible because of the intransitivity of the membership relation.

The same arguments can be made about relations. A relation can be linked by a coreference link to either a type field or a referent field and this allows relations to be recast as types. Types fall into two categories: *natural* types and *role* types. Natural types are those inalienable types to which entities belong. Thus Clyde is an elephant by nature and nothing that can be done to Clyde that will remove his 'elephantness'. However, Clyde could play certain roles in life:

[*circus* : *Barnums*]->(*performer*)->[*elephant* : *Clyde*].

which states that Clyde is a performer in Barnum's circus. This role is captured by the *performer* relation in the graph but by the words 'is a performer' in the sentence. The form of this sentence is identical to that of the sentence 'Clyde is an elephant' and so, in the same way that Clyde is an elephant, it is not unreasonable to state:

[*performer* : *Clyde*].

In other words, the *performer* relation is acting as a role type for Clyde. In fact, as long as the standard reading for relations is applied, any relation can be used as a role type for the individual to which the relation points. Using this notion it is possible to define a whole class of types with a single definition:

([*universal_type*]->(**rel*)->[*universal_type* : **x*]
([**rel* : **x*])).

This rule states that if there exists some relation which points to some concept then the individual represented by that concept is playing the role described by the relation. If CGP knows that:

[*person* : *John*]->(*father*)->[*person* : *Edward*]

then it is possible to deduce:

[*father* : *Edward*].

In the same way that types can be members of higher order types, relations can be members of higher order relations. Thus, the *sibling* relation is a member of the type *symmetrical_relation*:

[*symmetrical_relation : Sibling*].

This allows the following rule to be specified:

([*universal_type* : **x*]->(**rel*)->[*universal_type* : **y*]
[*symmetrical_relation* : **rel*]
([*universal_type* : **y*]->(**rel*)->[*universal_type* : **x*])).

With this rule it is only necessary to specify the following for each symmetrical relationship:

[*symmetrical_relation : sibling*].

When a fact such as:

[*person* : *John*]->(*sibling*)->[*person* : *Stuart*]

is known to CGP, it can work out from the rule and fact that:

[*person* : *John*]<-(*sibling*)<-[*person* : *Stuart*].

The use of this general rule for each symmetrical relation removes both the need to have a separate rule of symmetry for each symmetrical relation and to state the relation twice..

The *parent* relation is a member of the type *asymmetrical_relation*:

[*asymmetrical_relation* : *parent*].

Without repeating the reasoning as given for the previous cases, the rule for defining the properties of an asymmetrical relation is:

([*universal_type* : **x*]->(**rel*)->[*universal_type* : **y*]
[*asymmetrical_relation* : **rel*]
[*universal_type* : **y*]->(**rel*)->[*universal_type* : **x*]).

The simple fact:

[*asymmetrical_relation* : *parent*]

is now sufficient for CGP to conclude that given:

[*person* : *John*]->(*parent*)->[*person* : *Edward*]

then:

([*person* : *Edward*]->(*parent*)->[*person* : *John*]).

4.7 Theorem Proving

CGP contains an integral *theorem prover*. A theorem is any conceptual graph, a *formula*, that can be reduced to the empty context {} by Peirce's rules of inference. The theorem prover can take arbitrary contexts and attempt to reduce them to one of:

*	{}	empty context	the formula is TRUE (is a theorem),
*	()	empty clause	the formula is FALSE,
*	some other residue		the formula is UNKNOWN.

In order to fully understand the theorem prover it is necessary to understand Peirce logic.

The principle behind the theorem prover is that of trying to reduce the formula to {}, which means TRUE, or (), which means FALSE. That is, to remove as much of the formula as possible by proof and deiteration. The only *heuristic* that CGP uses is the one that requires for most of its operation that the theorem should shrink at each step. There are occasions, however, where it is necessary to use the rule of *iteration*. Such occasions arise when a proof requires a *tautology* to be created and a negated context must be iterated.

The main rules used, of which there are four basic forms, will now be described with symbolic and real world examples.

4.7.1 Removal Of Double Negations

Double negations are removed by CGP. Therefore, a formula such as:

((*p*))

will be reduced to:

p

and the proof taken from there. Also, nested double negations are removed so that:

(*p* ((*q*)))

would be reduced to:

(*p q*)

A real world example might be the statement *if clyde is an elephant then clyde is not a cat.* As written, this statement is an implication:

([*elephant* : *Clyde*]
(([*cat* : *Clyde*])))

Removal of the double negation gives:

([*elephant* : *Clyde*] [*cat* : *Clyde*])

which now states that *it is false that Clyde is an elephant and Clyde is a cat.*

4.7.2 Deiteration

Deiteration is the removal of graphs from a negated context if they have one of the following properties:

* the graph is an exact copy of another graph in the same context, therefore (*p p*) is reduced to (*p*),
* the graph projects onto another graph in the same context, therefore (*p p'*) is reduced to (*p'*),
* the graph is an exact copy of a graph at a less deep level of nesting such that the deeper-nested copy is contained within a context that is nested within the context of the less deeply-nested copy, therefore (*p* (*p*)) is reduced to (*p* ( )), but ((*p*) ((*p*))) is

not reduced to ((*p*) (( ))) because the deeper-nested *p* is not contained within a context that is nested within the context of the less deeply-nested *p*. However, in this case, the context could be reduced to ((*p*) ( )) by deiteration of (*p*).

* the graph is a subgraph of a graph at a less deep level of nesting and the deeper-nested graph is contained within a context that is nested within the context of the less deeply-nested graph, therefore (*p'* (*p*)) is reduced to (*p'* ( )), but ((*p'*) ((*p*))) is not reduced to ((*p'*) (( ))) because the deeper-nested *p* is not contained within a context that is nested within the context of the less deeply-nested *p'*. Note that the context ((*p*) ((*p'*))) is reducible to ((*p*) ( )) because the less deeply nested *p* is a subgraph of the more deeply nested *p'*.

* the graph is provable, either directly, by deduction or by the theorem prover.

A simple real world example might be the statement that *it is false that Clyde is a cat and that Clyde is a cat*:

([*cat* : *Clyde*] [*cat* : *Clyde*])

which can be reduced to *it is false that Clyde is a cat*:

([*cat* : *Clyde*])

4.7.3 Elimination Of Disjunctions

A *disjunction* has the general form:

((*p*) (*q*)).

A disjunction is true if either of the graphs *p* and *q* are true. If during a proof of a graph with this structure, the graph *p*, say, is proved and eliminated by deiteration to give:

(( ) (*q*))

then all graphs nested within the same context as the resulting ( ) can be eliminated. In this case the result is:

(( )).

A more complex example is:

((*p*) (*q* (*r*)) ( ))

which is reduced to give:

(( )).

As an example, the statement *Clyde is either an elephant or a cat* is represented as:

(([*elephant* : *Clyde*]) ([*cat* : *Clyde*]))

It is well known that Clyde is in fact an elephant and therefore the above context can be reduced by deiteration to:

(( ) ([*cat* : *Clyde*]))

The presence of the ( ) allows all other graphs within the same context to be removed, to give in this case:

(( ))

which is reducible to {} by removal of the double negation.

4.7.4 Iteration

Implication is transitive. Therefore if *p* implies *q* and *q* implies *r* then *p* implies *r*. The following would be an example knowledge base:

(*p* (*q*))

(*q* (*r*))

and the query would be:

(*p* (*r*))

The proof if this query requires the iteration of the two known implications in the knowledge base into the outer context:

(*p* (*r*) (*p* (*q*)) (*q* (*r*)))

following this, deiteration of *p* gives:

(*p* (*r*) ((*q*)) (*q* (*r*)))

Removal of double negations gives:

(*p* (*r*) *q* (*q* (*r*)))

deiteration of *q* gives:

(*p* (*r*) *q* ((*r*)))

Removal of double negations gives:

(*p* (*r*) *q r*)

Deiteration of *r* gives:

(*p* ( ) *q r*)

Elimination of disjunctions gives:

(( ))

which is directly provable.

An example of this is: given the two rules:

([*elephant* : *x]
([*mammal* : *x]->(*part*)->[*trunk*]))

([*mammal* : *x]
([*animal* : *x]->(*part*) ->[*fur*]))

the query is *are all elephants animals with fur?*:

([*elephant* : *x]
([*animal* : *x]->(*part*) ->[*fur*]))

The first stage of the proof is to find a known rule that links one of the nested graphs to some other graph. In this case that rule might be:

([*elephant* : **x*]
([*mammal* : **x*]->(*part*)->[*trunk*]))

and this would then give:

(([*elephant* : **x*]
([*mammal* : **x*]->(*part*)->[*trunk*]))
[*elephant* : **x*]
([animal : **x*]->(part)->[fur]))

deiteration of:

[*elephant* : **x*]

gives:

((([*mammal* : **x*]->(*part*)->[*trunk*]))
[*elephant* : **x*]
([*animal* : **x*]->(*part*)->[*fur*]))

removal of double negations gives:

(([*animal* : **x*]->(*part*)->[*fur*])
[*elephant* : **x*]
[*mammal* : **x*]->(*part*)->[*trunk*])

which can be proved directly in this case because it projects onto the known rule:

([*mammal* : **x*]
([*animal* : **x*]->(*part*)->[*fur*]))

4.7.5 Restriction

Under certain circumstances it is necessary to restrict type labels of concepts that lie on a line of identity on which there are other more specialised concepts. The Peirce rule of iteration allows the type labels of concepts and relations to be replace more general ones at deeper levels of nesting. The CGP rule of iteration allows exactly the same operation, with the constraint that the new type label is one of those that are present on the line of identity. This does not represent a restriction on the ability of CGP to prove a formula because the rule of deiteration, which the rule of iteration is intended to permit under certain conditions, requires that the deiterated graph is coreferent with its copy at the less deep level. Therefore there would be nothing gained by specialising a type label to one not on the line of identity.

