Access-Limited Logic — A Language for Knowledge Representation[1]

James Crawford
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
jc@cs.utexas.edu

October 2, 1990

[1]This work has taken place in the Qualitative Reasoning Group at the Artificial Intelligence Laboratory, The University of Texas at Austin. Research of the Qualitative Reasoning Group is supported in part by NSF grants IRI-8602665, IRI-8905494, and IRI-8904454, by NASA grants NAG 2-507 and NAG 9-200, and by the Texas Advanced Research Program under grant no. 003658–175.

Abstract

Access-Limited Logic (ALL) is a language for knowledge representation which formalizes the access limitations inherent in a network structured knowledge-base. Where a deductive method such as resolution would retrieve all assertions that satisfy a given pattern, an access-limited logic retrieves all assertions reachable by following an available access path. The time complexity of inference is thus a polynomial function of the size of the accessible portion of the knowledge-base, rather than the size of the entire knowledge-base. Access-Limited Logic, though incomplete, still has a well defined semantics and a weakened form of completeness, Socratic Completeness, which guarantees that for any query which is a logical consequence of the knowledge-base, there exists a series of queries after which the original query will succeed. We have implemented ALL in Lisp and it has been used to build several non-trivial systems, including versions of Qualitative Process Theory and Pearl's probability networks. ALL is a step toward providing the properties – clean semantics, efficient inference, expressive power – which will be necessary to build large, effective knowledge bases.

Contents

1 Introduction **6**

1.1 Background and Significance — Building Large Knowledge-Bases 7
1.2 Overview of Access-Limited Logic 8
1.3 Organization 10

I Algernon — Implementing Access Limited Logic **11**

2 Algernon **12**

2.1 Overview 12
2.2 Introduction to Frame Based Knowledge Representation 13
2.2.1 Frames 13
2.2.2 Frames and Their Names 13
2.2.3 Frames and Predicate Calculus 14
2.2.4 Access Paths 14
2.2.5 Rules Are Implications 15
2.2.6 Inference in Algernon 15
2.3 The Syntax and Semantics of Algernon 16
2.3.1 Overview 16
2.3.2 Introduction to the Syntax and Semantics of Algernon 16
2.3.3 The Syntax of Algernon 19
2.3.4 The Semantics of Algernon 22
2.3.5 Inference in Algernon 29
2.4 Overview of the Built in Knowledge-Base 30
2.5 Availability of Code 33

3 Techniques for Using Algernon **34**

3.1 Notation and Conventions 34
3.2 Using Access Paths 35
3.3 Simple Inference 43
3.4 Representing a Simple Discourse 45
3.5 Reasoning by Contradiction 48
3.6 Reasoning with Disjunctions Using *Modus Tolens* 57
3.7 Simple Reasoning About Sets of Similar Objects 63

3.8 More Complex Reasoning About Sets of Similar Objects . . . 66
3.9 Controlled Non-monotonicity . . . 71
3.10 Building a Finite State Machine . . . 72
3.11 A Generate and Test Cycle . . . 75
3.12 Reasoning with Different Views — Approaching Default Reasoning . . . 82

4 Systems **89**
4.1 The Headache Protocol . . . 89
4.1.1 Controlling Inference using Set Membership . . . 90
4.1.2 Controlling Sequence using Access Path Order . . . 90
4.1.3 Boolean Ask Slots . . . 91
4.2 Belief Networks . . . 92
4.3 QPC . . . 92
4.3.1 The QPC Knowledge-Base . . . 93
4.3.2 The QPC Algorithm . . . 93
4.3.3 The QPC Knowledge-Base (again) . . . 101

II Formalizing Access Limited Logic **103**

5 Formal Development of ALL **104**
5.1 Basic Notation . . . 104
5.2 Formal Syntax of ALL . . . 105
5.2.1 Alphabets, Terms, and Propositions . . . 105
5.2.2 Access Paths . . . 106
5.2.3 Rules . . . 106
5.2.4 Knowledge-Bases . . . 107
5.2.5 Operations and Formulas . . . 107
5.3 Knowledge Theory . . . 107
5.3.1 Partitions . . . 108
5.3.2 Substitutions . . . 109
5.3.3 Manipulating Knowledge-bases . . . 110
5.3.4 The Domain and Range of All Operations . . . 111
5.3.5 The Values of ALL Operations — Motivation . . . 111
5.3.6 The Problem of If-Added Incompleteness . . . 112
5.3.7 The Problem of Recursive Rules . . . 115
5.3.8 Partitions (again) . . . 116
5.3.9 The Values of ALL Operations — Definitions . . . 116
5.3.10 Implementation Note . . . 119
5.3.11 Basic Results . . . 120
5.4 Soundness . . . 121
5.4.1 Mapping ALL to Predicate Calculus . . . 122
5.4.2 Soundness Theorem. . . . 122
5.5 Completeness . . . 123

5.5.1 Some Results From the Study of Logic Programming . . . 125
5.5.2 Mapping ALL Knowledge-Bases to Logic Programs . . . 126
5.5.3 Socratic Completeness . . . 126
5.5.4 Partitional Completeness . . . 128

6 Adding Negation to ALL **130**
6.1 Two Examples . . . 130
6.2 Formal Development . . . 132
6.2.1 Syntax of ALL(+neg) . . . 132
6.2.2 Knowledge Theory . . . 134
6.2.3 Soundness . . . 139
6.2.4 Completeness . . . 140

7 Time Complexity of ALL **151**

8 Conclusion **158**
8.1 Related Work . . . 158
8.1.1 Knowledge Representation Systems . . . 158
8.1.2 Related Formalisms . . . 160
8.2 Discussion of Results . . . 161
8.2.1 About Socratic Completeness . . . 162
8.2.2 About Partitional Completeness . . . 162
8.2.3 About the Complexity Results . . . 163
8.3 Future Work — Expressing Full First Order Quantification . . . 163
8.4 Contributions . . . 164

A Three Systems Built Using Algernon **165**
A.1 The Headache Protocol . . . 165
A.2 Belief Networks . . . 176
A.3 QPC . . . 181

B Proofs of Technical Lemmas **207**
B.1 A Dialect of Predicate Calculus . . . 207
B.2 Proof of Soundness Theorem . . . 208
B.2.1 Basic Lemmas About $\mathcal{PC}$ . . . 209
B.2.2 Proof of Soundness of ALL . . . 210
B.3 Proofs of Lemmas for Completeness Results . . . 211
B.3.1 Preliminary Results . . . 211
B.3.2 Proofs of Lemmas . . . 213
B.4 Proofs of Lemmas for Complexity Result . . . 215
B.4.1 Basic Definitions and Lemmas . . . 215
B.4.2 Preliminary Lemmas . . . 219
B.4.3 Proof of Computation Lemma . . . 225

[In the *Theaeteus*] Plato saw that merely possessing knowledge (like birds in an aviary) is not enough; one must be able to command what one possesses. To perform well, one must be able to get the right bit of knowledge to fly to the edge at the right time (in real time, as the engineers say). But he underestimated the difficulty of this trick and hence underestimated the sort of theory one would have to give of the organization of knowledge in order to explain our bird-charming talents. Neither Plato nor any subsequent philosopher, so far as I can see, saw this as in itself a deep problem of epistemology, since the demands of efficiency and robustness paled into invisibility when compared with the philosophical demand for certainty, but so it has emerged in the hands of AI.

Daniel C. Dennett in [Graubard, 88]

Chapter 1

Introduction

It has long been a guiding principle of the semantic network approach to knowledge representation that by exploiting the network structure of knowledge, one can increase the efficiency of reasoning. Intuitively, in a semantic network related concepts are located "close together" in the network and thus search and inference can be guided by the structure of the knowledge-base. However, formalisms for semantic networks have generally treated the semantic network notation as a variant of predicate calculus, and have regarded the access limitations inherent in a network to be an extra-logical indexing mechanisms.

Access-Limited Logic (ALL) is based on two claims: first, that there exists a set of inference mechanisms, efficiently computable using a network structured representation, which are sufficient for a large class of common-sense reasoning problems, and second, that these inference mechanisms are Socratically Complete, thus guaranteeing that any logical consequence of a knowledge-base can be inferred after some series of 'leading' questions. Time complexity and Socratic Completeness are formal properties which will be proven in part two. The claim that the inference mechanisms in ALL are sufficient for a large class of common-sense reasoning problems can necessarily only be proven or disproven empirically. Part one presents work on the Lisp implementation of ALL and its application to a broad class of problems.

Reasoning is hard. If a knowledge representation language is as expressive as first-order predicate calculus, then the problem of deciding what an agent implicitly knows (*i.e.*, what an agent could logically deduce from its knowledge) is unsolvable [Boolos & Jeffrey]. Thus a sound and decidable knowledge representation and reasoning system must either give up expressive power, or use a weak inference system with an incomplete set of deduction rules or artificial resource limits (*e.g.*, bounds on the number of applications of *modus ponens*). However, such inference systems tend to be difficult to describe semantically and tend to place unnatural limits on an agent's reasoning ability [Levesque, 86].

Our solution to this dilemma is two fold. First, we have identified a set of inference mechanisms which can be efficiently computed in a network structured knowledge-base. The inference mechanisms in ALL are based on the idea of an *access path* and are guaranteed never to require searches of the entire knowledge-base. The time complexity of computing inference in ALL thus depends (polynomially) on the size of the *accessible* portion of the knowledge-base. Inference in ALL is thus necessarily incomplete. The second part of our approach is the development of formal results which characterize the (in)completeness of ALL. We have shown that ALL is Socratically Complete — for any query of a proposition which is a consequence (in predicate calculus) of the knowledge-base, there exists a series of preliminary queries after which the original query will succeed. We have also shown several technical properties which partially characterize the conditions under which a query will succeed immediately.

1.1 Background and Significance — Building Large Knowledge-Bases

One of the most basic lessons of artificial intelligence research has been that intelligence requires knowledge. Experience with expert system knowledge-bases, in particular, has demonstrated that adequate knowledge about a specific domain can provide an impressive degree of problem-solving power. At the same time, such a special-purpose system tends to be fragile and unreliable near the boundary of its knowledge, often suffering from a lack of "common sense" knowledge. (For example, an early medical expert system, when presented with the case of a patient with a bacterial infection, advised "Boil the patient." [G. J. Sussman, personal communication with B. Kuipers, 1973].) Responding to this problem, several research groups (*e.g.*, [Lenat et al., 86]) have proposed the creation of very large, general-purpose knowledge-bases of commonsense knowledge that could be shared by a wide range of different applications.

It has been apparent at least since the work of [Woods, 75] (and others such as [Hayes, 74]) that a knowledge representation language sufficient to support the building of large knowledge bases must have a clear semantics. Without a clear semantics one can never be sure exactly *what* a given expression represents, what deductions should follow from it, or how it compares to an expression in a different knowledge representation language. However, experience with formally specified knowledge representation systems has revealed an inevitable trade-off between the expressive power of a knowledge representation system and its computational complexity [Levesque, 86, Levesque & Brachman, 85]. Further, the trade-off is such that in order to ensure tractable and complete reasoning, one must restrict the expressiveness of a knowledge representation language almost to the point of unuseability (except within restricted domains [Allen, 83, Bundy et. al, 1985]). The most common solution to this problem has been to combine fast, special purpose inference with a first-order logic theorem prover [Brachman et al., 83, Schubert et al., 83] or a weaker deduction system [Levesque, 84, Patel-Schneider, 85, Vilain, 85]. The main problem with such systems is that though they answer some queries efficiently they mysterious fail on others (or never answer at all).

ALL represents a new approach to this problem. Inference in ALL is guided by the structure of the knowledge-base and its time complexity is polynomial in the size of the accessible portion of the knowledge-base. As such, ALL defines an efficient special purpose inference mechanism. However, Socratic Completeness obviates the need to combine ALL with a general purpose theorem prover. For any query which could be proven by a theorem prover, there exists a series of preliminary queries after which the query will succeed in ALL (thus one might want to combine ALL with a system for suggesting preliminary queries — see section 8.2.1). There are several advantages to the approach taken in ALL. First, all inference in ALL is guided by the structure of the knowledge-base (not just some special purpose inference). Second, ALL uses a single homogeneous inference mechanism and its behavior is thus more comprehensible (and more easily analyzed formally) than that of a composite system. Finally, any query or assertion in ALL is guaranteed to return in polynomial time.

1.2 Overview of Access-Limited Logic

Our approach in ALL begins with the well known mapping between atomic propositions in predicate calculus and slots in frames; the atomic proposition that the object a stands in relation r to the object b can be written logically as $r(a, b)$ or expressed, in frames, by including object b in the r slot of object a [Hayes, 79]:

$r(a, b) \quad \equiv$

```
a:
    r:
        values: { ...b... }
```

We refer to the pair $\langle a, r\rangle$ as a *frame-slot.* Thus $r(a, b)$ is equivalent to saying that the value b is in the frame-slot $\langle a, r\rangle$. The frames *directly accessible* from a frame-slot are those which appear in the frame-slot.[1] Extending this idea, we define an *access path*, in a network of frames, as a sequence of frames such that each is directly accessible from a frame-slot of its predecessor. It is useful to generalize this definition and allow access paths to branch on all values in the frame-slots. A sequence of propositions defines an access path through a network if any variable appearing as the first argument to a proposition has appeared previously in the sequence (operationally, this means that retrieval always accesses a known frame-slot). For example, "John's parent's sister" can be expressed in ALL as the path:

$$(parent(John, x), sister(x, y))$$

This defines an access path from the frame for $John$ to the frames for $John$'s parents (found by looking in the frame-slot $\langle John, parent\rangle$), to $John$'s parents' sisters.[2]

From access paths we build the inference rules of ALL. A rule is always associated with a particular slot in the network. Backward chaining *if-needed* rules are written in the form: $\beta \leftarrow \alpha$ (the structure of α and β is discussed in section 5.2.3) and applied when a value for the slot is needed. Forward chaining *if-added* rules are written in the form: $\alpha \rightarrow \beta$ and applied when a new value for the slot is inserted. In either case the antecedent of a rule must define an access path. For example, using the access path above we can write the if-needed rule:

$$aunt(John, y) \leftarrow parent(John, x), sister(x, y)$$

But we *cannot* write the (logically equivalent) rule:

$$aunt(John, y) \leftarrow sister(x, y), parent(John, x),$$

[1] Slots in ALL contain only frames and rules (defined below).

[2] Note that we use the term access-path ambiguously in that it refers to both the sequence of propositions and the sequence of frames it defines.

since the antecedent does not define an access path.[3]

Where a classical deductive method or logic programming language would retrieve all known assertions that satisfy a given pattern, an access-limited logic retrieves all assertions reachable by following an available access path. The use of access paths alone, however, is insufficient to guarantee computational tractability in very large knowledge-bases. The evaluation of a path can cause an explosive back-chaining of rules which can spread throughout the knowledge-base. To prevent this, ALL introduces a second form of access limitation. The knowledge-base in ALL is divided up into *partitions* and back-chaining is not allowed across partitions — facts in other partitions are simply retrieved. When used together, these two kinds of access limitations limit the complexity of inference to a polynomial function of the size of the accessible portion of the knowledge-base.

A price must be paid for the efficiency of access limitations. Inference in ALL is weaker than inference in predicate calculus, since only locally accessible facts and rules can be used in deductions. However, logical coherence does not necessarily require completeness. Rather, coherence is an informally defined collection of desirable formal properties. We have proven that a dialect of ALL has the following properties of a logically coherent knowledge representation system:

- ALL has a well defined syntax and proof theory.
- The semantics of ALL can be defined by a purely syntactic mapping of ALL knowledge-bases, queries and assertions to predicate calculus.
- In terms of this mapping, inference in ALL is sound, Socratically Complete, and Partitionally Complete.

Socratic Completeness guarantees that any fact which is logically implied by a knowledge-base can be deduced after a suitable series of preliminary 'leading' queries. Partitional Completeness guarantees that certain 'obvious' deductions (deductions provable using only locally accessible backward chaining rules) can be made immediately (*i.e.*, without any preliminary queries). These properties are stated more precisely as theorems in part two below.

The current formalism of ALL has expressive power equivalent to predicate calculus without existential quantification (the current implementation has considerably more expressive power). Extending the logic to allow existential quantification, but not *mixed* quantification, should be straightforward.

Predicate calculus without existential quantification is at least as expressive as propositional calculus. Inference in ALL in therefore NP hard. The Socratic Completeness and polynomial time complexity of ALL are thus simultaneously possible only because the problem of finding the right series of preliminary queries is NP hard. If we were to extend the expressive power of ALL to allow mixed nested quantification then ALL would have the expressive power of predicate calculus, and inference in ALL would be undecidable. The problems involved in allowing mixed nested quantification (and preserving tractability and Socratic Completeness) are discussed in chapter eight.

[3]The restriction to access paths limits the syntax of ALL, but is not a fundamental limit on its expressive power since one could always add a new constant and make it the first argument to every predicate. This would amount to making the entire knowledge-base a single frame.

1.3 Organization

This dissertation is divided into two main parts. Part one presents, Algernon, the Lisp implementation of ALL. Part two presents the formal development of ALL. The parts do not depend on each other and may be read in either order.

Part one comprises chapters two through four. Chapter two introduces Algernon, while chapter three shows a series of examples of the application of Algernon to problems in common-sense reasoning. Chapter four presents three non-trivial systems which have been built using Algernon.

Part two consists of chapters five through eight. Chapter five develops ALL for a restricted language (with representative power approximately that of a deductive data base). Chapter six shows how to introduce negation. Chapter seven proves the time complexity of ALL. Chapters five through seven build on each other and should be read sequentially. Chapter eight concludes with a discussion of related work and future plans.

Part I

Algernon — Implementing Access Limited Logic

Chapter 2

Algernon

Algernon is a knowledge representation system based on the ideas of Access-Limited Logic. There are several motivations for implementing ALL (as opposed to just theorizing about it):

1. Several of the claims underlying ALL are empirical in nature and can only be proven or disproven by working with examples. Two such claims are the claim that common-sense rules of inference can be written using access paths (see section 2.2.4), and the claim that a large knowledge-base can be cleanly organized into partitions (2.3.5).

2. Throughout our work on ALL the implementation has driven the formalism (while the formalism has guided the implementation). We generally begin with interesting examples (such as the bacteria example in section 3.7), code in Algernon, extract the generally useful formal theory (in this case arbitrary objects, see section 8.3), and then revise the implementation in light of the theory.

3. Algernon provides a useful 'laboratory' for studying problems and techniques in knowledge representation. Several such problems and techniques are discussed in chapter 3.

4. Algernon has proven useful for the implementation of non-trivial systems. Three of these systems are discussed in chapter 4.

2.1 Overview

This chapter is organized into three sections. Section 2.2 is an introduction to frame based knowledge representation in general and Algernon in particular.[1] Section 2.3 presents the syntax and semantics of Algernon. Section 2.4 overviews the background knowledge-base "built into" Algernon.

For information on the actual use of Algernon (how to run it, the commands it understands, its output, etc.) see the Algernon user's guide [Crawford].

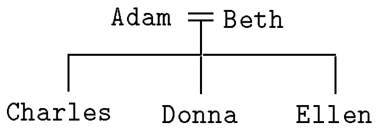
[1]Parts of section 2.2 are derived from class handouts prepared by Ben Kuipers.

2.2 Introduction to Frame Based Knowledge Representation

2.2.1 Frames

Frames are descriptions of conceptual individuals. Frames can exist for "real" objects such as "The Watergate Hotel", sets of objects such as "Hotels", or more "abstract" objects such as "Cola-Wars" or "Watergate". Frames are essentially defined by their relationships with other frames.[2] Relationships between frames are represented using *slots*. If a frame `f` is in a relationship `r` to a frame `g`, then we put the *value* `g` in the `r` *slot* of `f`.