One situation in which the rule of iteration is required is the following:

given the formula:

(*p* (*q*))

such that *p* contains a concept that is coreferent with a concept in *q* and the type label of the concept in *q* is less specialised than that in *p* and *q* cannot be deiterated directly because it has, perhaps, different relations to those attached to *p* attached to it. In this case, the formula can possibly be proved by the deiteration of *q* only if the type label of the concept in *q* is specialised to that of the coreferent concept in *p*. This is best explained by use of a real world example. If an elephant is a mammal with a trunk and a mammal is an animal with fur then is it true that an elephant is a mammal with fur? Because an elephant is a mammal then it should be possible to show that it has fur. This example is illustrated with an actual CGP run and it shows several other rules of inference in use. The proof that CGP produces is as follows:

```
>> !([elephant:*x]([mammal:*x]->(part)->[trunk])).

conformity relation added:
     elephant :: &.
```

```
lattice relation added:
     elephant << universal_type.

lattice relation added:
     absurd_type << elephant.

conformity relation added:
     mammal :: &.

lattice relation added:
     mammal << universal_type.

lattice relation added:
     absurd_type << mammal.

lattice relation added:
     elephant << mammal.

Is trunk a kind of mammal? n.

conformity relation added:
     trunk :: &.

lattice relation added:
     trunk << universal_type.

lattice relation added:
     absurd_type << trunk.

Is mammal a kind of trunk? n.

Checking for selectional constraint violations in:
     (
          [elephant:*1];
          (
               [mammal:*1]-
                       (part)->[trunk:*2]
          )
     ).

Is the general relation:

     [mammal:*3]-
             (part)->[trunk:*4].

the most general unambiguous usage? n.

Enter the most general unambiguous usage for the part relation:
[obj]->(part)->[obj].

Is obj a kind of mammal? n.

Is obj a kind of trunk? n.

conformity relation added:
     obj :: &.

lattice relation added:
     obj << universal_type.

lattice relation added:
     absurd_type << obj.
```

```
Is trunk a kind of obj? y.

lattice relation added:
     trunk << obj.

Enter definition for trunk in terms of obj:
.

Is mammal a kind of obj? y.

lattice relation added:
     mammal << obj.

lattice relation added:
     elephant << obj.

Enter definition for mammal in terms of obj:
.

conformity relation added:
     part :: &.

lattice relation added:
     part << universal_rel.

lattice relation added:
     absurd_rel << part.

Checking denotation of:
     (
          [elephant:*1];
          (
               [mammal:*1]-
                    (part)->[trunk:*2]
          )
     ).

I did not know that:
     (
          [elephant:*1];
          (
               [mammal:*1]-
                    (part)->[trunk:*2]
          )
     ).

Tidying up...

>> !([mammal:*x]([animal:*x]->(part)->[fur])).

Is animal a kind of obj? y.

Is animal a kind of trunk? n.

conformity relation added:
     animal :: &.

lattice relation added:
     animal << obj.

lattice relation added:
     animal << universal_type.
```

```
lattice relation added:
     absurd_type << animal.

Enter definition for animal in terms of obj:
.

lattice relation added:
     mammal << animal.

lattice relation added:
     elephant << animal.

Is trunk a kind of animal? n.

Is fur a kind of obj? y.

Is fur a kind of animal? n.

Is fur a kind of trunk? n.

conformity relation added:
     fur :: &.

lattice relation added:
     fur << obj.

lattice relation added:
     fur << universal_type.

lattice relation added:
     absurd_type << fur.

Enter definition for fur in terms of obj:
.

Is trunk a kind of fur? n.

Is animal a kind of fur? n.

Checking for selectional constraint violations in:
     (
          [mammal:*12];
          (
               [animal:*12]-
                       (part)->[fur:*13]
          )
     ).

Checking denotation of:
     (
          [mammal:*12];
          (
               [animal:*12]-
                       (part)->[fur:*13]
          )
     ).
```

```
I did not know that:
     (
          [mammal:*12];
          (
               [animal:*12]-
                       (part)->[fur:*13]
          )
     ).

Tidying up...

>> ?([elephant:*x]([mammal:*x]->(part)->[fur])).

Checking for selectional constraint violations in:
     (
          [elephant:*15];
          (
               [mammal:*15]-
                       (part)->[fur:*16]
          )
     ).

Checking denotation of:
     (
          [elephant:*15];
          (
               [mammal:*15]-
                       (part)->[fur:*16]
          )
     ).

The statement:
     (
          [elephant:*15];
          (
               [mammal:*15]-
                       (part)->[fur:*16]
          )
     ).

is:

TRUE because:
   1/0.     proof of:
                (
                     [elephant:*15];
                     (
                          [mammal:*15]-
                                  (part)->[fur:*16]
                     )
                );
   2/0.     is:
   3/1.          iteration of types:
   4/1.          giving:
                   (
                        [elephant:*15];
                        (
                             [elephant:*15]-
                                     (part)->[fur:*16]
                        )
                   );
```

```
5/1.        iteration of:
               (
                    [elephant:*1];
                    (
                         [mammal:*1]-
                              (part)->[trunk:*2]
                    )
               );
6/1.        giving:
               (
                    (
                         [elephant:*15];
                         (
                              [mammal:*15]-
                                   (part)->[trunk:*2]
                         )
                    );
                    [elephant:*15];
                    (
                         [elephant:*15]-
                              (part)->[fur:*16]
                    )
               );
7/1.        iteration of types:
8/1.        giving:
               (
                    (
                         [elephant:*15];
                         (
                              [elephant:*15]-
                                   (part)->[trunk:*2]
                         )
                    );
                    [elephant:*15];
                    (
                         [elephant:*15]-
                              (part)->[fur:*16]
                    )
               );
9/1.        deiteration of:
               [elephant:*15];
10/1.       giving:
               (
                    (
                         (
                              [elephant:*15]-
                                   (part)->[trunk:*2]
                         )
                    );
                    [elephant:*15];
                    (
                         [elephant:*15]-
                              (part)->[fur:*16]
                    )
               );
11/1.       removal of double negations:
```

```
12/1.          giving:
                  (
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       [elephant:*15];
                       (
                            [elephant:*15]-
                                   (part)->[fur:*16]
                       )
                  );
13/1.          deiteration of:
                  [elephant:*15];
14/1.          giving:
                  (
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       (
                            [elephant:*15]-
                                   (part)->[fur:*16]
                       )
                  );
15/1.          iteration of:
                  (
                       [mammal:*12];
                       (
                            [animal:*12]-
                                   (part)->[fur:*13]
                       )
                  );
16/1.          giving:
                  (
                       (
                            [mammal:*15];
                            (
                                 [animal:*15]-
                                        (part)->[fur:*16]
                            )
                       );
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       (
                            [elephant:*15]-
                                   (part)->[fur:*16]
                       )
                  );
17/1.          iteration of types:
18/1.          giving:
                  (
                       (
                            [elephant:*15];
                            (
                                 [elephant:*15]-
                                        (part)->[fur:*16]
                            )
                       );
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       (
                            [elephant:*15]-
                                   (part)->[fur:*16]
                       )
               );
```

```
19/1.          deiteration of:
                  [elephant:*15];
20/1.          giving:
                  (
                       (
                            (
                                 [elephant:*15]-
                                       (part)->[fur:*16]
                            )
                       );
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       (
                            [elephant:*15]-
                                    (part)->[fur:*16]
                       )
                  );
21/1.          deiteration of:
                  (
                       [elephant:*15]-
                              (part)->[fur:*16]
                  );
22/1.          giving:
                  (
                       (
                       );
                       [elephant:*15]-
                              (part)->[trunk:*2];
                       (
                            [elephant:*15]-
                                    (part)->[fur:*16]
                       )
                  );
23/1.          removal of disjunctions:
24/1.          giving:
                  (
                       (
                       )
                  );
25/1.          removal of double negations:
26/1.          giving:
                  {};
27/0.     to give:
             {}.

more? (y/n) n.
OK

>>
```

The important step in this proof is the one in which types are restricted by iteration. This takes place within the final deductive section of the proof and results in the modified rule as given after *so:*. The original query had been modified by insertions, deiterations and removals of double negations to a point where it would project onto a known rule when that rule was updated to correspond to the query as it had become.

4.7.6 Some Examples

A real formula will require several of the above steps in order to be reduced to the smallest amount. The first example is a proof of the formula *if p implies q then not q implies not p*. The proof of this formula is a proof that *modus tollens* is the inverse of *modus ponens*. It does not require any knowledge about the concepts involved as the whole formula is a *tautology*. In conceptual graphs, the formula is represented as (after use of the - command and the *%user* command to simplify the procedure):

```
user:>> ?(
                ([p:*x]([q:*x]))
                (
                (([q:*x])([p:*x])))
                )
          ).

conformity relation added:
     p :: &.

lattice relation added:
     p << universal_type.

lattice relation added:
     absurd_type << p.

conformity relation added:
     q :: &.

lattice relation added:
     q << universal_type.

lattice relation added:
     absurd_type << q.

Checking for selectional constraint violations in:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               (
                    (
                         [q:*0]
                    );
                    (
                         (
                              [p:*0]
                         )
                    )
               )
          )
     ).
```

```
Checking denotation of:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               (
                    (
                         [q:*0]
                    );
                    (
                         (
                              [p:*0]
                         )
                    )
               )
          )
     ).

The statement:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               (
                    (
                         [q:*0]
                    );
                    (
                         (
                              [p:*0]
                         )
                    )
               )
          )
     ).

is:

TRUE because:
```

```
1/0.    proof of:
                (
                    (
                        [p:*0];
                        (
                            [q:*0]
                        )
                    );
                    (
                        (
                            (
                                [q:*0]
                            );
                            (
                                (
                                    [p:*0]
                                )
                            )
                        )
                    )
                );
2/0.    is:
3/1.        removal of double negations:
4/1.        giving:
                (
                    (
                        [p:*0];
                        (
                            [q:*0]
                        )
                    );
                    (
                        [q:*0]
                    );
                    [p:*0]
                );
5/1.        deiteration of:
                (
                    [q:*0]
                );
6/1.        giving:
                (
                    (
                        [p:*0]
                    );
                    (
                        [q:*0]
                    );
                    [p:*0]
                );
7/1.        deiteration of:
                [p:*0];
8/1.        giving:
                (
                    (
                    );
                    (
                        [q:*0]
                    );
                    [p:*0]
                );
```

```
 9/1.          removal of disjunctions:
10/1.          giving:
                  (
                       (
                       )
                  );
11/1.          removal of double negations:
12/1.          giving:
                  {};
13/0.     to give:
             {}.

user:>>
```

The second group of examples shows how the system can use several separate rules to make proofs of formulae. All these examples make use of CGP's knowledge of the following rules:

```
(([p:*x]([q:*x]))([r:*x]([s:*x]))).

((p:*x]([q:*x]))[r:*x]([s:*x])).
```

The results of a selection of queries are best seen by reading the output:

```
user:>> !(([p:*x]([q:*x]))([r:*x]([s:*x])));
         (([p:*x]([q:*x]))[r:*x]([s:*x])).

conformity relation added:
     p :: &.

lattice relation added:
     p << universal_type.

lattice relation added:
     absurd_type << p.

conformity relation added:
     q :: &.

lattice relation added:
     q << universal_type.

lattice relation added:
     absurd_type << q.

conformity relation added:
     r :: &.

lattice relation added:
     r << universal_type.

lattice relation added:
     absurd_type << r.

conformity relation added:
     s :: &.

lattice relation added:
     s << universal_type.

lattice relation added:
     absurd_type << s.
```

```
Checking for selectional constraint violations in:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          [r:*0];
          (
               [s:*0]
          )
     );
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               [r:*0];
               (
                    [s:*0]
               )
          )
     ).

Checking denotation of:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          [r:*0];
          (
               [s:*0]
          )
     );
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               [r:*0];
               (
                    [s:*0]
               )
          )
     ).
```

```
I did not know that:
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          [r:*0];
          (
               [s:*0]
          )
     );
     (
          (
               [p:*0];
               (
                    [q:*0]
               )
          );
          (
               [r:*0];
               (
                    [s:*0]
               )
          )
     ).

Tidying up...

user:>> ?([p:*x]([q:*x])).

Checking for selectional constraint violations in:
     (
          [p:*22];
          (
               [q:*22]
          )
     ).

Checking denotation of:
     (
          [p:*22];
          (
               [q:*22]
          )
     ).

The statement:
     (
          [p:*22];
          (
               [q:*22]
          )
     ).

is:

FALSE because:
```

```
1/0.     proof of:
            (
                 [p:*22];
                 (
                      [q:*22]
                 )
            );
2/0.     is:
3/1.          iteration of:
                 (
                      (
                           [p:*0];
                           (
                                [q:*0]
                           );
                           [r:*0];
                           (
                                [s:*0]
                           )
                      );
4/1.          giving:
                 (
                      (
                           [p:*22];
                           (
                                [q:*22]
                           )
                      );
                      [r:*22];
                      (
                           [s:*22]
                      )
                 );
                 (
                      [p:*22];
                      (
                           [q:*22]
                      )
                 );
5/1.          deiteration of:
                 (
                      [p:*22];
                      (
                           [q:*22]
                      )
                 );
6/1.          giving:
                 (
                      [r:*22];
                      (
                           [s:*22]
                      )
                 );
                 (
                      [p:*22];
                      (
                           [q:*22]
                      )
                 );
```

```
7/1.          iteration of:
                 (
                      (
                           [p:*0];
                           (
                                [q:*0]
                           )
                      );
                      (
                           [r:*0];
                           (
                                [s:*0]
                           )
                      )
                 );
8/1.          giving:
              (
                   (
                        [p:*22];
                        (
                             [q:*22]
                        )
                   );
                   (
                        [r:*22];
                        (
                             [s:*22]
                        )
                   )
              );
              (
                   [r:*22];
                   (
                        [s:*22]
                   )
              );
              (
                   [p:*22];
                   (
                        [q:*22]
                   )
              );
9/1.          deiteration of:
                 (
                      [r:*22];
                      (
                           [s:*22]
                      )
                 );
```

```
10/1.          giving:
                  (
                        (
                              [p:*22];
                              (
                                    [q:*22]
                              )
                        )
                  );
                  (
                        [r:*22];
                        (
                              [s:*22]
                        )
                  );
                  (
                        [p:*22];
                        (
                              [q:*22]
                        )
                  );
11/1.          deiteration of:
                  (
                        [p:*22];
                        (
                              [q:*22]
                        )
                  );
12/1.          giving:
                  (
                  );
                  (
                        [r:*22];
                        (
                              [s:*22]
                        )
                  );
                  (
                        [p:*22];
                        (
                              [q:*22]
                        )
                  );
   13/1.          removal of disjunctions:
   14/1.          giving:
                     (
                     );
   15/0.       to give:
                  ().

user:>> ?[p:*x]([q:*x]).

Checking for selectional constraint violations in:
      [p:*6];
      (
            [q:*6]
      ).
```

```
Checking denotation of:
     [p:*6];
     (
          [q:*6]
     ).

The statement:
     [p:*6];
     (
          [q:*6]
     ).

is:

TRUE because:
   1/0.     proof of:
               [p:*6];
               (
                    [q:*6]
               );
2/0.       is:
3/1.            iteration of:
                   (
                        (
                             [p:*0];
                             (
                                  [q:*0]
                             )
                        );
                        [r:*0];
                        (
                             [s:*0]
                        )
                   );
4/1.            giving:
                   (
                        (
                             (
                                  [p:*6];
                                  (
                                       [q:*6]
                                  )
                             );
                             [r:*6];
                             (
                                  [s:*6]
                             )
                        );
                        (
                             [p:*6];
                             (
                                  [q:*6]
                             )
                        )
                   );
5/1.               deiteration of:
                   (
                        [p:*6];
                        (
                             [q:*6]
                        )
                   );
```

```
6/1.          giving:
                 (
                      (
                           [r:*6];
                           (
                                [s:*6]
                           )
                      );
                      (
                           [p:*6];
                           (
                                [q:*6]
                           )
                      )
                 );
7/1.          iteration of:
                 (
                      (
                           [p:*0];
                           (
                                [q:*0]
                           )
                      );
                      (
                           [r:*0];
                           (
                                [s:*0]
                           )
                      )
                 );
8/1.          giving:
                 (
                      (
                           [p:*6];
                           (
                                [q:*6]
                           )
                      );
                      (
                           [r:*6];
                           (
                                [s:*6]
                           )
                      )
                 );
                 (
                      [r:*6];
                      (
                           [s:*6]
                      )
                 );
                 (
                      [p:*6];
                      (
                           [q:*6]
                      )
                 )
            );
```

```
9/1.        deiteration of:
               (
                    [r:*6];
                    (
                         [s:*6]
                    )
               );
10/1.       giving:
               (
                    (
                         (
                              [p:*6];
                              (
                                   [q:*6]
                              )
                         );
                         (
                              [r:*6];
                              (
                                   [s:*6]
                              )
                         );
                         (
                              [p:*6];
                              (
                                   [q:*6]
                              )
                         )
                    );
11/1.       deiteration of:
               (
                    [p:*6];
                    (
                         [q:*6]
                    )
               );
12/1.       giving:
               (
                    (
                    );
                    (
                         [r:*6];
                         (
                              [s:*6]
                         )
                    );
                    (
                         [p:*6];
                         (
                              [q:*6]
                         )
                    )
               );
13/1.       removal of disjunctions:
14/1.       giving:
               (
                    (
                    )
               );
15/1.       removal of double negations:
```

```
  16/1.          giving:
                    {};
  17/0.     to give:
               {}.

user:>> ?[p].

Checking for selectional constraint violations in:
     [p:*8].

Checking denotation of:
     [p:*8].

The statement:
     [p:*8].

is:

TRUE because:
   1/0.     proof of:
               [p:*8];
   2/0.     is:
   3/1.          iteration of:
                    (
                         (
                              [p:*0];
                              (
                                   [q:*0]
                              )
                         );
                         [r:*0];
                         (
                              [s:*0]
                         )
                    );
 4/1.          giving:
                    (
                         (
                              (
                                   [p:*8];
                                   (
                                        [q:*8]
                                   )
                              );
                              [r:*8];
                              (
                                   [s:*8]
                              )
                         );
                         (
                              [p:*8]
                         )
                    );
 5/1.          deiteration of:
                    (
                         [p:*8];
                         (
                              [q:*8]
                         )
                    );
```

```
6/1.        giving:
               (
                    (
                         [r:*8];
                         (
                              [s:*8]
                         )
                    );
                    (
                         [p:*8]
                    )
               );
7/1.        iteration of:
               (
                    (
                         [p:*0];
                         (
                              [q:*0]
                         )
                    );
                    (
                         [r:*0];
                         (
                              [s:*0]
                         )
                    )
               );
8/1.        giving:
               (
                    (
                         (
                              [p:*8];
                              (
                                   [q:*8]
                              )
                         );
                         (
                              [r:*8];
                              (
                                   [s:*8]
                              )
                         )
                    );
                    (
                         [r:*8];
                         (
                              [s:*8]
                         )
                    );
                    (
                         [p:*8]
                    )
               );
9/1.        deiteration of:
               (
                    [r:*8];
                    (
                         [s:*8]
                    )
               );
```

```
10/1.          giving:
                  (
                        (
                              (
                                    [p:*8];
                                    (
                                          [q:*8]
                                    )
                              )
                        );
                        (
                              [r:*8];
                              (
                                    [s:*8]
                              )
                        );
                        (
                              [p:*8]
                        )
                  );
11/1.          deiteration of:
                  (
                        [p:*8];
                        (
                              [q:*8]
                        )
                  );
12/1.          giving:
                  (
                        (
                        );
                        (
                              [r:*8];
                              (
                                    [s:*8]
                              )
                        );
                        (
                              [p:*8]
                        )
                  );
13/1.          removal of disjunctions:
14/1.          giving:
                  (
                        (
                        )
                  );
15/1.          removal of double negations:
16/1.          giving:
                  {};
17/0.     to give:
               {}.

     user:>>
```

4.7.7 Resolution

The theorem prover is not a resolution theorem prover and the term 'resolution' as used here should not be thought of as referring to the resolution principle. A resolution theorem prover attempts to show that a formula cannot be proved false and is therefore true. This is fine for a closed world system but is not correct for an open world system.

There are, however, cases where a tautology theorem prover, such as this one, might be left with some graphs that cannot be proved and therefore appear to render the original formula unknown. In some cases this is true but in others it is possible to show that these graph do not actually matter. This occurs when one attempt at finding a proof fails because the unknown graphs cannot be shown to be true and another attempt at a proof fails because they cannot be shown to be false. In other words, there is a proof whether the unknown graphs are true or false.