For example, suppose we are describing the following genealogical tree:



The frame describing Adam might look something like:

```
Adam:
   sex:     Male
   spouse:  Beth
   child:   (Charlie Donna Ellen)
```

Where `sex`, `spouse`, and `child` are slots. Note that a single slot may hold several values (*e.g.*, the children of Adam).

The genealogical tree would then be described by (at least) seven frames, describing the following individuals: Adam, Beth, Charlie, Donna, Ellen, Male, and Female.

2.2.2 Frames and Their Names

Actually, this is not quite right. For clarity of presentation, we have been allowing a certain amount of confusion between the thing *itself*, and its name. Really, the frame for `Adam` would look something like:

```
frame-27:
   name:    ((Adam))
   sex:     (frame-2)
   spouse:  (frame-28)
   child:   (frame-29 frame-30 frame-31)
```

(Where the `name` slot of the frame holds a list of Lisp atoms used to refer to the frame.) However, since using the name instead of the generated atom `frame-27` is much clearer, we will continue to do so. One needs to remember that the machine only knows the internal form.[3]

[2] An exception to this rule are frames which are somehow "grounded" in sensory experience. A discussion of symbol grounding is far beyond the scope of this dissertation, but a suggestion of how it can be done is seen in the definition of places in [Kuipers and Byun, 1988].

[3] The code in Algernon contains some heuristics for generating the "internal form" such that in many cases it is identical to the name of the frame. However, one should regard this as a happy coincidence when it happens and should not depend on it.

2.2.3 Frames and Predicate Calculus

A frame can be considered just a convenient way to represent a set of predicates applied to constant symbols (*e.g.*, *ground instances* of predicates.). For example, the frame above could be written:

```
sex(Adam,Male)
spouse(Adam,Beth)
child(Adam,Charlie)
child(Adam,Donna)
child(Adam,Ellen)
```

More generally, as discussed in the introduction, the ground predicate $r(f, g)$ is represented, in a frame based system, by placing the value `g` in the `r` slot of the frame `f` [Hayes, 79]:

$$r(f,g) \quad \equiv \quad \boxed{\begin{array}{l}\texttt{f:}\\ \qquad\texttt{r:}\\ \qquad\qquad\texttt{values: }\{\ \ldots\texttt{g}\ldots\ \}\end{array}}$$

Throughout this chapter we use the teletype font (*e.g.*, `John`) for Algernon statements and the math font (*e.g.*, $John$) for predicate calculus. We use standard predicate calculus notation except that we find it convenient to use $\rightarrow$ in addition to the usual $\leftarrow$.

2.2.4 Access Paths

One advantage of a frame based representation is that the (conceptual) objects related to a frame can be easily accessed by looking in a slot of the frame (there is no need, for example, to search the entire knowledge-base). As discussed in the introduction, we define an *access path*, in a network of frames, as a sequence of frames each directly accessible from (*i.e.*, appearing in a slot of) its predecessor. A sequence of predicates defines an access path iff any variable appearing as the first argument to a predicate has appeared previously in the sequence. Recall that "John's parent's sister" can be expressed in Algernon as the path:

```
((parent John ?x) (sister ?x ?y))
```

(variables are always denoted in Algernon by Lisp atoms whose print names begin with '`?`'). The Algernon path `((parent John ?x) (sister ?x ?y))` is equivalent to the predicate calculus statement:

$$parent(John, ?x) \wedge sister(?x, ?y).$$

In predicate calculus this statement is equivalent to

$$sister(?x, ?y) \wedge parent(John, ?x).$$

However, the corresponding sequence of predicates:

```
((sister ?x ?y) (parent John ?x))
```

is *not* an access path because a query of `(sister ?x ?y)` would require a search of the knowledge-base.

2.2.5 Rules Are Implications

We represent logical implications as rules. For example, the logical assertion that

$$\forall x, y.[spouse(x, y) \rightarrow spouse(y, x)]$$

can be represented by the *forward-chaining rule*:

```
((spouse ?x ?y) -> (spouse ?y ?x))
```

Intuitively, such a rule says that whenever we learn a relation `(spouse f g)` we should immediately conclude `(spouse g f)`. Such *forward-chaining* ("if-added") rules are generally used to maintain invariants in the knowledge-base.

Algernon also allows *backward-chaining* ("if-needed") rules. For example,

```
((aunt John ?y) <- (parent John ?x) (sister ?x ?y))
```

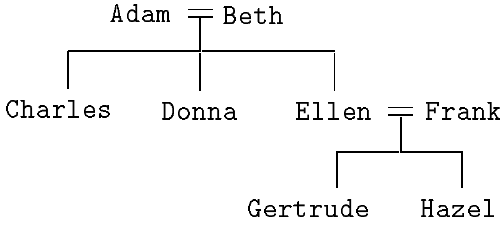
Intuitively, this rule says that if you need to find an aunt of `John` then you should look for a sister of a parent of `John`. Notice that the antecedent of this rule is an access path. All rules in Algernon must define access paths. Thus the implication:

```
((aunt John ?y) <- (sister ?x ?y) (parent John ?x))
```

is not allowed since the only way to "fire" this rule would be to globally search for all frames which are in a `sister` relation to other frames.

2.2.6 Inference in Algernon

There are two basic operations on a knowledge-base — *queries* and *assertions*. Queries generally retrieve knowledge from the knowledge-base, applying if-needed rules, while assertions generally add new facts to the knowledge-base and apply if-added rules. Of course, a successful if-needed rule will assert its consequent into the knowledge-base, and the antecedent of an if-added rule may have additional terms to be queried, so the two operations are not disjoint. Rules in Algernon are tied into the frames of the knowledge-base by being associated either with frames or with slots. To understand how this works, consider a knowledge-base of frames and rules representing the following genealogy:



Assume further that we have a frame `People` (representing the set of all people) and that the relation `isa`[4] has been asserted between the frames for the people in the genealogy and the frame `People` (*e.g.*, `(isa Adam People)`, `(isa Beth People)`, ...). Assume further that the rule:

```
((aunt ?z ?y) <- (parent ?z ?x) (sister ?x ?y))
```

is *associated* with the frame *People*. Formally this means:

$$(\forall z, x, y.(isa(z, People) \rightarrow [aunt(z, y) \leftarrow parent(z, x) \wedge sister(x, y)]))$$

What this means operationally is that any query to the `aunt` slot of a frame that is an `isa` relation to the frame `People` will result in the application of this rule. Rules are applied by first querying their antecedent and then, if the antecedent succeeds, asserting their consequents. For example, if we query `(aunt Hazel ?x)` then the rule will apply, the access path will be followed (from `Hazel` to `Ellen` to `Donna`) and Algernon will conclude (and assert) `(aunt Hazel Donna)`. However, if we query `(aunt Charlie ?x)` then the rule will fire, but the access path will "dead-end" (at `Adam` and `Beth`). Algernon will thus find no aunts of `Charlie` and the query is said to *fail*.

Rules may also (less commonly) be associated with slots or sets of slots. If an if-added rule is associated with a slot `r`, or with a set of which `r` is a member, then it is applied whenever a value is added to the `r` slot of *any* frame (if-needed rules associated with slots and sets of slots are similar). For example, one might want to put the slot `less` in the set of partial orders, and thus inherit rules, associated with the set of partial orders, for transitivity, reflexivity, and antisymmetry.

2.3 The Syntax and Semantics of Algernon

2.3.1 Overview

This section begins with an introduction to the syntax and semantics of Algernon. The next two subsections give the details of the syntax and semantics of Algernon, respectively. Section 2.3.5 discusses the inference algorithm used in Algernon. Finally, section 2.4 overviews the knowledge-base 'built into' Algernon.

2.3.2 Introduction to the Syntax and Semantics of Algernon

This subsection describes the syntax of Algernon in sufficient depth that the examples can be understood. Those interested in more detail should pay more attention to the subsequent subsections, while those uninterested in details can proceed from the end of this subsection directly to section 2.4.

The simplest type of assertion would be something like:

```
assert ((father Charles Adam))
```

Where `Charles` and `Adam` are names of frames in the knowledge-base and `father` is a slot in the knowledge-base. This assertion would add the value `Adam` to the `father` slot of the frame `Charles` (and apply any if-added rules for this slot — see section 2.3.5). The corresponding query:

[4] Intuitively, the relation `isa` holds between an object and a set of which it is a member.

```
query ((father Charles Adam))
```

would succeed iff `Adam` is in (or could be proven to be in) the `father` slot of `Charles`. Queries (and assertions) return all known (or provable) values of variables. Thus the query:

```
query ((father Charles ?x))
```

would retrieve all values which are in (or could be proven to be in) the `father` slot of `Charles`. Further the query:

```
query ((father Charles ?x) (sister ?x ?s))
```

would find all sisters of fathers (*i.e.*, aunts) of `Charles`. Note that access paths are queried by querying the first predicate and then, branching on all sets of variable bindings, querying the rest of the path. If a point is ever reached where there are no possible sets of variable bindings then the query of the path fails, and the rest of the path is not queried at all (*e.g.*, if the query of `(father Charles ?x)` fails to find any possible bindings for `?x`, then `(sister ?x ?s)` cannot be queried).

More complex assertions are similar. The assertion:

```
assert ((father Charles ?x) (sister ?x ?s) (likes ?s chocolate))
```

branches on all known fathers of Charles and all sisters of fathers of Charles, and asserts that they like chocolate. The corresponding query would return all Charles' father's sisters who like chocolate.

Variable use in Algernon is restricted to prevent searches of the entire knowledge-base. *Whenever an unbound variable occurs it must always be possible to bind the variable by simply retrieving values from a known slot of a known frame.* Thus the query:

```
query ((sister ?x ?s) (father Charles ?x) (likes ?s chocolate))
```

is illegal since it does not define an access path and there is thus no way to bind the variable `?x` (without a search of the entire knowledge-base). These restrictions apply to variable use in queries, assertions, and rules (see section 2.3.3).

Algernon slots can hold 'non-values' as well as values. Non-values are used to express negation; they say that an object does not stand in some relation to some other object. Non-values are denoted by `(not p)`, where `p` is a proposition. Thus:

```
assert ((not (likes suzan chocolate)))
```

asserts that suzan does not like chocolate (technically, it adds `chocolate` to the non-value facet of the `likes` slot of the frame with name `suzan`).

Algernon also supports several special predicates (or special forms). Short descriptions of the most important special predicates are given below. The exact syntax of all special predicates is given in section 2.3.3. Their semantics is more fully discussed in section 2.3.4.

`:taxonomy` Adds to the basic taxonomic structure in the knowledge-base. The taxonomic structure of sets forms the "backbone" of an Algernon knowledge-base. Thus this is often the first form used in Algernon applications. Sets are described by lists whose elements are (names of) the members of the sets and whose sublists are subsets of the set. For example, in the "flu" example (see section 3.3), we set up a taxonomy in which `Diseases` and `Symptoms` are subsets of `Objects`, `Flu` is a disease, and `Fever` and `Nausea` are symptoms:

```
(:taxonomy (Objects (Diseases Flu) (Symptoms Fever Nausea)))
```

`:slot` Declares a new slot. For example:

```
(:slot has-disease (people diseases))
```

declares the slot `has-disease` to be a relation between (members of the set of) people and (members of the set of) diseases. Several optional keyword descriptors may be added to the basic `:slot` form. These are discussed in section 2.3.4.

`:rules` and `:srules` Add deduction rules to the knowledge-base. `:rules` is followed by the name of the (frame for the) set the rules are to be associated with, and then the rules. `:srules` is similar except that it associates rules with a slot instead of a frame. For example,

```
(:rules people ((aunt ?x ?a) <- (parent ?x ?p) (sister ?p ?a)))
```

associates the back-chaining rule for aunt with the frame `People` (the set of all people). Similarly,

```
(:srules (:slot spouse) ((spouse ?x ?y) -> (spouse ?y ?x)))
```

associates the forward-chaining rule with the slot `spouse` (the construction `(:slot spouse)` is needed to refer to the *slot* `spouse` instead of the *frame* with name `spouse`).

`:forc` Find OR Create. `:forc` finds a frame satisfying a description, or creates a new frame and asserts the description. For example, `(:forc ?x (father Tom ?x))` binds `?x` to the father of Tom, if the father of Tom is known, otherwise it creates a new frame, binds `?x` to it, and asserts that it is the father of Tom.

`:the` Implements a simple approach to definite descriptions. As with `:forc`, its first argument is a variable and its second argument is a path. Also as for `:forc`, a new frame is created when a query of the path fails. The difference between `:the` and `:forc` is that if *multiple* bindings for the variable are found then the definite description fails (where the `:forc` would succeed). For example, in the genealogy in section 2.2.6, if `(:the ?x (brother Donna ?x))` appeared in a path, then it would succeed and bind `?x` to `Charlie`. However, `(:the ?x (sister Charlie ?x))` would fail because there are two known sisters of Charlie. There is an intended (though as yet imperfect) analogy between the English "a" and `:forc`, and between "the" and `:the`.

`:ask` Queries the user directly. Fully described in section 2.3.4.

A "functional" shorthand for Algernon paths is provided by the Algernon preprocessor and is used extensively in the examples. Instead of writing `((r1 f ?y) (r2 ?y ?z) (r3 ?z g))`, the functional shorthand allows one to write `((r3 (r2 (r1 f)) g))`. For example, the rule for `aunt` given in section 2.2.5 could have been written:

```
((aunt John ?y) <- (sister (parent John) ?y))
```

Such a rule is expanded by the preprocessor and kept internally in the longer form.

A final wrinkle in the syntax of Algernon is the special treatment of the slot `name`. For any slot other than `name`, a query of form `(slot ?x value)` would violate the variable use restrictions discussed above. However, a query of `(name ?x value)` will bind `?x` to all frames with name `value`. This is done to provide an easy way to access frames by name.

2.3.3 The Syntax of Algernon

Grammar

The following is the formal syntax of Algernon. (If you are just getting started you might find section 2.3.4 more useful).

Notation:

$\mathbf{x}^*$ denotes zero or more repetitions of x.

$\mathbf{x}^+$ denotes one or more repetitions of x.

Also, the teletype font, `font`, is used for terminals (*i.e.*, strings which actually occur in Algernon code).

path = (form$^+$)

rule = (form$^+$ `<-` form$^+$) |
(form$^+$ `->` form$^+$)

form = predicate |

(`:rules` term rule$^+$) |
(`:srules` term rule$^+$) |
(`:del-rule` term rule) |
(`:del-srule` term rule) |
(`:del-rules` term slot) |
(`:del-srules` term slot) |

(`:contra-positive` form$^+$) |
(`:wo-contra-positive` form$^+$) |
(`:in-own-partition` pred) |

(`:the` variable form$^+$) |
(`:the` (variable$^+$) form$^+$) |

```
        (:forc variable form+) |
        (:forc (variable+) form+) |
        (:unp form+) |
        (:all-paths path path) |

        (:assume predicate) |
        (:retrieve predicate) |
        (:ask predicate) |

        (:create variable atom*) |
        (:show term) |

        (:delete predicate) |
        (:clear-slot term slot) |

        (:decl-slots (slot number)+) |
        (:slot atom (term+) descriptor*) |

        (:lisp expression) |
        (:bind variable expression) |
        (:bind-to-values variable term slot) |
        (:branch-on-values variable expression) |
        (:apply function expression) |
        (:neq term term)

        (:taxonomy set-descriptor)

predicate = (slot term term+) | (not predicate)
term = variable |
        frame |
        (slot term) |
        (:slot slot) |
        (:quote expression)

descriptor = :cardinality number |
             :partition term |
             :backlink slot |
             :inverse slot |
             :comment string

set-descriptor = (term set-descriptor* term*)
```

frame = Any Lisp atom which is a name of a frame in the knowledge-base.
slot = Any Lisp atom declared as a slot in the knowledge-base.

variable = Any Lisp atom whose print name begins with a '?'.

number = Any number.
string = Any Lisp string.
expression = Any Lisp expression.
atom = Any Lisp atom.
function = Any Lisp function (in a form suitable to be passed to the Lisp function `apply`).

Variable Use Restrictions

In general, variables can only be used where they can be instantiated by retrieving a value from a known slot of a known frame. In paths (in queries and assertions) this means that variables can only occur as the first argument to a predicate if they have previously appeared in the path (and thus are now bound). For example:

```
((parent John ?x) (sister ?x ?s))
```

is a legal path, while:

```
((parent ?y ?x) (sister ?x ?s))
```

is not.

In rules the situation is slightly more complex. Consider first if-added rules. If-added rules are always matched against fully instantiated predicates (since they are fired when known new values are added to known slots of known frames). Thus, in the antecedent of an if-added rule, one can depend on all variables in the first predicate being instantiated. Thus one can write the if-added rule:

```
((spouse ?x ?y) (friend ?y ?z) -> (friend ?x ?z))
```

If-needed rules, on the other hand, are matched against queries. Thus only the frame and slot are necessarily known. Thus, in the antecedent of an if-needed rule, one can depend only on the first variable in the first predicate being instantiated. Thus one can write the if-needed rule:

```
((friend ?x ?z) <- (spouse ?x ?y) (friend ?y ?z))
```

But one *cannot* write the if-needed rule:

```
((spouse ?x ?y) <- (spouse ?y ?x))
```

Though one can write this as an if-added rule:

```
((spouse ?x ?y) -> (spouse ?y ?x))
```

This is one reason why if-added rules are most often used to maintain the invariants in the knowledge-base.

2.3.4 The Semantics of Algernon

Predicates

The simplest possible path in Algernon is a path of form:

```
((slot frame value))
```

An assertion of such a path simply adds the value `value` to the slot `slot` of the frame `frame`. Similarly a query of such a path will succeed iff `value` is in (or can be proven to be in) the `slot` slot of `frame`.

A slightly more complex path would be one using the functional shorthand. For example, to refer to a "brother of a friend of a sister of Tom" one could say:

```
((brother (friend (sister Tom)) ?x))
```

Such a path is expanded by the preprocessor to something like:

```
((sister Tom ?$x1) (friend ?$x1 ?$x2) (brother ?$x2 ?x))
```

In general:

$$\texttt{((r (}f_1\texttt{ (}f_2 \ldots \texttt{ (}f_n\texttt{ A) } \ldots \texttt{)) ?x))}$$

is equivalent to:

$$\texttt{((}f_n\texttt{ A ?}x_1\texttt{) (}f_{n-1}\texttt{ ?}x_1\texttt{ ?}x_2\texttt{) } \ldots \texttt{ (}f_{n-m}\texttt{ ?}x_m\texttt{ ?}x_{m+1}\texttt{) } \ldots \texttt{ (}f_1\texttt{ ?}x_{n-1}\texttt{ ?}x_n\texttt{) (r ?}x_n\texttt{ ?x))}$$

The functional shorthand is currently only supported for binary relations (and thus for unary functions).

Frames and Their Names (again)

As discussed in the introduction (section 2.2.2), we generally refer to frames using their names — the value in their `name` slot. Thus an assertion of:

```
((slot frame value))
```

actually places a pointer to the frame with name `value` into the `slot` slot of the frame with name `frame`. At times one wishes to "override" this mechanism — for example to put a Lisp atom directly in a slot of a frame. To do this one uses the `:quote` form. For example, in some robotics applications one might want to store a list of movement commands (these might be the commands necessary to navigate from one place to another [Kuipers and Byun, 1988]):

```
(commands path (:quote (turn-right straight turn-left)))
```

where `turn-right` is the name of a low level function (not an Algernon frame).

At other times, one may wish to refer to a slot itself. This is done with the `:slot` form. For example, one might wish to comment a slot. For example:

```
(comment (:slot isa) "(isa ?x ?s) = ?x is a member of the set ?s.")
```

This comments the *slot* `isa` (as opposed to the frame with name `isa` — this distinction is meant to parallel the Lisp distinction between the atom `'car` and the function `#'car`).