A *blocks world* example will be used to illustrate the point. In this example there are three blocks *a*, *b* and *c*. Block *a* is green and is placed on top of block *b* whose colour is unknown and which is placed on block *c* which is not green. The question is 'is there a green block immediately above a nongreen block?'. Clearly the answer is 'yes' because if block *b* were green then it would be a green block on top of a nongreen block *c*. If block *b* is nongreen then block *a* would be a green block on top of a nongreen block. In other words the query is definitely true although we cannot say what colour block *b* actually is. It is therefore possible to say that if the unprovable graphs could lead to a proof either if they were true or false then there must be a proof.

The information in this example can be coded as:

```
user:>> ![block:a]-
                    (attr)->[colour:green]
                    (on)->[block:b]->(on)->[block:c];
          ([block:c]->(attr)->[colour:green]).
```

The query is:

```
user:>> ?[block]-
                  (attr)->[colour:green]
                  (on)->[block:*x]
          ([block:*x]->(attr)->[colour:green]).
```

When submitted to CGP the result is:

```
Checking for selectional constraint violations in:
     [block:*2]-
               (on)->[block:*3]
               (attr)->[colour:green];
     (
          [block:*3]-
                    (attr)->[colour:green]
     ).

Checking denotation of:
     [block:*2]-
               (on)->[block:*3]
               (attr)->[colour:green];
     (
          [block:*3]-
                    (attr)->[colour:green]
     ).

The statement:
     [block:*2]-
               (on)->[block:*3]
               (attr)->[colour:green];
     (
          [block:*3]-
                    (attr)->[colour:green]
     ).

is:

TRUE because:
```

```
1/0.     proof of:
            [block:*2]-
                    (attr)->[colour:green];
            [block:*2]-
                    (on)->[block:*3];
            (
                 [block:*3]-
                         (attr)->[colour:green]
            );
2/0.     is:
3/1.          proof possible if it is TRUE that:
                 (
                      [block:b]-
                              (attr)->[colour:green]
                 );
4/1.          proof possible if it is TRUE that:
                 [block:b]-
                         (attr)->[colour:green];
5/1.          resolution of unknown graphs:
                 [block:b]-
                         (attr)->[colour:green];
                 (
                      [block:b]-
                              (attr)->[colour:green]
                 );
6/0.     to give:
            {}.
```

4.8 The Role Of The Theorem Prover In Deduction

The theorem prover is more than an academic tool. It can be invoked at any point in the proof or validation of any graph. As an example, consider the case:

$$(((p)(q))(r))$$

which can be stated in conceptual graphs as:

```
(
        (([p:*x])([q:*x]))
        (
        [r:*x]
        )
).
```

which states that if the object **x* is either a *p* or a *q* then it is also an *r*. A real world example might be: *if x is a bird or x is an insect or x is a bat then x is a flying animal*:

```
user:>> ~.

all retracted

user:>> !(
                (([bird:*x])([bat:*x])([insect:*x]))
                (
                [flying_animal:*x]
                )
          ).
```

Now, there may be a fact known about some entity called Joey, who is a budgerigar:

```
user:>> ![budgerigar:Joey].
```

We inform CGP of the fact that *budgerigar << bird*:

```
user:>> !budgerigar << bird.
```

The query 'is Joey a flying_animal' is made:

```
user:>> ?[flying_animal:Joey].

Checking for selectional constraint violations in:
     [flying_animal:Joey].

Checking denotation of:
     [flying_animal:Joey].

The statement:
     [flying_animal:Joey].

is:

TRUE because:
   1/0.      proof of:
                [flying_animal:Joey];
   2/0.      is:
   3/1.           deduction:
   4/1.           known rule:
                     (
                          (
                               (
                                    (
                                         [bird:*0]
                                    );
                                    (
                                         [bat:*0]
                                    );
                                    (
                                         [insect:*0]
                                    )
                               );
                               (
                                    [flying_animal:*0]
                               )
                          );
              5/1.        so:
                          (
                               (
                                    [flying_animal:Joey]
                               );
                               (
                                    (
                                         [budgerigar:Joey]
                                    );
                                    (
                                         [bat:Joey]
                                    );
                                    (
                                         [insect:Joey]
                                    )
                               )
                          );
```

```
6/1.          proving antecedent:
                 (
                      (
                           [budgerigar:Joey]
                      );
                      (
                           [bat:Joey]
                      );
                      (
                           [insect:Joey]
                      )
                 );
7/1.          proving graph in antecedent:
                 [budgerigar:Joey];
8/1.          proof of:
                 [budgerigar:Joey];
 9/1.            is:
10/2.                 fact:
                         [budgerigar:Joey];
11/1.            giving:
                    {};
12/1.            result:
                    [flying_animal:Joey];
13/0.      to give:
              {}.

user:>>
```

The theorem prover has helped the deduction mechanism to prove one of its antecedents and as such is a necessary part of the deduction process.

4.9 Other Forms Of Reasoning

The main form of reasoning of which the system is capable is precise logical reasoning. There are many other forms of reasoning that have been identified and formalised to an extent. Some of these can be performed with standard logic but others cannot. In this section, one such form of reasoning will be discussed; reasoning with possible worlds and modality.

4.9.1 Possible Worlds

A possible world proof involves the generation of a theory which can link together known facts and possibly explain them. CGP produces such theories (subject the the t command having been used to enable the facility) and attempts to answer queries by reference to these. To illustrate, we use the following example which is taken from a logic puzzle book of the type available at newsagents.

In this example there are four girls: Cheryl, Maddie, Isobel and Stella. Each of them wants to eat a different type of food, likes a different kind of music and wishes to participate in a different activity. We know the following facts:

- Isobel eats Chinese food.
- One of the girls wants to eat casserole.
- One of the girls both likes rock music and wants to eat burgers. This girl is not Stella.
- One of the girls both wishes to do some decorating and to eat vegetarian food.
- Cheryl likes Country music
- One of the girls likes Heavy Metal music.
- One of the girls both wishes to repair her bicycle and listen to Classical music. This girls is not

Isobel.

- One girl wishes to read.
- One girl wishes to do some dressmaking. This girls is not Maddie.

The nature of the puzzle is that each girls only likes one kind of music, only wishes to participate in one activity and only eats one kind of food. We therefore add this requirement to the list of known facts. The encoding into conceptual graphs may leave something to be desired in terms of the semantics of some of the relations used but the puzzle is not encoded in a way that ensures that the required answer will be generated.

The following graphs are used to model the puzzle and are submitted to CGP:

```
>> ![ girl : maddie ].

>> ![ girl : stella ].

>> ![ girl : isobel ]-
     ( eat )->[ food_type : chinese ].

>> ![ girl ]-
     ( eat )->[ food_type : casserole ].

>> ![ girl : *x ]-
     ( like )->[ music_type : rock ]
     ( eat )->[ food_type : burgers ];
    (
       [ girl : stella = *x ]
     ).

>> ![ girl ]-
     ( wish )->[ activity_type : decorating ]
     ( eat )->[ food_type : vegetarian ].

>> ![ girl : cheryl ]-
     ( like )->[ music_type : country ].

>> ![ girl ]-
     ( like )->[ music_type : metal ].

>> ![ girl : *y ]-
     ( wish )->[ activity_type : repair_bicycle ]
     ( like )->[ music_type : classical ];
    (
       [ girl : isobel = *y ]
     ).

>> ![ girl ]-
     ( wish )->[ activity_type : reading ].

>> ![ girl : *z ]-
     ( wish )->[ activity_type : dressmaking ].
    (
       [ girl : maddie = *z ]
     ).
```

The following rule is the extra information that is implicit in such puzzles and states that each girl in the puzzle can only be involved in one relation of each type:

```
(
   [girl]-
         (*rel)->[universal_type]
         (*rel)->[universal_type]
).
```

After submitting these graphs to CGP and turning on the theorem generation facility the query and output is:

```
>> t.

Theory generation = on

>> ?[girl:*48]-
         (like)->[music_type:*51]
         (eat)->[food_type:*50]
         (wish)->[activity_type:*49].

Checking for selectional constraint violations in:
     [girl:*52]-
            (like)->[music_type:*55]
            (eat)->[food_type:*54]
            (wish)->[activity_type:*53].

Checking denotation of:
     [girl:*52]-
            (like)->[music_type:*55]
            (eat)->[food_type:*54]
            (wish)->[activity_type:*53].

The statement:
     [girl:*52]-
            (like)->[music_type:*55]
            (eat)->[food_type:*54]
            (wish)->[activity_type:*53].

is:

TRUE because:
   1/0.     proof of:
               [girl:*52]-
                     (wish)->[activity_type:*53];
               [girl:*52]-
                     (eat)->[food_type:*54];
               [girl:*52]-
                     (like)->[music_type:*55];
   2/0.     is:
```

```
3/1.        possible theory:
               [girl:stella]-
                        (eat)->[food_type:casserole]
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
               [girl:maddie]-
                        (like)->[music_type:rock]
                        (eat)->[food_type:burgers]
                        (wish)->[activity_type:reading];
               [girl:cheryl]-
                        (wish)->[activity_type:decorating]
                        (eat)->[food_type:vegetarian]
                        (like)->[music_type:country];
               [girl:isobel]-
                        (like)->[music_type:metal]
                        (wish)->[activity_type:dressmaking]
                        (eat)->[food_type:chinese];
               (
                    [girl:isobel=stella]
               );
               (
                    [girl:isobel=maddie]
               );
               (
                    [girl:maddie=stella]
               );
               (
                    [girl:*26]-
                             (*25)->[universal_type:*28]
                             (*25)->[universal_type:*27]
               );
4/1.        generated by:
5/1.        known facts combined:
               [girl:*35]-
                        (wish)->[activity_type:dressmaking];
               [girl:isobel]-
                        (eat)->[food_type:chinese];
6/1.        giving:
               [girl:isobel]-
                        (wish)->[activity_type:dressmaking]
                        (eat)->[food_type:chinese];
7/1.        known facts combined:
               [girl:*31]-
                        (wish)->[activity_type:reading];
               [girl:maddie];
8/1.        giving:
               [girl:maddie]-
                        (wish)->[activity_type:reading];
9/1.        known facts combined:
               [girl:*32]-
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
               [girl:stella];
10/1.       giving:
               [girl:stella]-
                        (wish)->[activity_type:repair_bicycle]
                        (like)->[music_type:classical];
```

```
11/1.          known facts combined:
                  [girl:*30]-
                          (like)->[music_type:metal];
                  [girl:isobel]-
                          (wish)->[activity_type:dressmaking]
                          (eat)->[food_type:chinese];
12/1.          giving:
                  [girl:isobel]-
                          (like)->[music_type:metal]
                          (wish)->[activity_type:dressmaking]
                          (eat)->[food_type:chinese];
13/1.          known facts combined:
                  [girl:*34]-
                          (wish)->[activity_type:decorating]
                          (eat)->[food_type:vegetarian];
                  [girl:cheryl]-
                          (like)->[music_type:country];
14/1.          giving:
                  [girl:cheryl]-
                          (wish)->[activity_type:decorating]
                          (eat)->[food_type:vegetarian]
                          (like)->[music_type:country];
15/1.          known facts combined:
                  [girl:*33]-
                          (like)->[music_type:rock]
                          (eat)->[food_type:burgers];
                  [girl:maddie]-
                          (wish)->[activity_type:reading];
16/1.          giving:
                  [girl:maddie]-
                          (like)->[music_type:rock]
                          (eat)->[food_type:burgers]
                          (wish)->[activity_type:reading];
17/1.          known facts combined:
                  [girl:*29]-
                          (eat)->[food_type:casserole];
                  [girl:stella]-
                          (wish)->[activity_type:repair_bicycle]
                          (like)->[music_type:classical];
18/1.          giving:
                  [girl:stella]-
                          (eat)->[food_type:casserole]
                          (wish)->[activity_type:repair_bicycle]
                          (like)->[music_type:classical];
19/1.          result:
                  [girl:isobel]-
                          (like)->[music_type:metal]
                          (eat)->[food_type:chinese]
                          (wish)->[activity_type:dressmaking];
20/0.      to give:
              {}.

more? (y/n) n.
OK

>>
```