One interesting special case is numbers. It is fairly silly to create a frame for every number. On the other hand it is clumsy to have to always say `(:quote 23)`. Our current solution is to consider a number to be the name of itself (even though no frame actually exists). Thus an assertion of `(temperature John 23)` places (the Lisp number) `23` in the `temperature` slot of the frame (with name) `John` (there is no attempt to find a frame with name `23`). Again this is an attempt to parallel Lisp in which numbers evaluate to themselves. Strings are handled similarly.

More Complex Forms

Algernon supports a number of "special forms". This section provides a description of these forms and examples of their use. With the exception of `:retrieve`, all forms have the same effect in queries and assertions.

(`:rules` *term rule*$^+$)

(`:srules` *term rule*$^+$)

These two forms add rules, associated with the frame referred to by *term*, to the knowledge-base. The only difference between them is the way in which the rules are indexed (and therefore in the way they will later be accessed). Consider a query of `(r frame1 ?x)` (assertions are similar). Such a query will access rules from several sources:

1. It will access *slot* rules (rules added using `:srules`) associated with the slot `r`.
2. It will access 'normal' rules (rules added using `:rules`) associated with any set which `frame1` is a member of (*i.e.*, rules associated with any frame `f` such that `(isa frame1 f)` is a fact in the knowledge-base).
3. Finally, it will access slot rules associated with any set which `r` is known to be a member of.

As an example of a set of slots, consider the set `transitive-relations`. For this set one might want a rule like:

```
(:SRULES transitive-relations
 ((?r ?x ?z) <- (?r ?x ?y) (?r ?y ?z)))
```

One could then assert that `(isa (:slot less) transitive-relations)` and the rule above would enable Algernon to conclude `(less frame1 frame3)` from `(less frame1 frame2)` and `(less frame2 frame3)`. (This rule for transitivity is too in-efficient for use in a large knowledge-base. A more efficient variant is used in the background knowledge-base described in section 2.4.)

Note that this rule for transitive relations has a variable in the *slot* position. Such rules are called *generic* rules and can be associated with sets of frames or slots (though obviously not with individual slots). Such rules pose no computational difficulties for Algernon (since the slot being queried or asserted is always known), and should be considered to be shorthand for repeating the rule once for each slot.

(`:del-rule` *term rule*)

(`:del-srule` *term rule*)
Used to delete a rule (or slot rule) from a frame.

(`:del-rules` *term slot*)

(`:del-srules` *term slot*)
Used to delete all rules (or slot rules) from a slot of a frame.

(`:w-contra-positive` *form*$^+$)

(`:wo-contra-positive` *form*$^+$)
These forms are used to enable and disable the addition to the knowledge-base of the contra-positives of rules. The default is *not* to add contra-positives. Contra-positives are used to minimize proofs by contradiction when reasoning with disjunctions (an example of the use of disjunction is given in section 3.6).

(`:in-own-partition` *pred*)
Provides a primitive mechanism for temporarily changing the active partitions. In general, the active partitions are set by the query or assertion being performed. However, there are cases in which it is clear when writing a rule that a predicate in the antecedent of the rule should be queried in its own partition (usually this is because the predicate is in an obscure partition, but it is important for the query of the predicate to succeed). `(:in-own-partition pred)` temporarily switches to the partitions of `pred` and queries (or asserts) `pred`.[5] An example of the use of :in-own-partition can be found on page 64.

(`:forc` *variable form*$^+$)

(`:forc` (*variable*$^+$) *form*$^+$)
`:forc` implements find or create retrieval. `:forc` first queries the path *form*$^+$. If this query succeeds then the `:forc` succeeds (binding the variable *variable*$^+$). If the path fails then a new frame is created, *variable* is bound to it, and the path is asserted. (`:forc` (*variable*$^+$) *form*$^+$) is similar, but allows multiple variables.

One use of `:forc` is to guarantee that certain slots always hold some value. For example, one might express "Every car has a steering wheel." as:

```
(:rules cars
 ((steering-wheel ?c ?w) <- (:forc ?w (steering-wheel ?c ?w))))
```

Care must be taken with such rules, however, as they can easily cause infinite loops (*e.g.*, "Every man has a father") — see section 8.3.

(`:the` variable form$^+$)

[5]Technical note: The partitions of `pred` are updated before `pred` is queried (or asserted), and the active partitions are updated after the operation on `pred` (see section 2.3.5).

(`:the` (variable$^+$) form$^+$)

`:the` is just like `:forc` except that it fails if the query of its path returns multiple bindings for its variables.

(`:create` *variable atom**)

Make a new frame (with name *atom**, or a system chosen name if *atom** is `nil`) and binds *variable* to it. `:create` differs from `:forc` and `:the` (which can also create new frames) in that it always creates a new frame. For example, `(:create ?x Tom)` will create a new frame with name `Tom` and bind `?x` to it.

(`:unp` *form*$^+$)

Unprovable. Succeeds exactly when a query of its path (*i.e.*, *form*$^+$) fails. `:unp` is used primarily in default rules. For example:

```
(:rules Birds
 ((flies ?x True) <- (:unp (not (flies ?x True)))
                     (:assume (normal ?x Birds flies))))
```

(`:assume` is discussed below). `:unp` is a non-monotonic form and to use it successfully you will probably have to understand something of the internals of how Algernon does inference (see section 2.3.5).

(`:all-paths` *path1 path2*)

`:all-paths` first queries *path1*. It then queries *path2* under every substitution generated by the first query and succeeds iff all of these queries succeed. For example, in the genealogy in section 2.2.6, one could query "Are all Adam's children male" as:

```
((:all-paths ((child Adam ?x)) ((gender ?x male))))
```

Note that, like `:unp`, `:all-paths` is non-monotonic.

(`:assume` *predicate*)

Adds *predicate* to the knowledge-base as an assumption. Assumptions differ from facts in two ways. First, an attempt is made to prove the negation of the predicate, and if this attempt succeeds, the `:assume` fails. Second, any future conclusion which depends on the assumption is tagged with the assumption, so that if the assumption is later withdrawn the conclusion is also withdrawn.

(`:retrieve` *predicate*)

`:retrieve` suppresses the usual application of deduction rules while querying its predicate. It should be used only in queries.

(`:ask` *predicate*)

Asks user for a value for *predicate*. If *predicate* is ground then Algernon simply asks the user if *predicate* is true. If the user answers `yes` then the predicate is asserted and the `:ask` succeeds. If the user answers `no` then Algernon concludes the negation of the predicate and the `:ask` fails. If the predicate is not ground then Algernon asks for a value for the variable in the predicate. If the slot of the predicate is typed to hold values from a set, and the members of the set are known, then Algernon requires the user to enter a value in the set.

In applications making extensive use of `:ask` it may be useful to define a set of "ask-slots" using something like the following path:

```
((:taxonomy (slots (ask-slots)))
 (:srules ask-slots
  ((?r ?x ?y) <- (:ask (?r ?x ?y)))))
```

One could then define the slot `age` as an ask slot using the path:

```
((:slot age (things nil))
 (isa (:slot age) ask-slots))
```

This says that the slot `age` can appear in any frame which is a member of the set `things` (but the elements in the slot are not typed — this allows us to put numbers in the slot without having to create a frame for each number), and that `age` is an ask-slot. The effect of this path will be that if an `age` relation is queried, and no value is known, then the user will be queried for a value.

(`:show` *term*)
: Shows the contents of the frame referred to by *term*.

(`:delete` *predicate*)
: Removes *predicate* from the knowledge-base. If *predicate* is of form `(slot frame value)` then this involves removing `value` from the slot `slot` of `frame` (negated predicates are similar). Like `:unp`, `:delete` is non-monotonic and should be used with care.

(`:clear-slot` *term slot*)
: Removes *all* values from slot *slot* of the frame referred to by *term*. Like `:unp`, `:clear-slot` is non-monotonic and should be used with care.

(`:decl-slots` (*slot number*)$^+$)
: `:decl-slot` is the Algernon primitive for defining slots (`:slot` is actually a macro which expands to `:decl-slots`). For each pair (*slot number*), `:decl-slot` declares *slot* to be a slot holding a most *number* values. If *number* is `nil` then no restriction is put on the number of values the slot may hold.

(`:slot` *atom* (*term*$^+$) *descriptor**)
: Declares a new slot and types it using *term*$^+$. Typing is enforced using an if-added rule. For example:

```
(:slot has-disease (people diseases))
```

declares the slot `has-disease` to be a relation between people and diseases. If one then asserts:

```
(has-disease p1 d1)
```

Algernon concludes that `p1` isa `people`, and `d1` isa `diseases`.

`:slot` takes the following "keyword" arguments:

`:cardinality` *n* Asserts that the slot holds at most *n* values. Defaults to `nil` (no restriction). In the current implementation, cardinality information is primarily used to limit the application of if-needed rules (if a slot is queried but is full then the rules are not applied).

`:partition` *partition* Places the slot in partition *partition* (see section 2.3.5). Defaults to `main-partition`.

`:backlink` *slot2* Backlinks the slot to *slot2*. This means that if Algernon learns `(slot f1 f2)` it will conclude (via an if-added rule generated by the backlink) `(`*slot2* `f2 f1)`.

`:inverse` *slot2* Asserts that the inverse of the slot is *slot2*. This is equivalent to bi-directional backlinks.

`:comment` Adds a comment for the slot.

`:slot` is a macro. The form:

```
(:slot foo (set1 set2)
       :cardinality n
       :partition p
       :backlink b
       :inverse i
       :comment "Comment for foo")
```

is equivalent to:

```
((:decl-slot (foo n))
(type-slot (:slot foo) (set1 set2))
 (slot-partition (:slot foo) p)
 (backlink (:slot foo) b)
 (inverse (:slot foo) i)
 (comment (:slot foo) "Comment for foo"))
```

`(:lisp` *expression*`)`

Evaluates *expression* as a Lisp expression and succeeds iff it does not evaluate to `nil`. Note that (as with all Algernon special forms) if `:lisp` appears in a path then variable substitution is done before the form is evaluated. Thus one can print out the children of `Tom` using the path:

```
((child Tom ?x) (:lisp (format t "A child of Tom is ~a.~%" '?x))).
```

`(:bind` *variable expression*`)`

Binds *variable* to the result of evaluating *expression* (as a Lisp expression). As explained with `:lisp` above, if `:bind` appears in a path then variables substitution is done before the form is evaluated.

(`:bind-to-values` *variable term slot*)

Binds *variable* to a list of the values in *slot* of *term*. There are (at least) two uses of this form:

1. When using Algernon and Lisp it often turns out to be useful to be able to pass to a Lisp function a list of the objects in an Algernon frame-slot:

   ```
   ((:bind-to-values ?x frame slot) (:lisp (foo '?x)))
   ```

2. One can determine the number of values in a frame-slot using:

   ```
   ((:bind-to-values ?x frame slot) (:bind ?n (length '?x)))
   ```

(`:branch-on-values` *variable expression*)

Essentially the inverse of `:bind-to-values`. Like `:bind`, `:branch-on-values` evaluates *expression* as a Lisp expression. Unlike `:bind` it expects the result to be a list and, instead of binding *variable* to the resultant list, branches on the elements of the list, with each branch binding *variable* to an element of the list. Thus, within a path, `:branch-on-values` maps the rest of the path over the values in the list returned by *expression*. For example, assume that we have a lisp function, `simulate`, which given a 'state' returns a list of possible next 'states'. Further, assume that we have appropriately defined the frame `initial-state`, and the slots `description` (which holds a lisp readable description of a state), and `next-state`. We could then use the following path to link `initial-state` to its possible successors:

```
(description initial-state ?s) (:branch-on-values ?ns-desc (simulate
'?s)) (:create ?ns) (next-state initial-state ?ns) (description ?ns
(:quote ?ns-desc))
```

(`:apply` *function expression*)

Applies the Lisp function *function* to *expression*. As explained with `:lisp` above, if `:apply` appears in a path then variable substitution is done before the form is evaluated.

(`:neq` *term1 term2*)

Succeeds iff NOT *term1* = *term2*. `:neq` is actually a macro which expands to `(:lisp (not (eql '`*term1* `'`*term2*`)))`.

(`:taxonomy` *set-descriptor*)

Adds to the basic taxonomic structure of the knowledge-base. Sets are described by lists whose atomic elements are (names of) the members of the sets and whose sublists are subsets of the set. The set forming the root of the taxonomy (`set1` in the example below) most already exist.

`:taxonomy` is also a macro. The form:

```
(:taxonomy (set1
             (set2 frame1 frame2)
             (set3 frame3)))
```

expands to:

```
((:forc ?set2 (name ?set2 (:quote (set2))))
 (imp-superset ?set2 set1)

 (:forc ?frame1 (name ?frame1 (:quote (frame1))))
 (member ?set2 ?frame1)
 (:forc ?frame2 (name ?frame2 (:quote (frame2))))
 (member ?set2 ?frame2)

 (:forc ?set3 (name ?set3 (:quote (set3))))
 (imp-superset ?set3 set1)

 (:forc ?frame3 (name ?frame3 (:quote (frame3))))
 (member ?set3 ?frame3))
```

2.3.5 Inference in Algernon

Partitions

At a high level, processing a query (or assertion) consists of applying all accessible rules (see the discussion of `:rules` and `:srules` above). Applying any rule consists simply of querying its antecedent and then (if the query of the antecedent succeeds) asserting its consequent. One important complication to this simple picture is the idea of *partitions*.

Intuitively, a partition is a part of the knowledge-base which is somehow internally cohesive and distinct from the rest of the knowledge-base. Operationally, a partition limits rule chaining; rule chaining is always cut off at partitional boundaries. Assertions into other partitions are *queued* and their forward-chaining rules are not applied until their partition is *updated*. Partitions are currently updated just before operations are performed in them.

Rule Completion

In ALL, the application of an if-added rule is triggered by the assertion of a fact (into a slot from which the rule can be accessed) which matches the first predicate of the antecedent of the rule.[6] One potential problem with this approach is that, if one is not careful, it can be the case that accessible rules, whose antecedents are entailed by the knowledge-base, may never fire. Consider, for example, a knowledge-base in which `(r1 frame1 frame2)` is known and we have a rule:

```
((r1 ?x ?y) (r2 ?y ?z) -> (r3 ?x ?z))
```

If we then assert `(r2 frame2 frame3)`, this will not trigger the rule. We refer to this problem as *if-added incompleteness*. Algernon's solution to this problem is to *complete* the rule (when `(r1 frame1 frame2)` is added), by adding the shortened rule:

```
((r2 frame2 ?z) -> (r3 frame1 ?z))
```

[6] Unlike some expert system shells in which every rule is applied whenever a new fact is asserted.

This rule is added to the `selfset` of `frame2` (the selfset of a frame is the set consisting exactly of the frame). One might worry that rule completion would add a large number of rules to the knowledge-base and thus slow reasoning. However, since they are associated with selfsets, such rules are inaccessible to almost all operations.

One side effect of rule completion is that it requires one to be careful when removing facts from the knowledge-base (always a very dangerous operation). In the case above, if you later retract `(r1 frame1 frame2)`, using `(:clear-slot frame1 r1)` for example, then the rule completion is *not* undone and the shortened rule remains. This is a symptom of the larger problem that Algernon generally expects to reason monotonically and is not prepared to 'roll back' the assertion of a fact. If you want to be able to later retract a fact then you must enter it as an assumption (see section 2.3.4 above).

2.4 Overview of the Built in Knowledge-Base

The built-in knowledge-base sets up the top level of the taxonomy as well as several basic slots. The current built-in knowledge-base is simply a collection of sets, slots and rules which we have found useful in our examples, however, we hope to eventually expanded it into a true ontology for common-sense reasoning.

The following conventions have been used in naming frames and slots:

- Name for *sets* are generally plural (*e.g.*, `people`) to distinguish them from individuals.
- Names for slots are generally chosen so that `(r f1 f2)` can be read as "f1 r f2" or "An r of f1 is f2" (depending on whether `r` is a verb or a noun).

The taxonomy is set up as follows:

```
(:taxonomy (things
             (objects
               (sets things objects sets slots partitions
                     (partitions main-partition set-partition
                                 slot-info-partition partition-partition))
               (booleans true false)
               (physical-attributes
                 (colors)
                 (genders male female))
               (physical-objects
                 (people))
               (contexts global-context))
             (slots
               (order-relations
                 (tc-order-relations
                   (equivalence-relations))))))
```

Every frame in the knowledge-base is a member of the set `things` which breaks down into the sets `objects` and `slots`, and so on. In the knowledge-base, the taxonomy is represented using `isa` and `superset` relations (and their inverses `member` and `subset`). `isa` links an object to a set of which it is a member, and `superset` links a set to a superset of it (thus `member` links a set to one of its members, and `subset` links a set to one of its subsets).

A basic design decision in building large taxonomies is whether to link an object in the taxonomy to every set it is a member of, to link it to just the 'lowest' set in the taxonomy it is a member of (and then link this set up to the sets higher up in the taxonomy using `superset` links), or to link it to some of the sets it is a member of. We have chosen to take the third course, and link objects to just the "important" sets they are a member of. This distinction is represented using the relations `superset` and `imp-superset`. Intuitively, `imp-superset` links a set to an 'important' superset. Operationally this means that if Algernon learns that an object, `x` is a member of a set `s`, and `s` has an important superset `S`, then Algernon immediately adds an `isa` link between `x` and `S`.

The slot declarations for the important slots defined in the built-in knowledge-base are given below:

```
(:slot isa (things sets)
       :partition main-partition
       :comment "(isa ?x ?s) = ?x is a member of the set ?s.")

(:slot member (sets things)
       :partition set-partition
       :backlink isa
       :comment "(member ?s ?x) = A member of ?s is ?x.")

(:slot subset (sets sets)
       :partition set-partition
       :comment "(subset ?s1 ?s2) = A subset of ?s1 is ?s2.")

(:slot superset (sets sets)
       :partition set-partition
       :inverse subset
       :comment "(superset ?s1 ?s2) = A superset of ?s1 is ?s2.")

(:slot superset (sets sets)
       :partition set-partition
       :inverse subset
       :comment "(superset ?s1 ?s2) = A superset of ?s1 is ?s2.")

(:slot selfset (things sets)
       :cardinality 1
       :backlink member
```

```
       :comment "The selfset of x is the set consisting exactly of {x}.")


(:slot less (objects objects)
       :comment "(less ?x ?y) = ?x less than ?y.")

(:slot greater (objects objects)
       :inverse less
       :comment "(greater ?x ?y) = ?x greater than ?y.")

(:slot equal (objects objects)
       :inverse equal
       :comment "(equal ?x ?y) = ?x is equal to ?y.")

(:slot least (objects sets)
       :comment "(least ?x ?s) = ?x is the least member of ?s.")

(:slot greatest (objects sets)
       :comment "(greatest ?x ?s) = ?x is the greatest member of ?s.")


(:slot color (physical-objects colors)
       :cardinality 1
       :comment "(color x c) = The color of x is c.")

(:slot gender (physical-objects genders)
       :cardinality 1
       :comment "(gender x g) = The gender of x is g.")

(:slot temperature (physical-objects nil)
       :cardinality 1
       :comment "(temperature x temp) = The temperature of x is temp.")


(:slot spouse (people people)
       :cardinality 1
       :backlink spouse
       :comment "(spouse a b) = The spouse of a is b.")

(:slot wife (people people)
       :cardinality 1
```

```
        :backlink spouse
        :comment "(wife a b) = The wife of a is b.")

(:slot husband (people people)
        :cardinality 1
        :inverse wife
        :comment "(husband a b) = The husband of a is b.")

(:slot friend (people people)
        :comment "(friend a b) = A friend of a is b.")



(:slot current-context (contexts contexts)
        :cardinality 1
        :backlink super-context
        :comment "(current-context c1 c2) =
                 The current sub-context of c1 is c2.")

(:slot speaker (contexts people)
        :cardinality 1
        :comment "(speaker c s) = The speaker in c is s.")

(:slot recent (contexts objects)
        :comment "(recent c r) = A recently mentioned thing in c is r.")
```

2.5 Availability of Code

The code for Algernon should be available for research purposes from James Crawford by the first part of 1991.