In this example we reply n to the query more? but if y had been answered instead the system would have gone on to produce up to three further results, each one a different part of the overall solution.

4.10 More About The CGP Interface

The basic interface to CGP is simple, with single character commands, but underlying this apparent simplicity is a more complex knowledge base maintenance system. This system performs two functions: it keeps the knowledge base compact and properly organised by evaluating all new graphs upon entry and it performs a set of elementary inferences. As a result, the knowledge base is not simply a storage place for graphs and therefore it will not always exactly reflect that which was typed. In addition, graphs generated by the inference engine are examined in the same way as new graphs, therefore the knowledge base will change during the use of CGP in more complex ways than might be imagined. As each inference is made, CGP reports it to the screen.

The overall effect of these operations is that of an automatic classification mechanism which can take an input and classify it according to the definitions that CGP knows. Currently, this mechanism is somewhat primitive, especially in the field of inexact reasoning and default reasoning, but it provides the basis for a neural net-like classification and correlation mechanism.

Because of the operations of the interface, the * command will rarely display only that which has been typed. It will usually show that which was typed in a different form and will also show extra information that was not typed but was implied in what was typed.

4.10.1 Evaluation Of New Graphs By CGP

When a new graph is entered into CGP it is evaluated as if it were a query. This has the result that if the proposition made by the graph is already known, or is inconsistent with the knowledge base, then CGP does not record it. The obvious benefit of this is that the amount of knowledge within the system is kept to a minimum.

In addition to this, CGP's interface also joins the simple graphs into as few graphs as possible. This is also a space-saving measure but its main function is that of fulfilling the open world model's requirement that each individual marker should only appear once in the set of true graphs.

4.10.2 inferences Made By The Interface

The inferences made by the interface are of a simple nature and are primarily aimed at making the system more efficient. The following is a list of inferences that CGP makes as graphs are entered:

* Type lattice and relational lattice relations are extracted from type definitions and relational definitions and added to the appropriate lattice. This is a reasonable thing to do because these relations are implicit in each such definition. Redundancy within the lattices is removed.

* Newly encountered type labels and relational labels are added to the appropriate lattice between the appropriate universal type and absurd type.

* Conformity relations are extracted from new simple graphs. Newly encountered individuals are added to the conformity relations table and previously known individuals' conformities are updated if a new conformity to a subtype of the previously known conformity is discovered. Each newly encountered type or relation is given a unique prototype conformity.

* When it is known that an individual conforms to two types, neither of which is a subtype of the other, a new type label is created that is a combination of the two types. The individual is recorded as conforming to the new type and its conformities to the other two types are erased. This ensures that the type lattice is exactly that.

* When a new type definition is encountered, each concept within the semantic net that could be specialised to the type being defined is specialised.

* Each type label and relational label within an input graph is specialised to its most specialised level appropriate to its context.

Pairs of compound negated contexts are combined in any one of three ways:
a. (p (r)) ∧ (q (r)) → (((p) (q)) (r))
b. (p (q)) ∧ (p (r)) → (p (q r))
c. one is inserted into the other and an attempt is made to reduce the combination to something smaller than either original context.
These three rules lead to some reduction in the number of rules that CGP stores without reducing the amount of knowledge in the knowledge base.

4.11 CGP'S Operational Modes

As stated in section 3, CGP can operate in several different *modes*. Currently these are:

* speed (slow/quick),
* reports (verbose/brief),
* interaction (findall/single solution),
* evaluation (full/partial),
* check with user (on/off),
* iteration (on/off).

The commands that are responsible for selection of operational modes act as a *toggle*, and switch between one mode of each pair above and the other. Each mode will be described along with its purpose:

4.11.1 Speed

Selected with the s command, the speed of the system can be increased slightly by the use of *quick* mode. The price paid is that only one proof of each solution is found, which may not necessarily be the most intuitive. However, for long proofs involving large theorems, *quick* mode may be essential in ensuring that a proof is found within a reasonable time.

4.11.2 Reports

When *verbose* mode is selected each proof is given in full and when *brief* mode is selected only those facts and axioms that terminated the proof are reported. Should any query produce many solutions then *brief* mode is essential in preventing vast quantities of unmanageable output.

For users not too familiar with the theory of conceptual graphs, and users who trust that CGP's reasoning is always correct, then *brief* mode is sufficient, but for detailed analysis of the proof, *verbose* mode is required.

4.11.3 Interaction

The interaction with CGP can be either of two forms: one in which each proof is offered by CGP as it is found, following which the user is given the opportunity to stop or request more proofs (*single solution*), and one in which all proofs will be found before any are reported (*findall*). The advantage of the *single solution* mode is that the user can stop an evaluation as soon as the required result has been obtained whereas with *findall* mode the user cannot intervene. The disadvantage is that *single solution* mode does not allow batch processing in which interaction with the user is absent.

4.11.4 Evaluation

The mode of evaluation can be either *full* or *partial*. In *full* mode, the whole power of CGP and Peirce logic is available, whilst in *partial* mode, the theorem prover is abolished and a Prolog like deduction mechanism is left. This can be useful for optimising the system in situations where it is known that arbitrary theorems need never be proved and will result in a considerable increase in speed, but it is not recommended for general operation.

The theorem prover is vital for the evaluation of new rules for consistency. For example, if it known that:

$$[p : {}^*x] \ ([q : {}^*x])$$

then any attempt to add the rule:

$$([p : {}^*x]([q : {}^*x]))$$

should fail since this rule contradicts known facts. For this reason, when *partial* mode is selected for evaluation of queries, CGP temporarily reverts to *full* mode when new data are being presented. This may occasionally result in a loss of speed.

4.11.5 Check With User

CGP can be configured to ask the user whether any graph that is unknown to the system, but which is necessary for a proof to continue, is TRUE,FALSE or UNKNOWN. This feature allows CGP to answer more queries than it otherwise would and in the future will form a channel by which separate incarnations of the system, each with different knowledge bases, will be able to help each other with specialist knowledge.

The check with user facility is not available when scripts are being processed or when an assertion to the system is being made. In the first case, interaction with the user must be suspended because that is the nature of scripts and in the second case, any graph that cannot be proved should always be added to the knowledge base, which will happen whether check with user mode is set or not.

4.11.6 Iteration

The rule of iteration is the one inference rule that adds information to a context that is being proved. As a result. it is possible that proofs that require this rule may become very long and combinatorial explosion may result (although CGP contains guards against this). If it is known that any query will definitely not require the rule of iteration for its proof (for example, if the knowledge base is a simple one for which lookup will suffice) then the rule of iteration can (and should) be disabled with the i command.

APPENDIX I - CGP MESSAGES

Operational Messages

The ? command issues the message:

The statement:
<graph>
is

and then produces four basic types of output. These are:

1. TRUE because:

issued when a query evaluates to true,

2. FALSE because:

issued when a query evaluates to false. Under some circumstances the word 'because' is omitted and no further explanation is given. This occurs when a negated context is reducible to (),

3. INCONSISTENT

issued when the knowledge base is itself inconsistent,

4. UNKNOWN

issued when the query cannot be proved.

The ! command issues the following messages:

1. Yes, I know that:
<graph>
<reason>

issued when an input graph is already known,

2. I think the opposite is true to:
<graph>
<reason>

issued when an input graph is false for some reason,

3. This fact is inconsistent
<graph>
<reason>

issued when the knowledge base is already inconsistent. This message allows inconsistent facts to be located,

4. I did not know that
<graph>

issued when an input graph is wholly or partly unknown.

Outputs of type 4 may also include a warning message to the effect that the system gave up because of the timestep limit set by the @ command. In addition, warnings are displayed when some of the system options are disabled, *eg* the theorem prover or the Prolog like mechanism.

In addition, outputs of types 1, 2 and 3 include an explanation of the reasoning used by the program (if verbose mode is selected). Possible information given includes the following informative messages:

Information Messages

<graph>
above graph assumed to be the most general unambiguous usage

issued by the canonicity checking mechanism when the default canonical graph (.) has been entered by the user.

after reproof:
<list>

issued when data had to be added to the knowledge base and a proof repeated before a result could be found.