Chapter 3

Techniques for Using Algernon

This chapter presents a collection of techniques for the use of Algernon to represent knowledge. The first few sections are intended to give the reader a feel for the use of Algernon, while the later sections tackle increasingly difficult problems in knowledge representation. The sections are largely independent and the reader is encouraged to read just those which interest them.

3.1 Notation and Conventions

As in the built-in knowledge-base, I use the following conventions for choosing names of frames and slots:

- Name for *sets* are generally plural (*e.g.*, people) to distinguish them from individuals.
- Names for slots are generally chosen so that `(r f1 f2)` can be read as "f1 r f2" or "A(n) r of f1 is f2" (depending on whether `r` is a verb or a noun).

The sections below contain actual (running) Algernon code. The Lisp interface to Algernon consists of two functions `a-assert` and `a-query`. The syntax is as follows:

```
(a-assert string path)
(a-query string path)
```

where `string` is a comment string, and `path` is an Algernon path (defined formally in section 2.3.3).

Some of the sections also include Algernon output. The first line in the output produced by a query or assertion will be something like:

```
ASSERTING: The John Alden story.
```

This is only a sort of comment or label for the operation. Next you should see either a message that the operation failed, or a list of the variable bindings produced (you may see some variables with names beginning with `?$`, such variables are created by the pre-processor in expanding some of the Algernon shorthand). Variable bindings are shown in the form:

```
?u     --- frame [name]
```

Where, 'frame' is frame itself (in the current implementation this is the Lisp atom on whose property list the frame data structure can be found), and 'name' is the value in the `name` slot of the frame (this is provided to increase readability in cases in which the frame is something like `frame26`). Then you will see a list of the new frames created (if any), and the following performance data:

Insertions: The number of new facts added to the knowledge-base.

Rule applications: The number of (forward and backward chaining) rules applied.

Iterations: The depth of rule chaining when applying rules breadth-first. Always zero when applying rules depth-first (the current default is depth-first).

Unifications: The number of unifications performed.

Matches: The number of matches performed. A match binds variables in an expression against a variable free pattern while a unification unifies two expressions containing variables. Matches are thus significantly faster than unifications (by a factor of 2 to 3 in Algernon).

Frame insertions: The number of values added to slots of frames. This is distinct from the number of new facts added to the knowledge-base because Algernon uses the frames for various kinds of internal bookkeeping.

Frame accesses: The number of retrievals of values from frames. Even simple operations often require a lot of frame accesses, but frame access are low level operation and are fast (about 5000 a second in the current Algernon implementation on a Macintosh IIci).

3.2 Using Access Paths

One of the most straightforward applications of access paths is reasoning about family relationships.[1] To represent family relationships, the only set we need is the set `people` defined in the built-in knowledge-base (see page 30). However, we need several new slots. As discussed in section 2.3.2, the form `:slot` defines a new slot. For example, the form:

```
(:slot parent (people people)
       :cardinality 2
       :inverse child
       :comment "(parent p1 p2) = A parent of p1 is p2")
```

defines a slot `parent` which is a relationship between two people (technically two frames which are members of the set `people`), has cardinality two (*i.e.*, any person can have up to two parents), and is the inverse of the relation `child`. The comment string is stored in the knowledge-base in the frame for the slot `parent`. Operationally, this definition marks the slot `parent` as having cardinality two, and adds three forward-chaining rules (one to type the slot, and two to make `parent` the inverse of `child`):

```
(parent ?x ?y) -> (isa ?x people), (isa ?y people)
(parent ?x ?y) -> (child ?y ?x)
(child ?y ?x) -> (parent ?x ?y)
```

[1]The ancestry of the example presented in this section can be traced back to an example originally written by Ben Kuipers (sometime back in the dark ages).

The full set of relations necessary to represent simple family relationships are defined by the following assertion:

```
(a-assert "Family relations."
   '((:slot child (people people)
            :comment "(child p1 p2) = A child of p1 is p2.")
     (:slot son (people people)
            :comment "(son p1 p2) = A son of p1 is p2")
     (:slot daughter (people people)
            :comment "(daughter p1 p2) = A daughter of p1 is p2")

     (:slot parent (people people)
            :cardinality 2
            :inverse child
            :comment "(parent p1 p2) = A parent of p1 is p2")
     (:slot father (people people)
            :cardinality 1
            :comment "(father p1 p2) = The father of p1 is p2")
     (:slot mother (people people)
            :cardinality 1
            :comment "(mother p1 p2) = The mother of p1 is p2")

     (:slot sibling (people people)
            :inverse sibling
            :comment "(sibling p1 p2) = A sibling of p1 is p2.")
     (:slot brother (people people)
            :comment "(brother p1 p2) = A brother of p1 is p2.")
     (:slot sister (people people)
            :comment "(sister p1 p2) = A sister of p1 is p2.")

     (:slot grandchild (people people)
            :comment "(grandchild p1 p2) = A grandchild of p1 is p2.")
     (:slot grandson (people people)
            :comment "(grandson p1 p2) = A grandson of p1 is p2")
     (:slot granddaughter (people people)
            :comment "(granddaughter p1 p2) = A granddaughter of p1 is p2")

     (:slot grandparent (people people)
            :cardinality 4
            :inverse grandchild
            :comment "(grandparent p1 p2) = A grandparent of p1 is p2")
     (:slot grandfather (people people)
            :cardinality 2
            :comment "(grandfather p1 p2) = A grandfather of p1 is p2")
     (:slot grandmother (people people)
            :cardinality 2
            :comment "(grandmother p1 p2) = A grandmother of p1 is p2")

     (:slot uncle (people people)
            :comment "(uncle p1 p2) = An uncle of p1 is p2.")
     (:slot aunt (people people)
            :comment "(aunt p1 p2) = An aunt of p1 is p2."))))
```

Several rules are then needed to express the meanings of these relations. First we have a set of fairly uninteresting rules which define simple invariants between sex specific relationships and generic relationships (*e.g.*, a father is a male parent):

```
(a-assert "Simple rules."
     ;;
     ;; Sex specific links are "shorthands" from simpler links:
     '((:rules people
         ((father ?x ?f) -> (parent ?x ?f) (gender ?f male))
         ((father ?x ?f) <- (parent ?x ?f) (gender ?f male))
         ;;
         ((mother ?x ?f) -> (parent ?x ?f) (gender ?f female))
         ((mother ?x ?f) <- (parent ?x ?f) (gender ?f female))
         ;;
         ((son ?x ?s) -> (child ?x ?s) (gender ?s male))
         ((son ?x ?s) <- (child ?x ?s) (gender ?s male))
         ;;
         ((daughter ?x ?d) -> (child ?x ?d) (gender ?d female))
         ((daughter ?x ?d) <- (child ?x ?d) (gender ?d female))
         ;;
         ((brother ?x ?b) -> (sibling ?x ?b) (gender ?b male))
         ((brother ?x ?b) <- (sibling ?x ?b) (gender ?b male))
         ;;
         ((sister ?x ?b) -> (sibling ?x ?b) (gender ?b female))
         ((sister ?x ?b) <- (sibling ?x ?b) (gender ?b female))
         ;;
         ((grandfather ?x ?gf) -> (grandparent ?x ?gf) (gender ?gf male))
         ((grandfather ?x ?gf) <- (grandparent ?x ?gf) (gender ?gf male))
         ;;
         ((grandmother ?x ?gf) -> (grandparent ?x ?gf) (gender ?gf female))
         ((grandmother ?x ?gf) <- (grandparent ?x ?gf) (gender ?gf female))
         ;;
         ((grandson ?x ?gs) -> (grandchild ?x ?gs) (gender ?gs male))
         ((grandson ?x ?gs) <- (grandchild ?x ?gs) (gender ?gs male))
         ;;
         ((granddaughter ?x ?gs) -> (grandchild ?x ?gs) (gender ?gs female))
         ((granddaughter ?x ?gs) <- (grandchild ?x ?gs) (gender ?gs female))
         ;;
         ;; Aunt and Uncle are a bit different (there is no unisex term):
         ((uncle ?x ?u) -> (gender ?u male))
         ((aunt ?x ?a) -> (gender ?a female)))))
```

Then we have rules which allow relations to be computed from other relations. Notice how these rules follow paths through the knowledge-base. One interesting complication here is that one can never be one's own brother — thus `(brother frame1 frame2)` always presupposes that `frame1` and `frame2` are not coreferent (*i.e.*, correspond to distinct individuals in the 'real' world). The default assumption that distinct frames are not coreferent is made within the rule for `sibling` below (`(:NEQ ?x ?y)` succeeds iff `?x` and `?y` are bound to distinct frames, and `(:ASSUME (not (coreferent ?x ?y)))` adds the assumption that the frames are not coreferent):

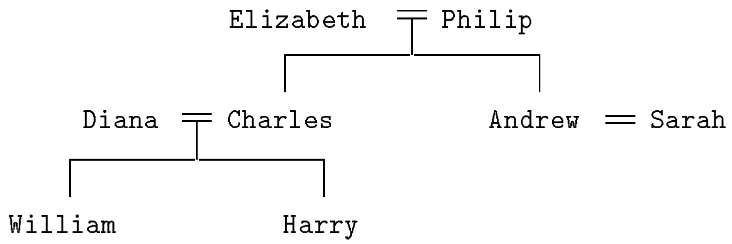


Figure 3.1: A family tree.

```
(a-assert "More complex rules."
     '((:rules people
         ((grandfather ?a ?c) <- (father (parent ?a) ?c))
         ((grandmother ?a ?c) <- (mother (parent ?a) ?c))
         ((grandson ?a ?c)    <- (son (child ?a) ?c))
         ((granddaughter ?a ?c)  <- (daughter (child ?a) ?c))
         ;;
         ((sibling ?x ?y)
         <-
         (child (parent ?x) ?y)
         (:NEQ ?x ?y)     ; This check is not logically necessary,
                          ; but is done for efficiency.
         (:ASSUME (not (coreferent ?x ?y))))
         ;;
         ((uncle ?x ?u) <- (brother (parent ?x) ?u))
         ((uncle ?x ?u) <- (husband (aunt ?x) ?u))
         ((aunt ?x ?a) <- (sister (parent ?x) ?a))
         ((aunt ?x ?a) <- (wife (uncle ?x) ?a))
         ;;
         ((husband ?w ?h)
         <-
         (gender ?w female) (spouse ?w ?h) (gender ?h male))
         ((wife ?h ?w)
         <-
         (gender ?h male) (spouse ?h ?w) (gender ?w female)))))
```

Then we create frames for a family, and assert their relationships. The family tree asserted is shown pictorially in figure 3.1.

```
(a-assert "A family."
    '((:create ?ch Charles)
      (:create ?di Diana)
      (:create ?ha Harry)
      (:create ?wi William)
      (:create ?ph Philip Mountbatten)
      (:create ?el Elizabeth II)
      (:create ?an Andrew)
```

```
(:create ?sa Sarah)
;;
(wife ?ch ?di)
(son ?ch ?ha)
(son ?ch ?wi)
(son ?di ?ha)
(son ?di ?wi)
(father ?ch ?ph)
(mother ?ch ?el)
(wife ?ph ?el)
(son ?ph ?an)
(son ?el ?an)
(wife ?an ?sa)))
```

Finally, we have a set of queries to test the knowledge-base. The form `:ALL-PATHS` (as discussed in section 2.3.4) succeeds iff every variable binding for the first path extends to a set of bindings for the second path. The Algernon output is shown following each query. For the first query only we include a partial trace of rule applications.

```
  (a-query "Who are William's grandfathers and grandmothers?"
    '((grandfather William ?gf)
      (grandmother William ?gm)))

QUERYING:   Who are William's grandfathers and grandmothers?

  ** Beginning logic Trace **
 Querying: (grandfather william ?gf).
   Updating partition: main-partition
   New preds: nil
   Applying: (grandfather william ?gf) <-
               (retrieve (coreferent william ?y))
               (Lisp (not (equal 'william '?y)))
               (retrieve (grandfather ?y ?gf)).
     Querying: (retrieve (coreferent william ?y)).
       Querying: (coreferent william ?y).
       Query succeeded: (coreferent william william).
     Query succeeded: (retrieve (coreferent william william)).
     Querying: (Lisp (not (equal 'william 'william))).
     Query failed.
   Rule failed.
   Applying: (grandfather william ?gf) <- (parent william ?$x1)
                                           (father ?$x1 ?gf).
     Querying: (parent william ?$x1).
     Query succeeded: (parent william diana) (parent william charles).
     Querying: (father diana ?gf).
       Applying: (father diana ?gf) <- (retrieve (coreferent diana ?y))
                                        (Lisp (not (equal 'diana '?y)))
                                        (retrieve (father ?y ?gf)).
         Querying: (retrieve (coreferent diana ?y)).
           Querying: (coreferent diana ?y).
           Query succeeded: (coreferent diana diana).
         Query succeeded: (retrieve (coreferent diana diana)).
         Querying: (Lisp (not (equal 'diana 'diana))).
```

```
    Query failed.
  Rule failed.
  Applying: (father diana ?gf) <- (parent diana ?gf) (gender ?gf male).
    Querying: (parent diana ?gf).
      Applying: (parent diana ?gf) <-
                  (retrieve (coreferent diana ?y))
                  (Lisp (not (equal 'diana '?y)))
                  (retrieve (parent ?y ?gf)).
        Querying: (retrieve (coreferent diana ?y)).
          Querying: (coreferent diana ?y).
          Query succeeded: (coreferent diana diana).
        Query succeeded: (retrieve (coreferent diana diana)).
        Querying: (Lisp (not (equal 'diana 'diana))).
        Query failed.
      Rule failed.
    Query failed.
  Rule failed.
Query failed.
Querying: (father charles ?gf).
Query succeeded: (father charles philip).
Asserting: (grandfather william philip).
  Inserting new value (grandfather william philip).
    Propagating addition of (grandfather william philip).
    Applying: (grandfather william philip) -> (isa william people)
                                              (isa philip people).
      Querying: (grandfather william philip).
      Query succeeded: (grandfather william philip).
      Asserting: (isa william people).
        Inserting old value (isa william people).
      Assert succeeded: (isa william people).
      Asserting: (isa philip people).
        Inserting old value (isa philip people).
      Assert succeeded: (isa philip people).
    Rule applied.
    Applying: (grandfather william philip) -> (grandparent william philip)
                                              (gender philip male).
      Querying: (grandfather william philip).
      Query succeeded: (grandfather william philip).
      Asserting: (grandparent william philip).
        Inserting new value (grandparent william philip).
          Propagating addition of (grandparent william philip).
          Applying: (grandparent william philip) -> (grandchild philip
                                                                  william).
            Querying: (grandparent william philip).
            Query succeeded: (grandparent william philip).
            Asserting: (grandchild philip william).
              Inserting new value (grandchild philip william).
                Propagating addition of (grandchild philip william).
                Applying: (grandchild philip william) -> (grandparent william
                                                                      philip).
                  Querying: (grandchild philip william).
                  Query succeeded: (grandchild philip william).
                  Asserting: (grandparent william philip).
                    Inserting old value (grandparent william philip).
```

```
                Assert succeeded: (grandparent william philip).
              Rule applied.
              Applying: (grandchild philip william) -> (isa philip
                                                             people)
                                                        (isa william
                                                             people).
                Querying: (grandchild philip william).
                Query succeeded: (grandchild philip william).
                Asserting: (isa philip people).
                  Inserting old value (isa philip people).
                Assert succeeded: (isa philip people).
                Asserting: (isa william people).
                  Inserting old value (isa william people).
                Assert succeeded: (isa william people).
              Rule applied.
          Assert succeeded: (grandchild philip william).
        Rule applied.
        Applying: (grandparent william philip) -> (isa william people)
                                                  (isa philip people).
          Querying: (grandparent william philip).
          Query succeeded: (grandparent william philip).
          Asserting: (isa william people).
            Inserting old value (isa william people).
          Assert succeeded: (isa william people).
          Asserting: (isa philip people).
            Inserting old value (isa philip people).
          Assert succeeded: (isa philip people).
        Rule applied.
    Assert succeeded: (grandparent william philip).
    Asserting: (gender philip male).
      Inserting old value (gender philip male).
    Assert succeeded: (gender philip male).
  Rule applied.
  Assert succeeded: (grandfather william philip).
Rule applied.
Applying: (grandfather william ?gf) <- (grandparent william ?gf)
                                       (gender ?gf male).
  Querying: (grandparent william ?gf).
    Applying: (grandparent william ?gf) <-
                (retrieve (coreferent william ?y))
                (Lisp (not (equal 'william '?y)))
                (retrieve (grandparent ?y ?gf)).
      Querying: (retrieve (coreferent william ?y)).
        Querying: (coreferent william ?y).
        Query succeeded: (coreferent william william).
      Query succeeded: (retrieve (coreferent william william)).
      Querying: (Lisp (not (equal 'william 'william))).
      Query failed.
    Rule failed.
  Query succeeded: (grandparent william philip).
  Querying: (gender philip male).
  Query succeeded: (gender philip male).
  Asserting: (grandfather william philip).
    Inserting old value (grandfather william philip).
```

```
    Assert succeeded: (grandfather william philip).
  Rule applied.
Query succeeded: (grandfather william philip).
...

Result:
  Bindings:           ?gm    --- elizabeth [elizabeth]
                      ?gf    --- philip [philip]

Insertions: 6          Rule applications: 25    Iterations: 0.
Unifications: 12       Matches: 35.
Frame insertions: 31   Frame accesses: 387



 (a-query "Who are William's aunts and uncles?"
   '((uncle William ?u)
     (aunt William ?v)))

QUERYING:   Who are William's aunts and uncles?

Result:
  Bindings:           ?v     --- sarah [sarah]
                      ?u     --- andrew [andrew]

 Assumptions:        ((not (coreferent andrew charles)))
                     ((not (coreferent charles andrew)))

Insertions: 13         Rule applications: 57    Iterations: 0.
Unifications: 31       Matches: 106.
Frame insertions: 95   Frame accesses: 1146



 (a-query "Who are William's parents?"
  '((parent William ?u)))

QUERYING:   Who are William's parents?

Result 1:
  Binding:            ?u     --- diana [diana]

Result 2:
  Binding:            ?u     --- charles [charles]

Insertions: 0          Rule applications: 0     Iterations: 0.
Unifications: 0        Matches: 2.
Frame insertions: 0    Frame accesses: 11



 (a-query "Are all Charles' children male ?"
   '((:ALL-PATHS ((child Charles ?b))
                 ((gender ?b male)))))
```

```
QUERYING:   Are all Charles' children male ?

 Query succeeded.

 Insertions: 0           Rule applications: 1     Iterations: 0.
 Unifications: 1         Matches: 3.
 Frame insertions: 2     Frame accesses: 39
```

```
  (a-query "Are all Charles' children female ?"
    '((:ALL-PATHS ((child Charles ?b))
                  ((gender ?b female)))))

QUERYING:   Are all Charles' children female ?

 *Query failed.*

 Insertions: 0           Rule applications: 1     Iterations: 0.
 Unifications: 1         Matches: 3.
 Frame insertions: 2     Frame accesses: 36
```

3.3 Simple Inference

Another straightforward application of Algernon is building simple expert systems. In this section we show how to set up a collection of rules to do diagnosis by checking for the presence of all symptoms of a disease.[2]

First we create a taxonomy of three sets, Objects, Diseases, and Symptoms, and three individuals. This is grafted onto the existing taxonomy at the existing set `objects` (which already includes the set `People`).

```
  (a-assert "Taxonomy"
           '((:taxonomy (Objects
                         (Diseases Flu)
                         (Symptoms Fever Nausea)))))
```

Next we define three new slots:

```
  (a-assert "Slot specifications"
           '((:slot has-disease (people diseases)
                    :comment "(has-disease x d) = x has disease d.")
             (:slot has-symptom (people symptoms)
                    :comment "(has-symptom x s) = x has symptom s.")
             (:slot symptom     (diseases symptoms)
                    :comment "(symptom d s) = The disease d causes symptom s.")))
```

[2]Some of the explanatory text in this section is taken from class handouts prepared by Ben Kuipers. The ancestry of this example can be traced back to an example originally written by Ben Kuipers.

We do not universally backlink `member` to `isa` since we do not want to link large sets (*e.g.*, `things`) to all their members. However, in the diagnostic rule below, we will need to access all members of the set Diseases. Therefore, we assert a forward-chaining rule to create the downward link.

```
(a-assert "Backlink diseases"
           '((:rules Diseases
              ((isa ?d1 Diseases) -> (member Diseases ?d1)))))
```

Backward chaining rules conclude that the patient has a symptom if certain conditions are satisfied. Some conditions can be checked only by querying the user, via the user-interface operator `:ask`.

```
(a-assert "Collecting symptoms"
         '((:rules People
            ((has-symptom ?p fever) <- (temperature ?p ?t) (:lisp (> ?t 99)))
            ((has-symptom ?p nausea) <- (:ask (has-symptom ?p nausea)))
            ((temperature ?p ?t) <- (:ask (temperature ?p ?t))))))
```

A generic diagnostic rule backward-chains to conclude that a patient has a disease if he has all the symptoms associated with that disease:

```
(a-assert "Diagnostic rule"
         '((:rules People
            ((has-disease ?x ?d1)
             <-
             (member Diseases ?d1)
             (:all-paths ((symptom ?d1 ?s)) ((has-symptom ?x ?s)))))))
```

Finally, we assert the facts known about a disease and its symptoms.

```
(a-assert "Symptoms"
         '((symptom Flu Fever)
           (symptom Flu Nausea)))
```

Obviously a real diagnostic expert system must be more complex, to control search in a large space of hypotheses, to focus question asking on the most useful observations, and to handle missing, incomplete, fuzzy, or contradictory information.

We can the invoke the diagnostic rules by asking what diseases a person has:

```
(a-query "What does John have?"
        '((:the ?john (name ?john (:quote (John))) (isa ?john People))
          (has-disease ?john ?d)))
```

```
QUERYING:   What does John have?
 Is it true that (has-symptom john nausea)?  (y or n)  y
 Give me a value for ?t in (temperature john ?t): 105

 Result:
   Bindings:           ?d     --- flu [flu]
                       ?john   --- john [john]
```

```
Created frame:       john

Insertions: 11         Rule applications: 25    Iterations: 0.
Unifications: 9        Matches: 32.
Frame insertions: 24   Frame accesses: 400
```

3.4 Representing a Simple Discourse

Some facility is provided in Algernon for representing a simple discourse. The slot `recent` in the frame for the current context can be used for simple pronoun resolution by storing objects which have been referred to recently. Algernon also provides a special form `:the` which implements a simple theory of definite descriptions. The form `(:the ?x path)` first queries `path`. If only a single binding for `?x` results, then the form succeeds, since `path` has referenced the unique frame for `?x`. If multiple bindings for `?x` result then the form fails, since the definite description is not unique. If the query fails then a new frame is created, `?x` is bound to the new frame, and `path` is asserted.

First the title of the story to establish a new context:[3]

```
(a-assert "The John Alden Story."
   '((:clear-slot global-context current-context)
     (:create ?ja-story ja-story)
     (current-context global-context ?ja-story)))
```

This example uses the taxonomy in the background knowledge-base, plus the new set `time-units`:

```
(a-assert "Extension to taxonomy."
   '((:taxonomy (objects (time-units years)))))
```

The new slot `age` is a three place relation between a physical object, its age (which will be a Lisp integer — we type this position to `nil` since we do not create frames each of the integers) and the time units that age is measured in:

```
(a-assert "Age."
   '((:slot age (physical-objects nil time-units)
            :cardinality 1
            :comment "(age x 23 years) = x is 23 years old.")))
```

We use the `speaker` slot in the current context to hold the speaker in the story. We then treat "my" or "I" as a definite description, creating the speaker if none is known:

```
(a-assert "My name is John Alden."
   '((:the ?me (speaker (current-context global-context) ?me))
     (name ?me (:quote (John Alden)))))

(a-assert "I am 25 years old."
   '((:the ?me (speaker (current-context global-context) ?me))
     (age ?me 25 years)))
```

[3] The ancestry of this example can be traced back to an example originally written by Ben Kuipers.

When a frame is mentioned we put it in the `recent` slot of the current context so we can access it with third person pronouns later:

```
(a-assert "My wife is 23 years old."
 '((:the ?me (speaker (current-context global-context) ?me))
   (:the ?w (wife ?me ?w))
   (age ?w 23 years)
   ;; Add to (unordered) context:
   (recent (current-context global-context) ?w)))

(a-assert "Her name is Priscilla."
 '((:the ?her
         (recent (current-context global-context) ?her)
         (gender ?her female))
   (name ?her (:quote (Priscilla)))))
```

Indefinite descriptions (*e.g.*, "a friend") are usually represented using `:forc`:

```
(a-assert "She has a friend named Miles Standish."
 '((:the ?her
         (recent (current-context global-context) ?her)
         (gender ?her female))
   (:forc ?ms
          (friend ?her ?ms)
          (name ?ms (:quote (Miles Standish)))
          ;; Have to explicitly add gender since
          ;; Algernon doesn't know Miles is a man's name:
          (gender ?ms male))
   (recent (current-context global-context) ?ms)))

(a-assert "He is 40 years old."
 '((:the ?him
         (recent (current-context global-context) ?him)
         (gender ?him male))
   (age ?him 40 years)))

(a-assert "Priscilla also has a friend named Cotton Mather."
 '((:forc ?cm
          (friend Priscilla ?cm)
          (name ?cm (:quote (Cotton Mather)))
          (gender ?cm male))
   (recent (current-context global-context) ?cm))))
```

We can then retrieve facts from the knowledge-base using queries. The form `:show` displays the current contents of a frame. Before the queries we first 'return attention' to the John Alden story (by setting the current context to the John Alden story):

```
;; First we return attention to the John Alden story:
;;
(a-assert "In the John Alden story ..."
   '((:clear-slot global-context current-context)
     (current-context global-context ja-story)))
```

```
ASSERTING:   In the John Alden story ...

 Assert succeeded.

 Insertions: 1          Rule applications: 2     Iterations: 0.
 Unifications: 0        Matches: 2.
 Frame insertions: 1    Frame accesses: 33


  (a-query "How old is my wife ?"
    '((:the ?me (speaker (current-context global-context) ?me))
      (:the ?w (wife ?me ?w))
      (age ?w ?a ?units)))

QUERYING:   How old is my wife ?

 Result:
   Bindings:            ?units   --- years [years]
                        ?a     --- 23
                        ?w     --- frame1-wife [priscilla]
                        ?me    --- frame1 [john alden]
                        ?$x12   --- ja-story [ja-story]

 Insertions: 0          Rule applications: 0     Iterations: 0.
 Unifications: 0        Matches: 4.
 Frame insertions: 0    Frame accesses: 32



  (a-query "Who is Priscilla's husband?"
   '((:show (husband Priscilla))))

QUERYING:   Who is Priscilla's husband?

Frame1:
  isa:        (objects nil) (physical-objects nil) (things nil) (people nil)
  coreferent: (frame1 nil)
  name:       ((john alden) nil)
  age:        ((25 years) nil)
  wife:       (frame1-wife nil)
  gender:     (male nil)
  spouse:     (frame1-wife nil)

 Result:
   Binding:             ?$x13   --- frame1 [john alden]

 Insertions: 0          Rule applications: 0     Iterations: 0.
 Unifications: 0        Matches: 1.
 Frame insertions: 0    Frame accesses: 10



  (a-query "What is Priscilla's friend's name ?"
    '((name (friend Priscilla) ?name)))
```

```
QUERYING:   What is Priscilla's friend's name ?

 Result 1:
   Bindings:             ?name   --- (cotton mather)
                         ?$x14   --- frame1-wife-friend1 [cotton mather]

 Result 2:
   Bindings:             ?name   --- (miles standish)
                         ?$x14   --- frame1-wife-friend [miles standish]

 Insertions: 0           Rule applications: 3     Iterations: 0.
 Unifications: 3         Matches: 7.
 Frame insertions: 6     Frame accesses: 78
```

3.5 Reasoning by Contradiction

Complex reasoning problems often involve more than simple forward and backward rule chaining. As a simple example, if one has two rules of form:

$$p \rightarrow q$$

$$p \rightarrow \neg q$$

then logically one should be able to derive $\neg p$. Many knowledge representation systems include a theorem prover to allow such conclusions to be drawn. The problem with such an approach is that (even without quantification) the time complexity of any known complete inference algorithm is exponential. What this means in practice is that the run time of any given operation is unpredictable. The solution in Algernon is to reason by contradiction. As discussed in chapter 6, the addition of *reductio ad absurdum* to forward and backward rule chaining allows Socratically Complete reasoning, while preserving the tractability of individual queries. In this section we examine two problems which Algernon can only solve after being given an appropriate set of leading questions. I hope to show that in such cases the desired conclusions are not intuitively obvious (*i.e.*, that there are good reasons for Algernon to have trouble) and that the preliminary queries required by Algernon are not very different from questions one asks oneself in solving such problems.

Consider first a situation in which:

1. If Mary waters the flowers and the flowers bloom then Mary is happy.

2. If Mary does not water the flowers then they do not bloom.

3. Mary is not happy.

Do the flowers bloom ? It may not be obvious, but one can show that they do not.[4] One can express this situation in Algernon using the following assertions:

[4]The conclusion that the flowers do not bloom follows using only predicate calculus — no closed world assumption or default reasoning is required.

```
(a-assert "Taxonomy"
          '((:taxonomy (objects (plants)))))

(a-assert "Slots"
          '((:slot happy (people booleans)
                   :comment "(happy p true) = p is happy.")
            (:slot waters (people plants)
                   :comment "(waters p plant) = p waters plant.")
            (:slot blooms (plants booleans)
                   :comment "(blooms plant true) = plant blooms.")))

(a-assert "Mary"
          '((:create ?m Mary) (isa ?m people)
            (:create ?f Marys-Flowers) (isa ?f plants)
            (:rules people
                    ((happy ?m true) <- (waters ?m ?f) (blooms ?f true)))
            (:rules plants
                    ((not (blooms ?f true)) <- (not (waters ?m ?f))))
            (not (happy ?m true))))
```

We can ask if the flowers fail to bloom, but the query fails after only three rule applications. This is a result of the access limitations built into Algernon. There are other rules which are relevant to the query, but Algernon has no access path to them:

```
  (a-query "Do the flowers bloom ?"
          '((not (blooms Marys-Flowers true))))

QUERYING:   Do the flowers bloom ?

 *Query failed.*

 Insertions: 0           Rule applications: 3     Iterations: 0.
 Unifications: 3         Matches: 2.
 Frame insertions: 5     Frame accesses: 48
```

To use *reductio ad absurdum* we first make an assumption, show it leads to a contradiction, and then conclude the negation of the assumption. To solve this problem we must make two assumptions. First we assume that the flowers bloom, and then, within the context of this assumption, we assume that Mary waters the flowers. We then ask if Mary is happy, and derive a contradiction which forces us to drop the assumption that Mary waters the flowers. We are thus able to conclude that Mary does not water the flowers. When we then ask if the flowers bloom, we derive a second contradiction, which forces us to conclude that the flowers do not bloom. The logical details of this example are discussed in more detail in section 6.1. In the output below we include a trace of contradictions and their resolutions.

```
(a-query "Proof by contradiction."
         '((:assume (blooms Marys-Flowers true))
           (:assume (waters Mary Marys-Flowers))
           (happy Mary true)))
```

```
QUERYING:   Proof by contradiction.

  ** Beginning raa Trace **
 Contradiction:       (happy mary true)
                      (not (happy mary true))

   (happy mary true) supported by assumptions:
     ((waters mary marys-flowers) (blooms marys-flowers true))
   (not (happy mary true)) supported by assumptions:
     nil

 Dropping assumption: (waters mary marys-flowers).
 Asserting its negation: (not (waters mary marys-flowers)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 4          Rule applications: 9     Iterations: 0.
 Unifications: 7        Matches: 14.
 Frame insertions: 16   Frame accesses: 234



  (a-query "Do the flowers not bloom ?"
           '((not (blooms Marys-Flowers true)))))

QUERYING:   Do the flowers not bloom ?

  ** Beginning raa Trace **
 Contradiction:       (not (blooms marys-flowers true))
                      (blooms marys-flowers true)

   (not (blooms marys-flowers true)) supported by assumptions:
     ((blooms marys-flowers true))
   (blooms marys-flowers true) supported by assumptions:
     ((blooms marys-flowers true))

 Dropping assumption: (blooms marys-flowers true).
 Asserting its negation: (not (blooms marys-flowers true)).
  ** End raa Trace **

 Query succeeded.

 Insertions: 2          Rule applications: 3     Iterations: 0.
 Unifications: 3        Matches: 2.
 Frame insertions: 6    Frame accesses: 89
```

One might argue that in more complex cases the series of preliminary queries (and assumptions) might be less obvious. Theoretically this argument must be correct since the problem of determining this series in general is intractable. However, in actual problems the series of necessary queries is often surprisingly intuitive. As an example, consider the following problem (from [Wylie, 57]):

In a certain bank the positions of cashier, manager, and teller are held by Brown, Jones and Smith, though not necessarily respectively. The teller, who was an only child, earns the least. Smith, who married Brown's sister, earns more than the manager. What position does each man fill ?

As we shall see, the necessary series of queries and assumptions corresponds to the following questions in English:

1. If Smith were the manager then how could he earn more than the manager ?
2. If Smith were the teller then how could he earn more than the manager ?
3. If Brown were the teller then how could he have a sister ?

Which is a set of questions which one might ask oneself in solving this problem.

One interesting representational issue which arises in this problem is how we should represent "though not necessarily respectively" — that is, how to we represent the one to one relationship between the set consisting of Brown, Jones and Smith, and the set consisting of the cashier, the manager, and the teller. We currently represent this relationship by creating frames for the two sets, and asserting the sets stand in a `one-to-one` relationship. One to one relationships between sets are an important part of common-sense reasoning and are defined by rules in the knowledge-base built into Algernon. We first define `one-to-one` as equivalent to bi-directional `one-to-one-into` links:

```
((one-to-one ?s1 ?s2) <- (one-to-one-into ?s1 ?s2) (one-to-one-into ?s2 ?s1))
((one-to-one ?s1 ?s2) -> (one-to-one-into ?s1 ?s2) (one-to-one-into ?s2 ?s1))
```

Intuitively `(one-to-one-into ?s1 ?s2)` means that every member of `?s2` is coreferent with some member of `?s1`.[5] The relationship `(cf-member ?x ?s1)` means that `?x` is coreferent with some member of `?s1` (*e.g.*, "the murderer is someone in this room"). We can thus derive `cf-member` from `one-to-one-into`:

```
((cf-member ?x ?s1) <- (isa ?x ?s2) (one-to-one-into ?s2 ?s1))
```

Finally, from `cf-member` we can use process of elimination to find out which member of `?s1` `?x` is coreferent with. Notice that this rule can only fire if `?s1` is `complete` — all its members are known:

```
((coreferent ?x ?y)
 <-
 (cf-member ?x ?s1) (complete ?s1 true) (member ?s1 ?y)
 (:ALL-PATHS ((member ?s1 ?m1) (:neq ?y ?m1))
             ((not (coreferent ?x ?m1)))))
```

The facts in the bank problem can be asserted as follows:

```
(a-assert "New sets."
          '((:taxonomy (objects (companies Bank)
                                (positions cashier manager teller)
                                (people)))))
(a-assert "New slots."
   '((:slot sister (people people)
```

[5] One-to-one mappings should eventually be extended to allow other relations besides coreference. For example one might want to say that there exists a one-to-one mapping "has-number" between citizens and Social Security numbers.

```
           :comment "(sister a b) = The sister of a is b.")
     (:slot only-child (people booleans)
           :cardinality 1
           :comment "(only-child a true) = a is an only child.")
     (:slot holds (people companies positions)
           :comment "(holds ?p ?c ?pos) =
                     ?p holds position ?pos in company ?c.")
     (:slot position (companies positions people)
           :comment "(position ?c ?pos ?p) =
                     In company ?c, ?pos is held by ?p.")))

(a-assert "Sisters and only children."
          '((:RULES people
                    ((sister ?p1 ?p2) -> (not (only-child ?p1 true))))))

(a-assert "Holds and position."
          '((:SRULES (:slot position)
                     ((position ?c ?pos ?p) -> (holds ?p ?c ?pos)))
            (:SRULES (:slot holds)
                     ((holds ?p ?c ?pos) -> (position ?c ?pos ?p)))))

(a-assert "Forward chaining rule for cf-member."
          '((:RULES sets
                    ((one-to-one-into ?s1 ?s2) (member ?s1 ?x) ->
                     (cf-member ?x ?s2)))))

(a-assert "In a certain bank the positions of cashier, manager, and teller
   are held by Brown, Jones and Smith, though not necessarily respectively."
   '(;; First the bank and its positions (note that we are careful here to
     ;; distinguish between the position of cashier and the person holding
     ;; the job cashier):
     (:forc ?cp (position Bank cashier ?cp))
     (:forc ?mp (position Bank manager ?mp))
     (:forc ?tp (position Bank teller ?tp))
     ;;
     (:create ?b Brown) (:create ?s Smith) (:create ?j Jones)
     (isa ?b people) (isa ?j people) (isa ?s people)
     ;;
     ;; the set of people holding the positions:
     (:create ?pos posts) (member ?pos ?cp) (member ?pos ?mp) (member ?pos ?tp)
     (complete ?pos true)
     ;;
     ;; the set {brown,smith,jones}:
     (:create ?emp employees) (member ?emp ?b) (member ?emp ?j) (member ?emp ?s)
     (complete ?emp true)
     ;;
     ;; and the relationship between these sets:
     (one-to-one ?pos ?emp)
     ;;
     ;; finally, the implicit assumption that Brown, Jones and Smith are
     ;; different people:
     (:assume (not (coreferent ?b ?j)))
     (:assume (not (coreferent ?j ?s)))
     (:assume (not (coreferent ?s ?b)))))
```

```
;; Hack -- Here I use "least" when I really should use "earns-least".
;; I use least since the rules for reasoning with it are included in the
;; built in knowledge-base.
(a-assert "The teller, who was an only child, earns the least."
   '((position Bank teller ?tp) (only-child ?tp true)
     (least ?tp posts)))

(a-assert "Smith, who married Brown's sister, earns more than the manager."
   '((:forc ?sis (sister Brown ?sis))
            (spouse Smith ?sis)
            (position Bank manager ?man)
            (greater Smith ?man)))
```

We can then determine which position each man fills. Notice that the initial query below fails after applying only seven rules, while the final solution (through the preliminary queries and assumptions) requires several hundred rule applications. Several of the operations consist of assumptions, followed by queries which uncover contradictions. Algernon responds that such operations "failed" since the final query fails (because it uncovers a contradiction causing an assumption to be retracted causing the query to fail). However, from the point of view of solving the problem as a whole the operation succeeds, because it results in the negation of the assumption being added to the knowledge-base. For the operations which result in contradictions we include some tracing output to show the contradiction and how it is resolved.