<label> allocated marker #<marker>

issued by the parser to show which marker has been allocated to the label *some* or to a label starting with *some_*.

antecedent not relevant:
<graph>

issued when the proof of the antecedents of a rule leads to the possibility of one or more being removed by deiteration rather than proof.

because:
<list>

issued to announce an explanation of part of a proof.

because of unknown graphs:
<list>

issued when a query has not been proved and the unknown graphs which prevented the proof are reported.

combination of antecedents

issued by the knowledge base maintenance system to show how two formulae were combined into one.

combination of consequents

issued by the knowledge base maintenance system to show how two formulae were combined into one.

concept specialised:
<concept>

issued when the knowledge base maintenance system automatically specialises a concept in either an input graph or the semantic net.

conformity relation added:
<conformity relation>

issued when the knowledge base maintenance system automatically adds a new conformity to the conformity relations table.

contradicted by:
<list>

issued when an attempt to tell CGP that two entities are the same but CGP has found contradictory information.

deduction:
<list>

issued when a graph has been proved by the Prolog-like inference mechanism.

definition inconsistent with lattice:
<graph>

issued when a definition that produces a lattice relation which defines a type *t1 < t2* when it is known that *t2 < t1*.

deiteration of:
<graph>

issued when deiteration of a graph more deeply nested graph than a copy which dominates it has occurred. **<graph>** is the less deeply nested copy.

removal of disjunctions:
<graph>

issued after the elimination of oddly enclosed graphs following the reduction of one of them to ().

fact:
<graph>

issued when a known fact is used during a proof.

false equality:
<graph>

issued when **<graph>** contains an equality that can be shown to be false.

Generating inferences...

issued to state what stage the system has reached in the addition of new knowledge to the knowledge base.

Getting tired! Continue? (y/n):

issued when not running a script and the timeout number has been reached. Reply 'y' to continue and 'n' to abort.

graphs inferred:
<graphs>

generated by the data driven deduction mechanism to state any new deductions.

giving:
<graph>

issued to show the result of an operation.

iteration of:
<graph>

issued when a known formula has been iterated into a query context.

into:
<graph>

issued when the knowledge base maintenance system combines two rules into a single, more compact logically equivalent form.

Is:
<list>

issued when reporting the means of proof of a graph. **<list>** is an arbitrary list of explanations.

Iteration of types:

issued when type labels have been iterated to coreferent, dominated concepts.

known context:
<context>

issued when a proof has involved the instantiation of a generic context with a known graph from the knowledge base.

known facts combined:
<graph1>
<graph2>

issued by the theory generator to show which graphs have been joined in the generation of the theory.

known rule:
<graph>

issued for each negated context with more than one nested structure that has been found in the knowledge base and used in a deduction.

lambda abstraction added:
<lambda>

issued when a double implication that implies a definition is detected in the knowledge base and a corresponding lambda abstraction needs to be added.

lambda abstractions combined:
<lambda>
<lambda>

issued when the system has shown that two type labels or relational labels are aliases and the definitions for each have been combined.

lattice relation added:
<lattice relation>

issued when the knowledge base maintenance system automatically extracts and adds a new lattice relation (type or relation) to the appropriate lattice.

no such lattice relation

issued when in brief mode to show that a lattice relation was not known.

objects iterated:
<list>

issued when an input context is normalised so that individual referents are copied to all deeper-nested coreferent concepts or relations.

possible theory:
<graphs>
generated by:

issued by the theory generator to show which theory has been used to answer the query.

previous proof:
<list>

announces a proof which has been previously found and reused. The proof may not necessarily have been previously printed if the overall proof of which it was a part failed at some point after its generation.

proof of:
<graph>

issued for each nested graph deiterated from a context. It forms an opening bracket which starts a complete proof of a subpart of the main query.

proof of antecedent:
<graph>

issued for each true antecedent of a rule.

proof possible if it is TRUE that:
<graph>

issued by the resolution mechanism to show which graphs would have allowed a proof to be found if they had been provable.

proving antecedent:
<graph>

issued to announce the fact that the next graph to be proved is an 'antecedent' of a rule which is being used by the deduction mechanism.

proving graph in antecedent:
<graph>

issued when an antecedent consists of more than a simple graph and the system is attempting to prove a simple graph from within it.

proving nested structure:
<graph>

issued when a graph nested within a context is proved and then deiterated.

recording data prior to reproof of:

under certain circumstances it is necessary for the system to record deductions made at certain points during a proof and then repeat the proof.

Recording rule:
<graph>

issued when a rule which may be implicitly true is not explicitly true and is therefore recorded.

reduced to:
{ }

issued when brief mode is set to show that a theorem has been reduced to { }.

removal of double negations:
<graph>

issued following the removal of double negations.

removal of negations:

issued when a set of negations has been reduced to the empty set.

reproving

issued to inform the user of what stage the proof of a graph has reached.

resolution of unknown graphs:
<graphs>

issued when the graphs **<graphs>** have been used by the resolution mechanism to show that a proof is possible whatever the truth values of **<graphs>** but that the exact proof cannot be determined.

result:
<graph>

issued at the end of a proof and gives the graph whose derivation completed the proof and that answered the original query.

formulae combined:
<list>

issued by the knowledge base maintenance system to state which two rules it is able to combine into one.

so:
<graph>

issued after the instantiation of the rule displayed previously.

taken on trust!

issued when the user informs CGP of the truth of a graph that CGP could not prove itself.

The proposition <graph> is unknown to me. Is it true, false or unknown? (t/f/u):

issued when the the system needs to consult the user about a graph that is required to be proved as part of the proof of some other graph.

tidying up...

issued when the knowledge base is being updated after addition of new knowledge. Its purpose is to warn the user that there may be a slight delay.

to:
<concept>

issued when a conceptual type within the semantic net is automatically specialised by the knowledge maintenance system. **<concept>** is the new concept.

to give:
<graph>

issued to show the result of an operation.

where:

issued to show where an object has been iterated to.

In addition to these messages, the &, p, s, v and o commands also produce output, but this is documented in section 3 and not repeated here.

The above messages are produced in groups which correspond to the proofs of subparts of larger graphs. To aid readability, the subparts are indented and, additionally, the messages that belong to a given subpart are allocated numbers that reflect the depth of nesting of each subproof within the whole proof.

Messages Produced By The Canonicity Checking System

The following messages will be issued as appropriate by the canonicity checking system. All of them will require information to be supplied by the user. In this section only the two modes of operation that either disallow or allow the use of upper and lower bound canonical graphs will be called standard mode and extended mode.

<graph>
Is this OK?

issued when a definition of a relation has been given and CGP has compiled one of several possible forms in terms of Peirce logic. Each form is offered to the user for verification before further processing. The coreference links need special attention in the inspection of the form offered. The response to this question is *y.* or *n.*.

Enter definition for *type1* in terms of *type2*:

issued when a request is made by the system for the definition of a newly encountered type or relation. If the definition is unknown then the default response is the full stop on its own.

Enter the most general unambiguous usage for the *relation* relation:

This a request made in standard mode for a graph of one relation which gives the most general unambiguous usage for the relation *relation*.

Enter the upper bound for the *relation* relation:

This a request made in extended mode for a graph of one relation which gives the upper bound for the relation *relation*.

Enter the lower bound for the *relation* relation:

This a request made in extended mode for a graph of one relation which gives the lower bound for the relation *relation*.

default upper bound assumed

issued when the default null response is given to a request for a canonical graph.

default lower bound assumed

issued when the default null response is given to a request for a canonical graph.

above graph assumed to be the most general unambiguous usage

issued when the default null response is given to a request for a canonical graph.

Is the general relation:
<graph>
the most general unambiguous usage?

issued when in standard mode for each newly-encountered relation in an attempt to find out the most general unambiguous usage for the new relation. Answer *y.* or *n.* as appropriate.

Is the general relation:
<graph>
the upper bound?

issued when in extended mode for each newly-encountered relation in an attempt to find the upper bound for the new relation. Answer *y.* or *n.* as appropriate.

Is the general relation:
<graph>
the lower bound?

issued when in extended mode for each newly-encountered relation in an attempt to find the lower bound for the new relation. Answer *y.* or *n.* as appropriate.

Is *type1* a kind of *type2*?

issued when a newly encountered type or relation is being placed within the appropriate lattice. The answer is *y.* or *n.* and must be lower case. Use *n* as the default.

***** ERROR *** try again:**

A graph that has been offered to CGP as a definition or most general unambiguous usage is incorrectly formed, either syntactically or semantically. Common faults are not using the type or relation in terms of which CGP requested the definition.

Error And Warning Messages

The following messages may be issued from time to time as a result of errors either within input data or with the operation of CGP.

& command ignored

. command ignored

z command ignored

Although almost any command can be used within a script, the & and . commands would cause problems if it were allowed to take effect. The z command does nothing anyway. Any attempt to use these commands is therefore ignored.

***** ERROR *** Context does not parse!**

Whilst the command was probably a valid CGP command there was some syntax error in the set of graphs that followed it. Unfortunately, the current parser does not give any indication of where the error has occured.

***** ERROR *** - no such option!**

The help system does not recognise the chosen option.

***** ERROR *** lattice relation not added - universal unknown:**
<lattice relation>

issued when the knowledge base maintenance system is unable to add a new lattice relation to either lattice because the supertype is not already known and therefore the system cannot decide to which lattice the relation belongs.

help not available within a script

When CGP is run on a PC, the help system is not invoked when an error occurs within a command script.

I do not understand!

The interface cannot handle the input.

***** Warning *** Deductive Mechanism Disabled!**

This message is issued when the I command has been used.

***** Warning *** Theorem Prover Disabled!**

This message is issued when the p command has been used.

***** Warning *** Got Tired**

This message is issued when the timeout limit has been reached and a proof has aborted.

Messages Issued By System Commands

Most of these are self explanatory and will not be repeated here. The only exception is:

CGP terminated - all data lost if not previously saved!

This message is issued at the end of a session following the use of the ^ command.

APPENDIX II - PEIRCE LOGIC

This appendix starts by listing some of the main logical relationships that can exist between entities. A symbolic form is used to aid readability, thus symbols such as *p* and *q* represent whole graphs (which may be simple graphs or negated contexts), and symbols such as *p'* and *q'* represent specialisations of graphs such that *p* will project onto *p'*.