Note that we give Algernon a small break here; We ask `(coreferent Brown ?be)` rather than `(holds Brown Bank ?post)`. Algernon inherits through known coreference links (thus from `(coreferent Brown frame1)` and `(holds frame1 Bank manager)` it could deduce `(holds Brown Bank manager)`), but when asked `(holds Brown Bank ?post)` Algernon would not know to answer this question by looking for frames coreferent with `Brown` (such a strategy should probably come from background knowledge about logic problems, but we have not studied how it could be represented in Algernon).

```
  (a-query "What positions do Brown, Smith and Jones hold ?"
          ;; Note that we read this as "Brown, Smith and Jones are
          ;; coreferent with which employees ?".
          '((coreferent Brown ?be) (holds ?be bank ?post)
            (coreferent Smith ?se) (holds ?se bank ?post)
            (coreferent Jones ?je) (holds ?je bank ?post)))

QUERYING:   What positions do Brown, Smith and Jones hold ?

 *Query failed.*

 Insertions: 3          Rule applications: 7     Iterations: 0.
 Unifications: 7        Matches: 26.
 Frame insertions: 25   Frame accesses: 182


  (a-query "If Smith were the manager then could he earn more than the manager ?"
          '((position Bank manager ?man)
            (:assume (coreferent Smith ?man))
```

```
            (not (greater Smith ?man))))

QUERYING:   If Smith were the manager then how could he earn more than the manager ?

  ** Beginning raa Trace **
 Contradiction:        (not (greater smith frame2))
                       (greater smith frame2)

   (not (greater smith frame2)) supported by assumptions:
     ((coreferent smith frame2))
   (greater smith frame2) supported by assumptions:
     nil

 Dropping assumption: (coreferent smith frame2).
 Asserting its negation: (not (coreferent smith frame2)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 27         Rule applications: 96    Iterations: 0.
 Unifications: 35       Matches: 258.
 Frame insertions: 166  Frame accesses: 1789



  (a-query "If Smith were the teller then could he earn more than the manager?"
          '((position Bank teller ?t)
            (:assume (coreferent Smith ?t))
            (position Bank manager ?man)
            (not (greater Smith ?man))))

QUERYING:   If Smith were the teller then how could he earn more than the manager?

  ** Beginning raa Trace **
 Contradiction:        (not (greater smith frame2))
                       (greater smith frame2)

   (not (greater smith frame2)) supported by assumptions:
     ((coreferent smith frame3))
   (greater smith frame2) supported by assumptions:
     nil

 Dropping assumption: (coreferent smith frame3).
 Asserting its negation: (not (coreferent smith frame3)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 28         Rule applications: 90    Iterations: 0.
 Unifications: 35       Matches: 243.
 Frame insertions: 164  Frame accesses: 1837
```

```
  (a-query "Hence, Smith holds which job ?"
          '((coreferent Smith ?se) (holds ?se bank ?post)))

QUERYING:   Hence, Smith holds which job ?

 Result 1:
   Bindings:            ?post   --- cashier [cashier]
                        ?se    --- frame1 [nil]

 Result 2:
   Bindings:            ?post   --- cashier [cashier]
                        ?se    --- smith [smith]

 Created frame:        frame1-selfset

 Insertions: 15          Rule applications: 33    Iterations: 0.
 Unifications: 6         Matches: 88.
 Frame insertions: 50    Frame accesses: 536



  (a-query "If Brown were the teller then would he be an only child ?"
          '((position Bank teller ?t)
            (:assume (coreferent Brown ?t))
            (only-child Brown true)))

QUERYING:   If Brown were the teller then would he be an only child ?

  ** Beginning raa Trace **
 Contradiction:        (only-child brown true)
                       (not (only-child brown true))

   (only-child brown true) supported by assumptions:
     ((coreferent brown frame3))
   (not (only-child brown true)) supported by assumptions:
     nil

 Dropping assumption: (coreferent brown frame3).
 Asserting its negation: (not (coreferent brown frame3)).
  ** End raa Trace **


 *Query failed.*

 Insertions: 19          Rule applications: 40    Iterations: 0.
 Unifications: 14        Matches: 122.
 Frame insertions: 72    Frame accesses: 789



  (a-query "Hence Brown holds which job ?"
          '((coreferent Brown ?be) (holds ?be bank ?post)))

QUERYING:   Hence Brown holds which job ?
```

```
 Result 1:
   Bindings:            ?post   --- manager [manager]
                        ?be    --- frame2 [nil]

  Assumption:          ((not (coreferent smith brown))
                        (not (coreferent jones smith))
                        (not (coreferent brown jones)))

 Result 2:
   Bindings:            ?post   --- manager [manager]
                        ?be    --- brown [brown]

  Assumption:          ((not (coreferent smith brown))
                        (not (coreferent jones smith))
                        (not (coreferent brown jones)))

 Deleted value:        (not (coreferent jones frame3))

 Insertions: 13          Rule applications: 40    Iterations: 0.
 Unifications: 17        Matches: 150.
 Frame insertions: 101   Frame accesses: 1206



  (a-query "Hence Jones holds which job ?"
          '((coreferent Jones ?je) (holds ?je bank ?post)))

QUERYING:   Hence Jones holds which job ?

 Result 1:
   Bindings:            ?post   --- teller [teller]
                        ?je    --- frame3 [nil]

 Result 2:
   Bindings:            ?post   --- teller [teller]
                        ?je    --- jones [jones]

 Created frame:        jones-selfset

 Insertions: 21          Rule applications: 63    Iterations: 0.
 Unifications: 14        Matches: 213.
 Frame insertions: 110   Frame accesses: 1134
```

Note that the last query (as well as two others) returns two sets of variable bindings. This is because the path succeeds with `?je` bound to either `frame3` (the frame for "the teller") or `jones`. Since these two frames are now known to be coreferent, any fact true of either frame is inherited by the other.

3.6 Reasoning with Disjunctions Using *Modus Tolens*

One hard problem in rule based systems is the representation of pure disjunctions (*e.g.*, `manager(Jones) OR teller(Jones) OR cashier(Jones)`). The basic problem is that for a disjunction of length n there are n possible rules that can represent the disjunction. Choosing any one such rule (*e.g.*, `manager(Jones) <- (not teller(Jones)), (not cashier(Jones))`) seems like a hack since we really want to say – "one of these is true but we have no way at this point to tell which one".

If one knows that a disjunction holds, and one knows that all but one of the disjuncts are false then one obvious deduction one should be able to draw is that the last disjunct must be true. If we represent the disjunction using a rule, unless we chose exactly the right rule (or include all n rules), we will only be able to draw this conclusion in Algernon using a fairly obscure proof by contradiction. However, this proof by contradiction can be avoided in all cases by adding to Algernon the ability to reason by *modus tolens*.

While the basic form of *modus ponens* is:

$$\begin{array}{l} p \rightarrow q \\ \underline{p \qquad\quad} \\ q \end{array}$$

the basic form of *modus tolens* is:

$$\begin{array}{l} p \rightarrow q \\ \underline{\neg q \qquad\quad} \\ \neg p \end{array}$$

The difficulty in using *modus tolens* within Algernon is that rules do not seem to be correctly indexed. If we make an assertion of form $\neg q$, it is not at all clear how to access the rule $p \rightarrow q$, so as to conclude $\neg p$. The solution is to *invert the roles of backward and forward chaining rules.* Rules which would apply as forward chaining rules (using *modus ponens*) apply as back chaining rules using *modus tolens*. Thus, if we have an if-needed rule of form $p \leftarrow q$, and we assert $\neg p$, then we can forward chain and conclude $\neg q$. Similarly, if we have an if-added rule of form $p \rightarrow q$, and a ground query $\neg p$, then we can backchain and query $\neg q$.

However, many rules have antecedents with more than one term which makes things a bit more complex. Consider a backward chaining rule of form:

$$p \leftarrow q_1, q_2, \ldots, q_n.$$

This is logically equivalent to:

$$p \leftarrow \neg(q_1, q_2 \ldots, q_{n-1} \rightarrow \neg q_n),$$

and so if we assert $\neg p$ then by *modus tolens* we can add the rule:

$$q_1, q_2, \ldots, q_{n-1} \rightarrow \neg q_n$$

Note that this gives a form of *if-needed rule completion* which is symmetrical to the if-added rule completion discussed in section 2.3.5 and section 5.3.6. Similarly, a forward chaining rule of form:

$$p_1, p_2, \ldots, p_n \rightarrow q$$

is logically equivalent to:

$$p_1 \rightarrow \neg(p_2, \ldots, p_n, \neg q)$$

and so if we have a ground query $\neg p_1$ then by *modus tolens* we can backchain and query the path $p_2, \ldots, p_n, \neg q$. Both types of inference respect Algernon's access limitations (*i.e.*, only paths are asserted and queried).

Consider a simple disjunction of form $a \vee b \vee c$. One possible rule to represent this disjunction is:

$$a \leftarrow \neg b, \neg c$$

There are then three cases:

1. We assert $\neg a$ and $\neg b$, and then query c. By *modus tolens*, $\neg a$ adds rule:

$$\neg b \rightarrow c$$

   Thus the assertion of $\neg b$ forward chains and asserts c, and the query of c immediately succeeds.

2. We assert $\neg b$ and $\neg c$, and query a. The query succeeds by *modus ponens*.

3. We assert $\neg a$ and $\neg c$ and query b. Again, $\neg a$ adds the rule:

$$\neg b \rightarrow c$$

   Thus when we query b this rule applies backwards by *modus tolens* and succeeds.

The reader can verify that if we had chosen a forward chaining rule then the three queries would have still succeeded.

In general, the combination of *modus ponens* and *modus tolens* allows Algernon to reason from a disjunction and the negation of all but one disjunct and conclude the last disjunct, no matter which rule is chosen to represent the disjunct.

One application of *modus tolens* is the representation of one to one relationships between small sets. For example, in the bank problem from the previous section, there is a one to one relationship between the set consisting of Brown, Jones and Smith, and the set consisting of the cashier, the manager, and the teller. One way to represent this relationship is to use three disjunctions saying: each employee holds a position, one employee can hold at most one position, and one position can be held by only one employee. This representation is more verbose than that used in the last section, but it avoids the use of the form `:all-paths` and thus allows the bank problem to be solved within ALL as formalized in part two of this dissertation. The full bank example using this representation is shown below. The Algernon form `:w-contra-positive` enables reasoning by *modus tolens* within its scope (reasoning by *modus tolens* is currently implemented in Algernon by adding the contra-positive of rules, however it could equivalently, and with some savings of space, be implemented by adding *modus tolens* to inference in Algernon).

```
(a-assert "New slots."
          '((:slot sister (people people)
                   :comment "(sister a b) = The sister of a is b.")
            (:slot only-child (people booleans)
                   :cardinality 1
                   :comment "(only-child a true) = a is an only child.")))

(a-assert "Sisters and only children."
          '((:RULES people
                    ((sister ?p1 ?p2) -> (not (only-child ?p1 true))))))

(a-assert "In a certain bank the positions of cashier, manager, and teller
           are held by Brown, Jones and Smith, though not necessarily
           respectively."
          '(; the set of positions:
            (:create ?c cashier) (:create ?t teller) (:create ?m manager)
            (:create ?pos positions)
            (member ?pos ?c) (member ?pos ?m) (member ?pos ?t)
            ;;
            ;; the employees:
            (:create ?b Brown) (:create ?j Jones) (:create ?s Smith)
            (isa ?b people) (isa ?j people) (isa ?s people)
            (:create ?e employees) (member ?e ?b) (member ?e ?j) (member ?e ?s)

            ;; Here are the key rules for establishing the one to one
            ;; relationship between the set of employees and the set
            ;; of positions:
            (:contra-positive
             (:rules ?e
                     ;; These rules hold for all employees:
                     ;;
                     ;; First existence -- Every employee holds a job:
                     ((coreferent ?e1 ?c)
                      <- (not (coreferent ?e1 ?t)) (not (coreferent ?e1 ?m)))

                     ;; Then uniqueness -- one employee per position
                     ;; and vice versa:
                     ((coreferent ?e1 ?p1) (member ?pos ?p1)
                      ->
                      (member ?e ?e2) (:neq ?e1 ?e2)
                      (not (coreferent ?e2 ?p1)))
                     ((coreferent ?e1 ?p1) (member ?pos ?p1)
                      ->
                      (member ?pos ?p2) (:neq ?p1 ?p2)
                      (not (coreferent ?e1 ?p2)))))))

;; Hack -- Here I use "least" when I really should use "earns-least".
(a-assert "The teller, who was an only child, earns the least."
          '((only-child teller true) (least teller positions)))

(a-assert "Smith, who married Brown's sister, earns more than the manager."
          '((:forc ?sis (sister Brown ?sis))
            (spouse Smith ?sis)
            (greater Smith manager))))
```

Then the queries as in the last section (page 53). Here we set the variable `*contra-positive*` to `t` so that reasoning by *modus tolens* will be enabled during all the queries.[6]

```
  (let ((*contra-positive* t))
    (a-query "What positions do Brown, Smith and Jones hold ?"
             ;;; Note that we read this as "Brown, Smith and Jones
             ;;; are coreferent with which employees ?".
             '((coreferent Brown ?be) (isa ?be positions)
               (coreferent Smith ?se) (isa ?se positions)
               (coreferent Jones ?je) (isa ?je positions)))

QUERYING:   What positions do Brown, Smith and Jones hold ?

 *Query failed.*

 Insertions: 1          Rule applications: 6     Iterations: 0.
 Unifications: 6        Matches: 5.
 Frame insertions: 10   Frame accesses: 96



    (a-query "If Smith were the manager then could he earn more than the manager ?"
             '((:assume (coreferent Smith manager))
               (not (greater Smith manager))))

QUERYING:   If Smith were the manager then how could he earn more than the manager ?

  ** Beginning raa Trace **
 Contradiction:        (not (greater smith manager))
                       (greater smith manager)

   (not (greater smith manager)) supported by assumptions:
     ((coreferent smith manager))
   (greater smith manager) supported by assumptions:
     nil

 Dropping assumption: (coreferent smith manager).
 Asserting its negation: (not (coreferent smith manager)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 31         Rule applications: 107   Iterations: 0.
 Unifications: 39       Matches: 283.
 Frame insertions: 183  Frame accesses: 1750


    (a-query "If Smith were the teller then could he earn more than the manager."
             '((:assume (coreferent Smith teller))
               (not (greater Smith manager))))
```

[6] We should not have to do this. Future version of Algernon will allow individual rules to be marked as "disjunctions" and Algernon will apply *modus tolens* to these rules only.

```
QUERYING:   If Smith were the teller then how could he earn more than the manager?

  ** Beginning raa Trace **
 Contradiction:       (not (greater smith manager))
                      (greater smith manager)

   (not (greater smith manager)) supported by assumptions:
     ((coreferent smith teller))
   (greater smith manager) supported by assumptions:
     nil

 Dropping assumption: (coreferent smith teller).
 Asserting its negation: (not (coreferent smith teller)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 28         Rule applications: 93    Iterations: 0.
 Unifications: 34       Matches: 234.
 Frame insertions: 162  Frame accesses: 1682



    (a-query "Hence, Smith holds which job ?"
             '((coreferent Smith ?se) (isa ?se positions)))

QUERYING:   Hence, Smith holds which job ?

 Result 1:
   Binding:             ?se    --- cashier [cashier]

 Result 2:
   Binding:             ?se    --- smith [smith]

 Deleted values:      (not (coreferent brown teller))
                      (not (coreferent jones teller))
                      (not (coreferent cashier smith))

 Created frames:      brown-selfset jones-selfset cashier-selfset

 Insertions: 33         Rule applications: 64    Iterations: 0.
 Unifications: 8        Matches: 106.
 Frame insertions: 65   Frame accesses: 977



    (a-query "If Brown were the teller then would he be an only child ?"
             '((:assume (coreferent Brown teller))
               (only-child Brown true)))

QUERYING:   If Brown were the teller then would he be an only child ?

  ** Beginning raa Trace **
```

```
Contradiction:       (only-child brown true)
                     (not (only-child brown true))

  (only-child brown true) supported by assumptions:
    ((coreferent brown teller))
  (not (only-child brown true)) supported by assumptions:
    nil

Dropping assumption: (coreferent brown teller).
Asserting its negation: (not (coreferent brown teller)).
 ** End raa Trace **

*Query failed.*

Insertions: 19         Rule applications: 72    Iterations: 0.
Unifications: 20       Matches: 267.
Frame insertions: 140  Frame accesses: 1108



   (a-query "Hence Brown holds which job ?"
            '((coreferent Brown ?be) (isa ?be positions)))

QUERYING:  Hence Brown holds which job ?

Result 1:
  Binding:             ?be   --- manager [manager]

Result 2:
  Binding:             ?be   --- brown [brown]

Insertions: 2          Rule applications: 12    Iterations: 0.
Unifications: 11       Matches: 24.
Frame insertions: 24   Frame accesses: 182



   (a-query "Hence Jones holds which job ?"
            '((coreferent Jones ?je) (isa ?je positions))))

QUERYING:  Hence Jones holds which job ?

Result 1:
  Binding:             ?je   --- teller [teller]

Result 2:
  Binding:             ?je   --- jones [jones]

Insertions: 4          Rule applications: 17    Iterations: 0.
Unifications: 7        Matches: 44.
Frame insertions: 28   Frame accesses: 268
```

3.7 Simple Reasoning About Sets of Similar Objects

An important class of common-sense reasoning problems involves reasoning about sets of similar objects. Some such reasoning requires full first order quantification, but in other cases simpler and more efficient representations can be used. One interesting type of problem is exemplified by the sterilization problem proposed by McCarthy in [McCarthy, 87] p. 1032:

> Consider the rationale of canning. We say a container is sterile if it is sealed and all bacteria in it are dead. This can be expressed as a fragment of a Prolog program as follows:
>
> ```
> sterile(X) :- sealed(X), not alive-bacterium(Y,X).
> alive-bacterium(Y,X) :- in(Y,X),bacterium(Y),alive(Y).
> ```
>
> However, a Prolog program incorporating this fragment directly can sterilize a container only by killing each bacterium individually. It would require that some other part of the program successively generate the names of the bacteria. It cannot be used to discover or rationalize canning — sealing the container and then heating it to kill the bacteria at once. The reasoning rationalizing the practice of canning involves the use of quantifiers in an essential way.

Notice that the first rule cannot be translated directly into Algernon. We could try:

```
(sterile ?x) <- (sealed ?x), (not (alive-bacterium ?y ?x)).
```

This is not an access path but can be made into one by simply re-ordering the arguments to `alive-bacterium`. The real problem is that this Algernon rule is equivalent to the predicate calculus statement:

$$(\forall x, y. sterile(?x) \leftarrow (sealed(?x) \wedge \neg alive\_bacterium(?y, ?x)))$$

which is not at all the intended meaning ($sterile(?x)$ would follow if *any* bacteria in the container were dead). The intended meaning is something like:

$$(\forall x. sterile(?x) \leftarrow (sealed(?x) \wedge (\forall ?y. \neg alive\_bacterium(?y, ?x))))$$

To express such a statement in Algernon we introduce the idea of the *representative* of a set. A representative is a frame in the knowledge-base which does not correspond to any object in the world. Rather, the representative holds the facts which are true of *all* members of the set.[7] If `bacteria-in` relates a container to the set of bacteria in it, and `representative` relates a set to its representative, then we can write the rule as:

```
(sterile ?x) <- (sealed ?x) (bacteria-in ?x ?bin)
                (representative ?bin ?r) (dead ?r)
```

[7] One might also want to introduce *prototypes* which hold facts true of 'most' members of a set. Default reasoning is outside the scope of this section, but some thoughts on it are given in section 3.12.

Representatives are a special case of the general idea of *arbitrary objects*. Arbitrary objects are intuitively the "x's" and "y's" in mathematical proofs ("... consider an x, y and z such that $x^2 + y^2 = z^2$ ..."). A full reformulation of first order logic using arbitrary objects has been worked out in [Fine, 85]. We expect future work on the expression of quantification in Algernon and ALL to be based on the theory of arbitrary objects.

The full text of the Algernon solution to the sterilization example is given below. First we define representatives. Note that there is a lot about representatives we do not say here because it is not needed for this example (*e.g.*, the rule that any fact about a representative can be inherited by any member of its set).

```
(a-assert "Taxonomy"
          '((:taxonomy (objects (representatives)))))

(a-assert "Slots"
          '((:slot representative (sets representatives)
                   :partition set-partition
                   :comment "(representative ?s ?r) =
                             The representative of set ?s is ?r.")))

(a-assert "Rules"
          '((:rules sets
                    ;; Every set has a representative:
                    ((representative ?s ?r) <- (:forc ?r (representative ?s ?r)))
                    ;; Every representative is a member of the set:
                    ((representative ?s ?r) -> (member ?s ?r)))))
```

Then we set up the sterilization problem itself. One interesting wrinkle in this problem is that `representative` is in the set partition, while the rest of the example is in the main partition. Solving the problem requires a certain amount of reasoning about the representative of the set of bacteria in the container and this reasoning can only be done in the set partition. To solve this problem we use the form `:in-own-partition` (within the rule for `sterile`) to instruct Algernon to reason about the representative in its own partition. The issue of how knowledge-base can be broken into partitions, and how these partitions should be dynamically reconfigured would make a dissertation in itself and should be an interesting subject for future research.

```
(a-assert "Taxonomy"
          '((:taxonomy (physical-objects (bacteria) (containers)))))

(a-assert "Slots"
          '((:slot in (physical-objects containers)
                   :comment "(in ?po ?c) = ?po is in the container ?c.")
            (:slot bacteria-in (containers sets)
                   :comment "(bacteria-in ?c ?s) =
                             The set of bacteria in ?c is ?s.")
            (:slot sterile (containers booleans)
                   :comment "(sterile ?c true) = ?c is sterile.")
            (:slot sealed (containers booleans)
                   :comment "(sealed ?c true) = ?c is sealed.")
            (:slot dead (physical-objects booleans)
```

```
                    :comment "(dead ?x true) = ?x is dead.")))

(a-assert "Rules"
          '((:rules containers
                    ;; This rule is really a type restriction on the slot
                    ;; bacteria-in -- the set of bacteria in a container
                    ;; must be a subset of the set of all bacteria:
                    ((bacteria-in ?c ?s) -> (imp-superset ?s bacteria))
                    ;; Any member of the set of bacteria in a container must be
                    ;; in the container:
                    ((bacteria-in ?c ?s) (member ?s ?b) -> (in ?b ?c))
                    ;; We are willing to create a frame for the set of
                    ;; bacteria in a container if we need one:
                    ((bacteria-in ?c ?s) <- (:forc ?s (bacteria-in ?c ?s)))
                    ;; A container is sterile if it's sealed and all the bacteria
                    ;; in it are dead:
                    ((sterile ?c true)
                     <-
                     (sealed ?c true)
                     (bacteria-in ?c ?bac)
                     (:in-own-partition (representative ?bac ?r))
                     (dead ?r true)))
            (:rules physical-objects
                    ;; Physical objects inherit the temperature of their
                    ;; container:
                    ((temperature ?po ?t) <- (in ?po ?c) (temperature ?c ?t)))
            (:rules bacteria
                    ;; How to kill a bacterium:
                    ((dead ?b true) <- (temperature ?b ?t)
                                       (:lisp (>= ?t 212))))))
```

Finally, we create a container and see if we can sterilize it.

```
(a-assert "Facts"
          '(;;A sealed container is heated to 212 degrees:
            (:create ?c can)
            (sealed ?c true)
            (temperature ?c 212)))

ASSERTING:  Facts

 Result:
   Binding:            ?c     --- can [can]

 Created frame:        can

 Insertions: 8          Rule applications: 14    Iterations: 0.
 Unifications: 0        Matches: 18.
 Frame insertions: 8    Frame accesses: 212
```

```
  (a-query "Is the container sterile ?"
          '((sterile can true)))

QUERYING:   Is the container sterile ?

 Query succeeded.

 Created frames:       can-bacteria-in-representative can-bacteria-in-selfset
                       can-bacteria-in

 Insertions: 35          Rule applications: 75    Iterations: 0.
 Unifications: 18        Matches: 107.
 Frame insertions: 67    Frame accesses: 1141
```

Thus, by reasoning about the representative (using rules which hold for any individual) we can draw valid conclusions about entire sets. In general the goal of reasoning with arbitrary objects is to reduce as much complex reasoning about quantification as possible to simpler frame based reasoning about individuals.

3.8 More Complex Reasoning About Sets of Similar Objects

More complex reasoning about sets often involves dynamically creating the set of objects satisfying a given path. In general, any predicate defines the set of objects for which the predicate is true. The traditional mechanism for creating this set is lambda abstraction.[8]

Algernon cannot by itself represent lambda abstraction; an auxiliary Lisp function is required. The assertions, and Lisp function, required to define lambda abstraction are shown below. The basis of the representation is the relation `setof` which links a set to a subset of it created by lambda abstraction. For example, we can create the set of people who like chocolate by:

```
  (a-assert "Chocaholics"
          '((:create ?c chocaholics)
            (setof people (:lambda ?x (loves ?x chocolate)) ?c)))
```

The special form `:lambda` is used to surround the path defining a set. It is required for technical reasons (it prevents the path from being affected by Algernon's pre-processor). We refer to the sets created using lambda abstraction as described sets.

The form `:bind-to-values` (used in the rule for `cardinality` below) binds a variable to the list of all values in a slot. For example, `(:bind-to-values ?list set member)` binds `?list` to the list of a members of `set`.

```
  (a-assert "Taxonomy"
          '((:taxonomy (sets (described-sets)))))

  (a-assert "Slots"
          '((:slot setof (sets nil described-sets)
                   :partition set-partition
                   :comment "(setof s1 (:lambda ?x path) s2) <->
```

[8]The utility of lambda abstraction in Algernon was first suggested by Chinatsu Aone.

```
                        s2 = {?x in s1 | path}")
         ;; description links a described set to its description:
         (:slot description (sets nil)
                :partition set-partition
                :comment "(description s1 (:lambda ?x path)) <->
                          s1 = {?x in (domain s1) | path}")
         ;; domain links a described set to its domain:
         (:slot domain (described-sets sets)
                :partition set-partition
                :backlink subset
                :comment "(domain s1 s2) <->
                          s1 = {?x in s2 | (description s1)}")
         (:slot cardinality (sets nil)
                :partition set-partition)))

(a-assert "Rules"
        '((:rules sets
                 ((setof ?s1 ?desc ?s2) -> (description ?s2 ?desc)
                                           (domain ?s2 ?s1))
                 ;; One can determine the cardinality of a set by
                 ;; counting its elements:
                 ((cardinality ?s ?n) <- (:bind-to-values ?list ?s member)
                                         (:bind ?n (length '?list))))
          ;; Finally we add the setof rule:
          (:rules described-sets
                 ((description ?s ?desc) (domain ?s ?dom)
                  ->
                  ;; The selfset of a frame is the set consisting
                  ;; exactly of that frame.
                  (:forc ?ss (selfset ?s ?ss))
                  (:rules ?ss (:lisp (build-setof-rule '?s '?dom '?desc)))))))
```

The key property of a described set is that any frame satisfying its predicate should be a member of the set. We use the Lisp function below to build the rule necessary to draw this deduction. For example, given a frame for `Tom` we could create the set of his children using:

```
(a-assert "Tom's children."
        '((:create ?c Tom-children)
          (setof people (:lambda ?x (child Tom ?x)) ?c)))
```

The membership rule for the set `Tom-children` would then be:

```
(member ?x Tom-children) <- (child Tom ?x)
```

However, consider the set of chocaholics above. One might expect that the membership rule for it would be:

```
(member ?x chocaholics) <- (loves ?x chocolate)
```

but this rule is illegal because its antecedent does not define an access path. Instead, Algernon would create the rule:

```
(member ?x chocaholics) <- (member people ?x) (loves ?x chocolate)
```

(this rule is created by the first case in the `cond` statement in the Lisp function below while the 'simple' case is covered by the second case).

```
(defun build-setof-rule (set domain description)
  (let ((free-var (second description))
        (path (third description)))
    (cond ((input-var? (frame (car path)))
           ;; If the first argument to the first predicate in path is a variable
           ;; then the 'obvious' rule would not define an access path, so we add
           ;; an additional member link:
           (preprocess `((member (:quote ,set) ,free-var)
                         <-
                         (member (:quote ,domain) ,free-var)
                         ,@path)))
          (t
           ;; Otherwise its simple:
           (preprocess `((member (:quote ,set) ,free-var)
                         <-
                         ,@path
                         (isa ,free-var (:quote ,domain))))))))
```

There are many possible application of lambda abstraction. One very simple application is shown below. First we define a situation:

```
(a-assert "Slots"
          '((:slot parent (people people))
            (:slot child (people people)
                   :inverse parent)
            (:slot likes (objects objects))))

(a-assert "For this example link the set people down to all frames for people."
          '((:rules people ((isa ?x people) -> (member people ?x)))))

(a-assert "Facts"
          '((:create ?tom Tom)
            (:create ?lisa Lisa)
            (:create ?fred Fred)
            (:create ?betty Betty)

            (child ?tom ?lisa)
            (child ?tom ?fred)
            (child ?tom ?betty)

            (:create ?c chocolate)
            (likes ?lisa ?c)
            (likes ?fred ?c)))
```

Then we can define some sets:

```
(a-query "Which of Tom's children like chocolate ?"
         '((child tom ?x)
           (likes ?x chocolate)))
```

```
QUERYING:   Which of Tom's children like chocolate ?

 Result 1:
   Binding:            ?x     --- fred [fred]

 Result 2:
   Binding:            ?x     --- lisa [lisa]

 Insertions: 0          Rule applications: 2     Iterations: 0.
 Unifications: 2        Matches: 5.
 Frame insertions: 4    Frame accesses: 67



  (a-query "How many people like chocolate ?"
           '((:forc ?s (setof people (:lambda ?x ((likes ?x chocolate))) ?s))
             (cardinality ?s ?n)))

QUERYING:   How many people like chocolate ?

 Result:
   Bindings:           ?n     --- 2
                       ?s     --- frame1 [nil]

 Created frames:      frame1-selfset frame1

 Insertions: 28         Rule applications: 41    Iterations: 0.
 Unifications: 13       Matches: 51.
 Frame insertions: 63   Frame accesses: 778



  (a-query "Do two of Tom's children like chocolate ?"
           '((:forc ?s (setof people (:lambda ?x ((child tom ?x)
                                                  (likes ?x chocolate))) ?s))
             (cardinality ?s 2)))

QUERYING:   Do two of Tom's children like chocolate ?

 Result:
   Binding:            ?s     --- frame2 [nil]

 Created frames:      frame2-selfset frame2

 Insertions: 27         Rule applications: 24    Iterations: 0.
 Unifications: 6        Matches: 28.
 Frame insertions: 46   Frame accesses: 581



  (a-query "Some interesting frames:"
           '((:show people)
             (setof people ?desc ?s)
             (:show ?s)))
```

```
QUERYING:   Some interesting frames:

People:
  name:       ((people) nil)
  isa:        (objects nil) (sets nil) (things nil)
              (isa ?x people) -> (member people ?x)
              (isa ?x people) -> (isa ?x physical-objects)
  imp-superset: (physical-objects nil)
  superset:   (physical-objects nil)
  wife:
              (wife ?x ?p1) -> (gender ?p1 female)
  husband:
              (husband ?x ?p1) -> (gender ?p1 male)
  member:     (betty nil) (fred nil) (tom nil) (lisa nil)
  setof:      (((lambda ?x ((child tom ?x) (likes ?x chocolate))) frame2) nil)
              (((lambda ?x ((likes ?x chocolate))) frame1) nil)
  subset:     (frame2 nil) (frame1 nil)


Frame2:
  isa:        (things nil) (frame2-selfset nil) (sets nil) (described-sets nil)
  description: ((lambda ?x ((child tom ?x) (likes ?x chocolate))) nil)
  domain:     (people nil)
  selfset:    (frame2-selfset nil)
  superset:   (people nil)
  cardinality: (2 nil)
  member:     (lisa nil) (fred nil)


Frame1:
  isa:        (things nil) (frame1-selfset nil) (sets nil) (described-sets nil)
  description: ((lambda ?x ((likes ?x chocolate))) nil)
  domain:     (people nil)
  selfset:    (frame1-selfset nil)
  superset:   (people nil)
  cardinality: (2 nil)
  member:     (lisa nil) (fred nil)

 Result 1:
   Bindings:            ?s     --- frame2 [nil]
                        ?desc   --- (lambda ?x
                                      ((child [nil] tom [tom] ?x)
                                       (likes [nil] ?x chocolate [chocolate])))

 Result 2:
   Bindings:            ?s     --- frame1 [nil]
                        ?desc   --- (lambda ?x
                                      ((likes [nil] ?x chocolate [chocolate])))

 Insertions: 0          Rule applications: 1     Iterations: 0.
 Unifications: 1        Matches: 2.
 Frame insertions: 2    Frame accesses: 33
```

3.9 Controlled Non-monotonicity

Many 'real world' problems require degenerate types of non-monotonic reasoning which involve only a few facts changing in a predictable way. Such problems are considerably simpler than (and quite possibly orthogonal to) the general problem of non-monotonic reasoning. We examine several such problems in the next three sections, and consider the general problem of non-monotonic reasoning in section 3.12.

Consider the problem of building a "counter". Assume we have a collection of rules which fire to "increment" the counter, and at any given time we have to be able to return the current count. Intuitively one would expect that a reasonably principled solution to such problems could be devised without a general theory of non-monotonic reasoning. The intuition behind our counter implementation in Algernon is to confine the non-monotonicity so that only a few slots change non-monotonically, in a carefully controlled way, while the rest of the knowledge-base behaves more reasonably.

A first cut at a counter might look something like:

```
((increment counter true)
 (current-count counter ?n)
 ->
 (:clear-slot counter current-count)
 (:bind ?m (+ ?n 1))
 (current-count counter ?m))
```

The first problem with this rule is that it will fire only the first time `(increment counter true)` is asserted (since after the first time this will not be a new fact and if-added rules will not be applied). A possible fix would be to add a clause to the consequent:

```
((increment counter true)
 (current-count counter ?n)
 ->
 (:clear-slot counter current-count)
 (:clear-slot counter increment)
 (:bind ?m (+ ?n 1))
 (current-count counter ?m))
```

Unfortunately there is another problem. The rule will *complete* (see section 2.3.5) and add the infinite looping rule:

```
((current-count counter ?n)
 ->
 (:clear-slot counter current-count)
 (:clear-slot counter increment)
 (:bind ?m (+ ?n 1))
 (current-count counter ?m))
```

The solution to this problem is to make the increment predicate the only clause in the antecedent:

```
((increment counter true)
 ->
 (current-count counter ?n)
 (:clear-slot counter current-count)
 (:clear-slot counter increment)
 (:bind ?m (+ ?n 1))
 (current-count counter ?m))
```

The antecedent is now of length one (and so the rule cannot be completed), but when the rule fires and the consequent is asserted, there will be no known value for the variable `?n`. Algernon will therefore look up the `current-count` of the counter before asserting the rest of the consequent (thus the predicate `(current-count counter ?n)` still behaves as though it were in the antecedent).

Only the slots `increment` and `current-count` change non-monotonically. The rest of the knowledge-base should generally behave monotonically (if the other rules are written so that they always fire for values of the counter greater or equal to some threshold — rules which fire only for a certain values of the counter will necessarily 'propagate' the non-monotonicity of the counter to the rest of the knowledge-base).

3.10 Building a Finite State Machine

Many reasoning problems can be broken down into a sequence of smaller steps. For example in a diagnosis problems one might want to find a possible diagnosis, look for a way to test it, run the test, and then, if the test fails, loop back to look for another diagnosis. Situations requiring a sequence of reasoning steps often come up in building expert systems. The usual solution is to use an ad hoc collection of variables to control reasoning. A cleaner solution is to build the expert system on top of a finite state machine. This section shows how to build a finite state machine in Algernon. The next will show how to use it to implement a simple expert system.

The finite state machine, by itself, is completely monotonic. As the machine "runs" we create new times and calculate the state of the machine at each time. In order to build on top of the finite state machine it will be necessary (in the next section) to add a pointer to the 'current time'. This pointer will introduce 'controlled' non-monotonicity (see section 3.9).

```
(a-assert "Taxonomy"
         '((:taxonomy (objects (times fsm-time)
                               (states (non-final-states)
                                       (final-states final-state))
                               (inputs)))))

(a-assert "New slots"
         '((:slot next-time (times times)
                  :cardinality 1
                  :comment "(next-time t1 t2) = The time after t1 is t2.")
           (:slot state (times states)
                  :cardinality 1
                  :comment "(state t s) = The fsm is in state s at time t.")
           (:slot input (times inputs)
                  :cardinality 1
                  :comment "(input t i) = The input to the fsm at time t is i.")
           (:slot next-state (states inputs states)
                  :comment "(next-state s1 i s2) =
                           The next state after s1 under input i is s2.")))

(a-assert "Time"
         '((:rules times
                   (;; There's always next time:
                    (next-time ?t1 ?t2)
```

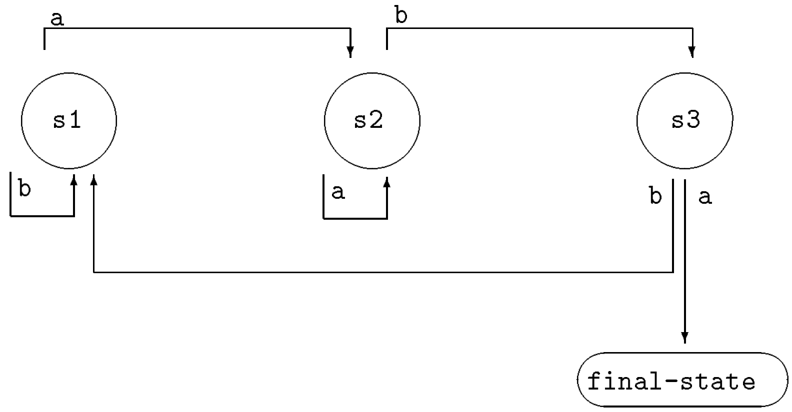


Figure 3.2: Finite state machine to recognize aba.

```
                    <-
                    (:forc ?t2 (name ?t2 (:quote (fsm-time)))
                               (next-time ?t1 ?t2))))))

(a-assert "Transitions"
          '((:rules times
                   ;; When we know everything about current time
                   ;; then we can go to next time:
                   ((state ?t1 ?s1) (isa ?s1 non-final-states)
                    (next-time ?t1 ?t2)
                    (input ?t1 ?i1)
                    (next-state ?s1 ?i1 ?s2)
                    ->
                    (state ?t2 ?s2)))))

(a-assert "User interface"
          '((:rules times
                  ((state ?t ?s)
                   ->
                   (:lisp (format t "~% State is ~(~a~).~%" '?s))))
            (:rules final-states
                   ((state ?t ?s)
                    ->
                    (:lisp (format t "~%Reached final state.~%"))))
            (:rules times
                   ((input ?t ?i) <- (:ask (input ?t ?i)))))))
```

We then have only to add the transition rules and run the finite state machine. In this case the machine looks for the the pattern aba (*i.e.*, it recognizes the pattern *aba). The finite state machine is shown in figure 3.2.

```
(a-assert "States"
          '((:taxonomy (non-final-states s1 s2 s3))
            (:taxonomy (inputs a b))))
(a-assert "Transitions"
          '((next-state s1 a s2)
            (next-state s2 b s3)
            (next-state s3 a final-state)
            (next-state s1 b s1)
            (next-state s2 a s2)
            (next-state s3 b s1)))
```

A sample "run" of the machine is shown below:

```
  (a-assert "Initialize and run."
            '((state fsm-time s1)))

ASSERTING:  Initialize and run.

 State is s1.

 Give me a value for ?i1 in (input fsm-time ?i1) [possible values: b a] b

 State is s1.

 Give me a value for ?i1 in (input fsm-time1 ?i1) [possible values: b a] a

 State is s2.

 Give me a value for ?i1 in (input fsm-time2 ?i1) [possible values: b a] b

 State is s3.

 Give me a value for ?i1 in (input fsm-time3 ?i1) [possible values: b a] b

 State is s1.

 Give me a value for ?i1 in (input fsm-time4 ?i1) [possible values: b a] a

 State is s2.

 Give me a value for ?i1 in (input fsm-time5 ?i1) [possible values: b a] b

 State is s3.

 Give me a value for ?i1 in (input fsm-time6 ?i1) [possible values: b a] a

 State is final-state.

 Assert succeeded.

 Created frames:      s1-selfset s2-selfset s3-selfset final-state-selfset
                      fsm-time7 fsm-time6 fsm-time5 fsm-time4 fsm-time3
```

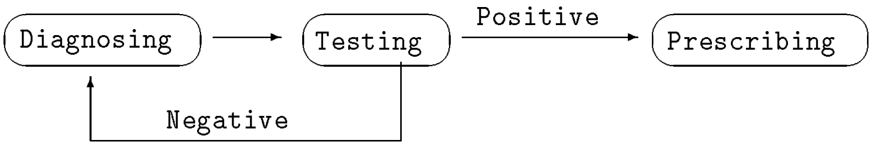


Figure 3.3: Finite state machine for generate and test cycle.

```
                    fsm-time2 fsm-time1

Insertions: 89          Rule applications: 190   Iterations: 0.
Unifications: 34        Matches: 345.
Frame insertions: 166   Frame accesses: 3051
```

3.11 A Generate and Test Cycle

Using the finite state machine from the last section we can build a simple expert system. The system will classify patients according to whether they have the flu, mononucleosis, or pneumonia (using very simple rules). Each disease is indicated by certain symptoms. If the symptoms are present then the system orders the test for the disease. If the test succeeds then the system outputs a diagnosis and a 'prescription'. The disease flu is an exception in that there is no test for it — flu is diagnosed if the patient has the requisite symptoms, but no other diagnosis is possible.