Logical Forms

Name	Standard Notation	Peirce Form
TRUE	*TRUE*	*( )*
FALSE	*FALSE*	*( )*
conjunction (AND)	$p \wedge q$	*p q*
negation (NOT)	$\neg p$	*(p)*
disjunction (OR)	$p \vee q$	*((p) (q))*
exclusive OR (XOR)	$p \vee q \wedge \neg(p \wedge q)$	*((p) (q)) (p q)*
neither of (NOR)	$\neg p \wedge \neg q$	*(p) (q)*
not both (NAND)	$\neg(p \wedge q)$	*(p q)*
implication	$p \Rightarrow q$	*(p (q))*
definition	$p \Leftrightarrow q$	*(p (q)) (q (p))*

Peirce Logic Operations

The following is a list of the more important operations that CGP uses and with which the user should be familiar in order to understand the workings of CGP.

Name	Symbolic Form
removal of double negations	$(\,(\,p\,)\,) \rightarrow p$
deiteration	$(\,p'\,p\,) \rightarrow (\,p'\,)$ $(\,p'\,(\,p\,)\,) \rightarrow (\,p'\,(\,)\,)$
iteration	$p\,(\,q\,) \rightarrow p\,(\,p\,q\,)$
individuals	$(p'$ --$(-$ $q\,)\,) \rightarrow (\,p'\,(\,q'\,)\,)$
elimination of disjunctions	$(\,(\,)\,p\,) \rightarrow (\,(\,)\,)$
restriction	p --- $q \rightarrow p$ --- p *(if p < q)*

APPENDIX III - INTERNAL REPRESENTATION

The following description of the internal representation is given for those who need to alter data contained within an existing knowledge base which has been saved with the > command. A complete knowledge base contains the following items:

* semantic net,
* rules,
* type and relational lattice,
* conformity relations table,
* canon,
* lexicon,
* alias tree,
* lambda abstractions,
* next available internal markers,
*

Taking each in turn, the role will be described along with the Prolog clauses used to encode them.

1 Semantic Net

The semantic net consists of all the simple graphs that the system contains. The detailed representation of graphs is given in section 12 below.

2 Rules

A 'rule' is any negated context whatever. Again, the details of their representation in Prolog are given in section 12 below.

3 Type And Relational Lattice

The lattice is stored as separate Prolog clauses of the form:

lattice(*subtype*,*supertype*).

where *subtype* and *supertype* are both internal markers (sections 10 and 12.5) representing the highest level in the alias tree (section 7) for the type. Each type must have such a relation present within the lattice for each of its supertypes up to the universal type.

4 Conformity Relations Table

This is stored in Prolog relations of the form:

conf(*type*,*individual_marker*).

where *type* is an internal marker representing the highest level in the alias tree for the type. There must only be one such conformity relation for each individual, and this must be a conformity relation to the most specific type to which the individual conforms.

The *individual_marker* is the integer that uniquely identifies the individual within the system and is not the whole individual object (see section 12.5) except in the case where the conformity relation is of a prototype to a type in which case the whole prototype object is used..

5 Canon

The canon is composed of Prolog clauses of the form:

canon(*label*,*upper_bound*,*lower_bound*).

where *label* is the marker for the relational label concerned, *upper_bound* is a simple graph, consisting of a single relation, in the form described in section 12 and which gives the upper bound for the relation, and

lower_bound is the graph expressing the lower bound for the relation.

6 Lexicon

Each lexicon entry has the form:

lex(*word_list*,*marker*).

The *word_list* is a Prolog list of words that have ben used within the model description. If there is more than one word in the word list then each word in this list must be a synonym for the same entity in the model. The *marker* is the individual marker that was assigned to the words by the system.

When a new label is encountered a lex entry is created:

lex([*new_label*],*marker*).

7 Alias Tree

The alias tree is made up of clauses of the form:

alias(*marker*,*marker_list*).

The *marker* is an individual marker that was assigned by the system when it was proved that two other markers represented the same individual within the model. The *marker_list* is a list which contains these other two markers.

In order to prime the system, each newly created marker is allocated an alias relation of the form:

alias(*new_marker*,[*new_marker*]).

8 Lambda Abstractions

Lambda abstractions are generated by the canonicity checking mechanism and the parser and take either of the following forms:

lambda([*marker*],*parameter_list*,*context*).

lambda([*type1_list*,*type2_list*,*marker*],*parameter_list*,*context*).

If the lambda abstraction is a type definition then the first form is used. If the lambda abstraction is a relational definition the second form is used and *type1_list* and *type2_list* are the type labels of the concepts that form the single relation graph within the definition. A third for is also possible in which the *parameter_list* is empty. In this case the lambda abstraction is used to associate an arbitrary graph with an individual marker. The *marker* is an individual marker and it is this marker that is used to refer to the label being defined. The *parameter_list* is a list of the parameters of the lambda expression and is used to indicate genus concepts. The *context* is the actual definition and can contain any arbitrary graphs or negated contexts. All of the formal parameters within *parameter_list* must also occur somewhere within *context*.

9 Episodic Memory

The system maintains a list of literal statements made to it during the existence of the current knowledge base. These statements are time-stamped (currently with a serial number derived from emarker) and are used for rebuilding the model in the light of new data that contradict the current model. An episode has the form:

episode(*marker*,*context*).

Each graph entered with the ! command and each graph given as a definition is recorded; queries are not currently recorded.

10 Next Available Internal Markers

It is necessary to store the next available integer for four kinds of entity within the system. These entities are:

* generic markers,
* individual markers,
* prototype markers,
* episode markers.

The next generic marker is stored in a clause of the form:

vmarker(*marker*).

The next individual marker is stored in a clause of the form:

imarker(*marker*).

The next prototype marker is stored in a clause of the form:

pmarker(*marker*).

The next episode marker is stored in a clause of the form:

emarker(*marker*).

The next generic context is stored in a clause of the form:

gmarker(*marker*).

11 Internal Representation Of Graphs And Negated Contexts

11.1 Negated Contexts

Negated contexts are represented by Prolog clauses of the form:

neg(*list*).

where *list* is a Prolog list containing arbitrary simple graphs and further negated contexts.

11.2 Simple Graphs

Simple graphs are represented by Prolog clauses of the form:

graph(*list*).

where *list* is either a Prolog list of conceptual relations or a list containing a single concept. If *list* is a list of conceptual relations then the graph that they represent must be connected and generally well-formed.

11.3 Conceptual Relations

A conceptual relation is represented by a Prolog clause of the form:

rel(*marker_list*,*concept_in_list*,*concept_out_list*).

where *marker_list* is a Prolog list of CGP objects, *concept_in_list* is the list of concepts which are joined to the relation by the arrow which points towards the relational node and *concept_out_list* is the list of concepts which are joined to the relation by the arrow which points away from the relational node.

11.4 Concepts

Concepts are represented by Prolog clauses of the form:

cons(*type*,*referent*).

where *type* and *referent* are Prolog lists of objects.

11.5 Objects

A CGP object is any marker that represents either an individual or a variable. There are currently three types of object. These are:

generic object,

individual object,

prototype object,

measure object,

context object.

The generic object is used to represent all generic variables and has the form:

*(*marker*).

where *marker* is an integer that uniquely identifies the object. It acts as the line of identity between concepts and relations.

The individual object is used to represent all individuals that are existentially quantified within the system and has the form:

#(*marker*).

where *marker* is an integer that uniquely identifies the object.

The prototype object is used to represent all typical individuals that exist within the system and has the form:

&(*marker*).

where *marker* is an integer that uniquely identifies the object.

The measure object is used to represent value referents where it is necessary for the system to be able to compare different values. Each has the general form:

@comp(value).

where *comp* is an optional operator from the set { < <= <> >= > }. The = operator is the default adopted when *comp* is absent. The parameter *value* has one of the following forms:

number.

string.

m(value,value).

d(value,value).

The context object holds either nested graphs or a generic object marker:

~([graph,...,graph]).

or

~((marker)).*

11.6 Lines Of Identity

Lines of Identity are stored within graphs as stated in section 11.5. Additionally, for efficiency reasons, each line of identity that originates within a graph at depth 0 has a record of its marker stored within a Prolog clause of the form:

head(marker)

where *marker* is the actual marker used for the line of identity and does not include the * and parentheses. Only lines of identity that originate in graphs at depth 0 are recorded in this manner and the record is only made when such graphs are added to the knowledge base.

12 Factors And Scales

Factors and scales are used in the quantity processing mechanism to provide a means of adding and comparing different types of value along a scale. Factors convert a value in one unit system to an equivalent value in another and scales give the actual possible values, lowest first, for a unit. Factors have the form:

factor(quantity,quantity).

where *quantity* has the form:

m(number,unit).

or

d(number,unit).

Scales have the form:

range(list).

where *list* is a list of values.

13 Frames

A *frame* is a template for a fixed field record which can be chosen and filled in with appropriate values. Each frame is stored as:

frame(type,param,graph).

where *type* is the marker for the type to which the frame corresponds, *param* gives the coreference link to the concept of type *type* in the frame to which the frame corresponds and *graph* is the frame.

APPENDIX IV - KNOWN PROBLEMS

The current system is under constant development and as a consequence there are a number of small problems that may be encountered. These relate to areas of the program that are supplementary to its main purpose and therefore have a low priority. Some of these are listed here:

1. The explanation mechanism will be unreliable in brief mode because this mode was originally only added rather quickly to allow CGP to run on a PC. It is therefore somewhat crude and, depending upon the route taken by the proof, may not always give any explanation at all, or may get the indentation and numbering wrong.

2. The time taken for evaluations is unpredictable because the route taken by the system is not necessarily the best. Part of the research is the optimisation of this aspect of the system and much has been accomplished, but there are areas where the efficiency is still less than it ought to be. Any evaluations that are found to take a long time should be recorded and reported to the author.

3. As the theory of the system is still incomplete there will inevitably be situations where a query that appears to be provable is found to be UNKNOWN. These should be reported to the author. There should not be any cases where CGP gives FALSE instead of TRUE or *vice versa*, but if this arises then there is definitely a problem.

4. The use of the measure referent ought to be restricted to non-dimensional and one dimensional values as the current mathematics module is at best useless and at worst worse than useless.

As problems arise and are solved the supplement to this user guide will be updated.

INDEX

A

actors 2.3
adding new relations 4.3.3
addinging new types to the lattices 4.3.1
adding subtypes of known types 4.3.2
adding simple knowledge 4.1
AND 2.5.1, 2.5.2
antecedent 4.5.1, 4.5.2
Aristotelian definitions 4.3.5
arity 2.2, 2.6.4
associativity in measure referents 2.3.6
assymetrical relations 4.6.7

B

backward chaining 4.6.2
batch processing 2.8.2
braces, use of 2.5.1
brief mode 3.15
built-in relation 2.9, 2.10
built-in type 2.9, 2.10

C

canon 2.6.4
canonicity checking 4.3
canonicity checking, disabling 3.5
canonical graph 2.6.4, 4.3.4
built-in 2.10, 2.10.3
CGP commands 3
\ command 3.4
^ command 3.9
! command 3.11
$ command 3.10
% command 3.5
& command 3.1
* command 3.6, 4.9
+ command 3.8
- command 3.7
. command 3.3
< command 3.1
> command 3.2
? command 3.12
@ command 3.30
b command 3.28
c Command 3.18
d command 3.25
f command 3.21
g command 3.22
i command 3.17
l command 3.24
m command 3.29
n command 3.23
o command 3.19
p command 3.16
s command 3.14
t command 3.26
u command 3.27

v command 3.15
z command 3.20
end_of_file command 4.3
CGP design principles 1.1
CGP messages appendix I
CGP objects 2.3
CGP operational modes 4.10
CGP world model 1.3
chaining, backward 4.6.2
chaining, forward 4.6.2
chaining of rules 4.6.2
check with user mode 3.18, 4.10.5
commands, CGP see CGP commands
commands, host system 3.3, 3.8
comments in scripts 3.4
common subtype 4.6.1
comparison operator 2.3.6
complex graph 2.5
compound graph 2.5
concepts 2.1
conceptual graph syntax and semantics 2
configuration 3.19
conformity relations 2.6.2
conformity relations, adding 4.9.2
consequent 4.5.1, 4.5.2
context 2.3.7, 2.5
context boundary 2.3.2
contexts, negated 2.5.2
contexts, nested 2.5.2
contexts, positive 2.5.1
coreference marker 2.3.1
coreferent generic object - *var 2.3.2

D

deduction 4.5, 4.6, 4.8
deduction requiring a chain of rules 4.6.2
deduction requiring combinations of rules 4.6.6
deduction requiring several facts 4.6.1
deduction requiring use of conditional rules 4.6.5
deduction with recursive use of a rule 4.6.3
deduction, Prolog_like 3.16
defaults 1.3.3
definitions 2.6.3
definition inconsistencies 4.3.5
deiteration 4.5.1, 4.5.2, 4.6.4, 4.6.6, 4.7.2
denotation 1.3.2
differentiae 4.3.2, 4.3.3
disjunction 2.5.2, 4.7.3
disjunction, elimination of 4.6.6, 4.7.3
double implication 2.6.3.1
double negation 4.5.1, 4.6.1, 4.6.6, 4.7.1
double negation, removal 4.7.1

E

EITHER_OR 4.7.3
elimination of disjunctions 4.6.6, 4.7.3
empty clause 4.7
empty context 4.7
end_of_file command 4.3
entity_relation theory 2.2

equality symbol - = 2.3.8
evaluation of data 4.1
exceptions 4.3.5
existential quantifier 2.1
explanations, nesting of 4.2

F

FALSE 1.3.1, 3.12
false facts 4.4
false set 1.3.3
file handling 4.11.1
findall mode 3.13, 4.10.3
formatting of graphs 2.7
forward chaining 4.6.2
full evaluation mode 3.16 4.10.4

G

general information 1
generic context 2.3.8
generic coreferent object - *var 2.3.2
generic object - * 2.3.1
graphs, canonical 2.6.4
graphs, nested 2.4

H

heuristic 4.7
hierarchy, relational 2.6.3.2, 4.6.7
hierarchy, type 4.6.7
higher order reasoning 4.6.7
host system commands 3.3, 3.8

I

IF_THEN rules 4.5
implication 2.5.2, 4.5.1, 4.7.4
INCONSISTENT 1.3.1, 3.12
individual object - #marker 2.3.3
inheritance of properties 4.3.5
interaction with computer 1.1
interaction with CGP 3.13, 4.10.3
interface 4.9
interface, evaluation of new graphs by 4.9.1
interface, inferences made by 4.9.2
internal representation appendix III
introduction 1.1
iteration 3.17, 4.6.6, 4.7, 4.7.4, 4.7.5
iteration mode 3.17, 4.10.6

J

joining of graphs 4.9.1

K

knowledge adding 3.11
knowledge deleting 3.7
knowledge base, editing 3.10, 3.11
knowledge base, initial state 4

knowledge base, keyword searches of 3.6
knowledge base, maintenance system 4.8
knowledge base, perusing 3.6
knowledge base, rebuilding 3.5
knowledge base, reloading a saved 3.1
knowledge base, saving the current 3.2
knowledge base, specialisation of graphs within 4.9.2
known problems appendix IV

L

label, built-in type 2.9, 2.10.1
lambda expressions 2.5.4, 2.6.3.1, 2.6.3.2
lexicon 2.6, 2.6.5
line of identity 2.3.1
logical primitive 2.1, 2.3.8, 2.5.1, 2.5.2
AND 2.5.1, 2.5.2
NOT 2.5.2
equality 2.3.8
existential quantification 2.1
lower bound 2.6.4

M

maximal common subtypes 4.9.2
measure object - @ 2.3.6
measure object, associativity 2.3.6
measure object, order of precedence 2.3.6
metaknowledge 1.4, 2.9
metasyntax 2.9
modes of operation 4.10
check with user 3.18, 4.10.5
default values 4, see also separate modes
evaluation 3.16, 4.10.4
interaction 3.13, 4.10.3
iteration 3.17, 4.6.10
reports 3.15, 4.10.2
speed 3.14, 4.10.1
modus ponens 4.7.6, 4.5.1
modus tollens 4.6, 4.6.4, 4.7.6

N

name object - string 2.3.4
NAND 4.5.2
natural language 4.6
natural types 4.6.7
necessary and sufficient conditions 4.3.5
negated contexts 2.5.1, 2.5.2, 4.4
negated contexts, combining of compound 4.9.2
nested contexts 2.5.2
nested graphs 2.4
nested models 1.3.4
nesting of explanations 4.6.2, 4.6.3

O

object 2.1
object, & 2.3.5
object, * 2.3.1
object, *var 2.3.2

object, # 2.3.3
object, @ 2.3.6, 2.6.4
object, string 2.3.4
online help 1.2, 4
open world model 1.3, 4.9.1
open world evaluation game 1.1
operational modes 3.19, 4.10, see also modes of operation
option 3.19
other data 2.6
output, verbosity 3.15

P

partial evaluation mode 3.16, 4.10.4
partial order 4.6.7
PC users 4.11
Peirce logic 4.5, 4.6, 4.7, appendix II
positive context 2.5.1
possible worlds 4.9.1
projection 4.2
Prolog_like deduction 1.1, 3.16
prototype object - & 2.3.5
prototypes 1.3.3, 4.9.2

Q

quantification, universal 4.3.5
quantifier 2.1
quantifier, existential 2.1
quantifiers generalised 2.3.6
queries 3.12, 4
queries, simple 4.2
quick mode 3.14, 4.10.1
quitting CGP 3.3, 3.9
quitting the *%user* script 4.3

R

referent field 2.1
rel, built-in type 2.9, 2.10.1
relational definitions 2.6.3.2
relational hierarchy 2.6.1, 2.6.3.2, 4.6.7
relational lattice 2.6.1, 2.6.3.2, 4.6.7
built-in 2.10, 2.10.2
relational lattice, adding new relations 4.3.1, 4.3.2, 4.3.3, 4.9.2
relational definitions, adding 4.3.2
relational order 2.6.4
relations 2.2, 2.6.4
relations, assymetrical 4.6.7
relations, diadic 2.2
relations, non_reflexive 4.2
relations, symmetrical 2.2, 4.6.7
relations, usage 2.6.4
reports of reasoning 4.11.2
resolution 4.7.7
restriction 4.7.5
restrictions in CGP 2.7
roles 2.2, 4.6.7
role types 4.6.7
rules 1.1, 4.5.1
running CGP 1.1, 1.2

running CGP on a PC 1.2
running CGP on a Unix machine 1.2
running CGP on an Apple Macintosh 1.2

S

schemata 1.3.3
scripts 2.8, 2.8.2
scripts, comments in 3.4
scripts, running 3.5
selectional constraint checking 1.3.1
selectional constraint violations 4.3.4
semantic net 1.3, 1.3.3, 4.4
semicolon 2.5.3
separating graphs within contexts 2.5.3
simple graphs 2.4
simple rules 4.5.1
single solution mode 3.13, 4.10.3
slow mode 3.14, 4.10.1
solution, number 3.13, 3.14
some 2.3.4
some_... 2.3.4
speed of operation 3.14, 4.10.1
splitting of graphs 4.6.3
sublabel, built-in relation 2.9, 2.10.2
subrel, built-in relation 2.9, 2.10.2
suptype, built-in relation 2.9, 2.10.2
superlabel, built-in relation 2.9, 2.10.2
superrel, built-in relation 2.9, 2.10.2
supertype, built-in relation 2.9, 2.10.2
symmetrical relations 2.2, 4.6.7

T

tautology 4.7
terminting graphs 2.5.3, 2.5.4
text files 2.8, 2.8.1
theorem prover 1.1, 4.8
theorem proving 4.7
transitive closure 4.6.3
transitivity 4.6.3
TRUE 1.3.1, 3.12
true set 1.3.3
truth values 1.3.1, 3.12
type, built-in type 2.9, 2.10,1
type contraction 4.6.1
type definitions 2.6.3.1
type definitions, adding 4.3.2
type field 2.1
type hierarchy 2.6.1, 4.6.7
type lattice 2.6.1, 4.6.7
built-in 2.10, 2.10.1
type lattice, adding new types 4.3.1, 4.3.2, 4.9.2

U

universal_rel, built-in reltation 2.9, 2.10.2
universal_type, built-in type 2.9, 2.10.1
universal quantifier 4.3.5
UNKNOWN 1.3.1, 3.12
upper bound 2.6.4

using the program 4

V

variables 2.3.1
variables in relation field 4.2, 4.6.7
variables in type field 4.6.7
variables, use of 4.2
verbose mode 3.15, 4.10.2
verbosity of explanations 3.15, 4.10.2