The finite state machine has three states `Diagnosing`, `Testing`, and `Prescribing`. In the `Diagnosing` state we collect possible diagnoses, and go to the `Testing` state. In the `Testing` state we test the diagnoses. If the test for some possible diagnosis is positive then we go to the state `Prescribing`. If all tests are negative then we return to the `Diagnosing` state. This control strategy implements a breadth first search for a diagnoses. The finite state machine is shown in figure 3.3.

There are actually two sources of non-monotonicity in this example. First, the value in the slot `current-time` changes as the finite state machine advances. As discussed with respect to the counter in section 3.9, this sort of non-monotonicity is controlled and relatively tame. The second source of non-monotonicity is the use of the form `:unp` in several of the rules. `(:unp p)` succeeds iff a query of `p` fails.

```
(a-assert "Taxonomy"
          '((:taxonomy
              (objects
                (times fsm-time)
                (states (non-final-states diagnosing testing prescribing)
                        (final-states final-state))
                (symptoms low-fever high-fever cough tiredness)
                (diseases flu mono pneumonia)
                (tests mono-test xray)
```

```
                (results positive negative)
                (prescriptions rest lots-of-rest penicillin)))))

(a-assert "New slots"
         '((:slot next-time (times times)
                  :cardinality 1
                  :comment "(next-time t1 t2) = The time after t1 is t2.")
           (:slot current-time (contexts times)
                  :cardinality 1
                  :comment "(current-time c t) =
                           The current time in context c is t.")
           (:slot state (times states)
                  :cardinality 1
                  :comment "(state t s) = The fsm is in state s at time t.")
           (:slot next-state (states times states)
                  :comment "(next-state s1 i s2) =
                           The next state after s1 under input i is s2."))
           (:slot has-symptom (people symptoms)
                  :comment "(has-symptom p s) = p has symptom s.")
           (:slot diagnosis (people diseases)
                  :comment "(diagnosis p d) = The diagnosis for p is d.")
           (:slot probable-diagnosis (people diseases)
                  :comment "(probable-diagnosis p d) =
                           The diagnosis for p is probably d.")
           (:slot patient (contexts people)
                  :cardinality 1
                  :comment "(patient c p) = The patient in context c is p.")
           (:slot test (diseases tests)
                  :comment "(test d t) = The test for d is t.")
           (:slot result (tests people results)
                  :comment "(result t p r) =
                           The result of test t on person p is r.")
           (:slot prescription (people prescriptions)
                  :comment "(prescription p prescript) =
                           The prescription for p is prescript.")))

(a-assert "Times"
         '((:rules contexts
                   ;; There's a first time for everything:
                   ;; (this saves us from having to assert a first time
                   ;; in each new context).
                   ((current-time ?cc fsm-time)
                    <-
                   (:unp (current-time ?cc ?t))))
           (:rules times
                   ;; And there's always next time:
                   ((next-time ?t1 ?t2)
                    <-
                    (:forc ?t2 (name ?t2 (:quote (fsm-time)))
                               (next-time ?t1 ?t2))))))
```

The transition rule is just like in the finite state machine except that:

- It changes the `current-time` in the `current-context`.

- It will not fire to create a next state for any state except the state at the current time (this avoids some strange bugs involving old states executing transitions ...).

```
(a-assert "Transitions"
         '((:rules times
                   ;; Transitions:
                   ((state ?t1 ?s1)
                    (isa ?s1 non-final-states)
                    (next-time ?t1 ?t2)
                    (next-state ?s1 ?t1 ?s2)
                    (current-context global-context ?cc)
                    (current-time ?cc ?t1)
                    ->
                    (:clear-slot ?cc current-time)
                    (current-time ?cc ?t2)
                    ;; Output
                    (:lisp (progn (format t "~% State is now ~(~a~).~%" '?s2)
                                  t))
                    (state ?t2 ?s2)))))

(a-assert "Symptoms"
         '((:rules people
                   ((temperature ?x ?t)
                    <-
                    (:ask (temperature ?x ?t)))
                   ;;
                   ((has-symptom ?x low-fever)
                    <-
                    (temperature ?x ?t)
                    (:lisp (and (> ?t 99) (< ?t 102))))
                   ;;
                   ((has-symptom ?x high-fever)
                    <-
                    (temperature ?x ?t)
                    (:lisp (or (= ?t 102) (> ?t 102))))
                    ;;
                   ((has-symptom ?x tiredness)
                    <-
                    (:ask (has-symptom ?x tiredness)))
                   ;;
                   ((has-symptom ?x cough)
                    <-
                    (:ask (has-symptom ?x cough))))))

(a-assert "Tests"
         '((test mono mono-test)
           (test pneumonia xray)
           ;;
           ;; Performing a test:
           (:rules tests
                   ((result ?test ?patient ?result)
                    <-
                    ;; Only instruct user to perform test if no value
                    ;; is known at all.  The construction "(:unp (:retrieve p))"
```

```
;; succeeds iff p is not currently in the knowledge-base:
(:unp (:retrieve (result ?test ?patient ?any-result)))
(:lisp (progn () (format t "~% Apply test ~(~a~).~%" '?test)
                 t))
(:ask (result ?test ?patient ?result)))

;; These last two rules should be handled in a more
;; general way:
((result ?test ?patient positive)
 ->
 (not (result ?test ?patient negative)))
((result ?test ?patient negative)
 ->
 (not (result ?test ?patient positive))))))
```

For each state in the finite state machine we have three types of rules:

1. Backchaining rules to fill the slot for the state.
2. Forward chaining rules to print some tracing information.
3. Transition rules to calculate the next state.

```
(a-assert "Diagnosing"
          '((:rules people
             ;;
             ;; Finding the probable-diagnosis:
             ((probable-diagnosis ?p flu)
              <-
              (state (current-time (current-context global-context)) diagnosing)
              (has-symptom ?p low-fever))
             ((probable-diagnosis ?p flu)
              <-
              (state (current-time (current-context global-context)) diagnosing)
              (not (diagnosis ?p pneumonia))
              (has-symptom ?p high-fever))
             ((probable-diagnosis ?p mono)
              <-
              (state (current-time (current-context global-context)) diagnosing)
              (has-symptom ?p low-fever)
              (has-symptom ?p tiredness))
             ((probable-diagnosis ?p pneumonia)
              <-
              (state (current-time (current-context global-context)) diagnosing)
              (has-symptom ?p high-fever)
              (has-symptom ?p cough))
             ;;
             ;; Output
             ((probable-diagnosis ?p ?x)
              ->
              (:lisp (format t "~% ~(~a~) may have ~(~a~).~%" '?p '?x))))
            ;;
            ;; Transition rule:
            (:rules times
```

```
             ((state ?t diagnosing)
              (probable-diagnosis (patient (current-context global-context)) ?d)
              ->
              (next-state diagnosing ?t testing)))))

(a-assert "Testing"
          '((:rules people
             ;;
             ;; Finding the diagnosis:
             ((diagnosis ?p ?d)
              <-
              (state (current-time (current-context global-context)) testing)
              (probable-diagnosis ?p ?d)
              (result (test ?d) ?p positive))
             ((not (diagnosis ?p ?d))
              <-
              (state (current-time (current-context global-context)) testing)
              (probable-diagnosis ?p ?d)
              (result (test ?d) ?p negative))
             ;; No test exists:
             ((diagnosis ?p ?d)
              <-
              (state (current-time (current-context global-context)) testing)
              (probable-diagnosis ?p ?d)
              (:unp (test ?d ?test))
              ;; And everything else has been ruled out:
              (:all-paths ((probable-diagnosis ?p ?d2) (:neq ?d ?d2))
                          ((not (diagnosis ?p ?d2)))))
             ;;
             ;; Output
             ((diagnosis ?p ?x)
              ->
              (:lisp (format t "~% ~(~a~) has ~(~a~).~%" '?p '?x))))
            ;;
            ;; Transition rules:
            (:rules times
             ((state ?t testing)
              (diagnosis (patient (current-context global-context)) ?d)
              ->
              (next-state testing ?t Prescribing))
             ((state ?t testing)
              (patient (current-context global-context) ?patient)
              (not (diagnosis ?patient ?d))
              ->
              ;; If all probable-diagnoses fail then it's back to
              ;; the drawing board:
              (:all-paths ((probable-diagnosis ?patient ?diag))
                          ((not (diagnosis ?patient ?diag))))
              (next-state testing ?t diagnosing)))))

(a-assert "Prescribing"
          '((:rules people
             ;;
             ;; Finding the prescription:
```

```
((prescription ?p rest)
 <-
 (state (current-time (current-context global-context)) Prescribing)
 (diagnosis ?p flu))
((prescription ?p lots-of-rest)
 <-
 (state (current-time (current-context global-context)) Prescribing)
 (diagnosis ?p mono))
((prescription ?p  penicillin)
 <-
 (state (current-time (current-context global-context)) Prescribing)
 (diagnosis ?p pneumonia))
;;
;; Output
((prescription ?p ?x)
 ->
 (:lisp (format t "~% The prescription for ~(~a~) is ~(~a~).~%"
                '?p '?x))))
;;
;; Transition rule:
(:rules times
 ((state ?t Prescribing)
  (prescription (patient (current-context global-context)) ?p)
  ->
  (next-state Prescribing ?t final-state))))))
```

Finally, two runs of the system are shown below. One interesting feature of this system is that since there is no test for `flu`, it is diagnosed only by process of elimination; flu becomes the final diagnosis iff it is the only possible diagnosis. In the first run below, the system initially thinks John may have Pneumonia. When the test for Pneumonia fails, the system returns to the diagnosing state and decides John may have flu. It then decides John does have flu (since it is the only possible diagnosis). In the second run, the system thinks John may have flu or mononucleosis. When the test for mononucleosis fails, it concludes John must have flu.

```
  (a-assert "Diagnosing John."
            '((:create ?con context)
              (:create ?j John)
              (:clear-slot global-context current-context)
              (current-context global-context ?con)
              (patient ?con ?j)

              (state (current-time ?con) diagnosing)))

ASSERTING:   Diagnosing John.

 Give me a value for ?t in (temperature john ?t): 104
 Is it true that (has-symptom john cough)?  (y or n)  y
 john may have pneumonia.

 State is now testing.
```

```
Apply test xray.
Is it true that (result xray (john negative))?  (y or n)  y
State is now diagnosing.

john may have flu.

State is now testing.

john has flu.

State is now prescribing.

The prescription for john is rest.

State is now final-state.


Result:
  Bindings:            ?$x1   --- fsm-time [fsm-time]
                       ?j     --- john [john]
                       ?con   --- context [context]

Created frames:       final-state-selfset fsm-time5 fsm-time4-selfset
                      prescribing-selfset fsm-time4 fsm-time3-selfset
                      fsm-time3 fsm-time2-selfset fsm-time2 fsm-time1-selfset
                      testing-selfset fsm-time1 fsm-time-selfset
                      diagnosing-selfset john-selfset john context

Insertions: 128         Rule applications: 336   Iterations: 24.
Unifications: 59        Matches: 795.
Frame insertions: 461   Frame accesses: 5767



ASSERTING:   Diagnosing John.

Give me a value for ?t in (temperature john ?t): 101

john may have flu.
Is it true that (has-symptom john tiredness)?  (y or n)  y
john may have mono.

State is now testing.

Apply test mono-test.
Is it true that (result mono-test (john negative))?  (y or n)  y
john has flu.

State is now prescribing.

The prescription for john is rest.

State is now final-state.
```

```
Result:
  Bindings:           ?$x1   --- fsm-time [fsm-time]
                      ?j     --- john [john]
                      ?con   --- context [context]

Created frames:      final-state-selfset fsm-time3 fsm-time2-selfset
                     prescribing-selfset fsm-time2 fsm-time1-selfset
                     testing-selfset fsm-time1 fsm-time-selfset
                     diagnosing-selfset john-selfset john context

Insertions: 103        Rule applications: 254   Iterations: 19.
Unifications: 49       Matches: 547.
Frame insertions: 325  Frame accesses: 4234
```

3.12 Reasoning with Different Views — Approaching Default Reasoning

Default reasoning is a hard problems and much has been written about it [Etherington, 88, Ginsberg, 87, Reiter, 87]. One issue which has received little attention until recently is the issue of efficiency. Most proposals for default reasoning are computationally intractable, while in common sense reasoning, defaults seem to be used to speed up inference (see [Brachman, 90] p. 1090, and [Cadoli & Lenzerini, 90]). One possible application of the access limitations in Algernon (particularly the association of rules with sets of objects) is in the development of efficient approaches to default reasoning. The example in this section represents the beginning of what could be a major area of future work.

Consider an automobile mechanic getting into his car to drive to work. By default he should expect to be able to simply put his key in the ignition and start the car. But consider what happens if we try to represent this default using default logic [Reiter, 80]. We get a rule something like:

$$\frac{on(ignition) : starts(car)}{starts(car)}$$

which requires the mechanic to apply everything he knows about automobiles to make sure it is consistent to assume his car will start, before assuming it will start. Similar problems occur with other approaches to default reasoning.

The intuitions behind Access Limited Logic suggest a different approach. One can begin a reasoning problem by taking simple *views* of the objects in the problem. Such simple views give us access to only a small part of what we know about the objects. The automobile mechanic probably begins with a view of his car which says "you turn the key and it starts." If the simple views prove inadequate then one can switch to more complex views. A set of views can prove inadequate if they contradict each other, contradict observations (*e.g.*, he turns the key and nothing happens), or do not describe the situation in adequate detail to answer some question.

To illustrate the idea of views and how they can be used, consider the (in)famous Nixon diamond. We know that Nixon is a Republican and a Quaker. Further, we know that most Quakers are pacifists and most Republicans are not. The question is, "Is Nixon a pacifist ?".

We represent this problem by creating a simple view of Republicans which says that they are not pacifists, and a simple view of Quakers which says that Quakers are pacifists. We also create a more complex view of Quakers which says that there are actually two kinds of Quakers, and only one kind of Quakers are pacifists.

I am not claiming here that the use of views solves all problems in default reasoning. Many of the familiar problems in default reasoning re-occur when using views. For example, in the case of the Nixon diamond it is not at all clear whether we should take a more complex view of Quakers or a more complex view of Republicans. Further, if several more complex views are available then it is not at all clear which one to take (in the example below we avoid this problem by only putting one complex view in the knowledge-base). Notice, however, that the issues in selecting between views in default reasoning are quite similar to the issues in selecting between model fragments in qualitative model building (see section 4.3). The problem of building a qualitative model of a complex system by selecting appropriate model fragments describing the objects in the system is an active area of research in the qualitative reasoning community [Falkenhainer and Forbus, 1988, Iwasaki, 90]. Some of the results of this work might eventually be applicable to the problem of selecting views of objects for default reasoning.

A view in Algernon is always linked to the set of objects which are described by the view, and the rules which define what the view means are stored with this set. Thus in the example below we create the view `simple-quaker-view` and link it to the set `simple-quakers`. We then add a rule asserting that any object viewed using the view `simple-quaker-view` is a member of the set `simple-quakers`. Finally, we associate with the set `simple-quakers` a rule saying that all simple quakers are pacifists.

As discussed above, one of the hardest problems in any approach to default reasoning is deciding what defaults to give up when a contradiction occurs. In Algernon facts (and rules) are tagged with the assumptions they are believed under. Further, at any time an assumption can be believed or not believed. Any query to Algernon succeeds if it is believed under some set of assumptions which are currently believed. When a contradiction is detected, enough assumptions must be given up to resolve the contradiction (*i.e.*, enough assumptions must be dropped so that both sides of the contradiction are no longer believed). Algernon currently uses two simple heuristics in deciding which assumptions to drop:

1. It prefers to drop assumptions of form `(take-view frame view)` when a more complex view exists. This heuristic is implemented by the 'view-changer'. In the tracing output below, recommendations based on this heuristic are tagged with the message: `View-changer recommends dropping assump: ...`.

2. If the assumptions a_1 and a_2 lead to a contradiction, and the assumption a_2 depends on the assumption a_1 (*i.e.*, the assumption a_2 is only believed under the assumption a_1) then Algernon will prefer to drop a_2 (such cases often arise in proofs by contradiction when we assump p and, within the context of this assumption, assume q). In the trace output below, recommendations based on this heuristic are tagged with the message: `Dependent-assumps recommends dropping assump: ...`.

First we add the sets, slots and rules which define views:

```
(a-assert "Taxonomy"
```

```
         '((:taxonomy (objects (views)))))

(a-assert "Slots"
         '((:slot view (sets views)
                  :comment "(view s v) = A view of elements of s is v.")
           (:slot simple-view (sets views)
                  :cardinality 1
                  :comment "(simple-view s v) =
                           The simple view of elements of s is v.")
           (:slot more-detailed-view (views views)
                  :cardinality 1
                  :comment "(more-detailed-view v1 v2) =
                           A more detailed view than v1 is v2.")
           (:slot take-view (things views)
                  :cardinality 1
                  :comment "(take-view x v) = Take view v of x.")
           (:slot rule-set (views sets)
                  :cardinality 1
                  :comment "(rule-set v s) =
                           The rules for view v are associated with set s.")))

(a-assert "Rules"
         '((:rules things
                   ;; Take the simple view by default:
                   ((isa ?x ?set)
                    ->
                    (simple-view ?set ?view) (:assume (take-view ?x ?view)))
                   ;; When a view fails take a more complex one:
                   ((not (take-view ?x ?v))
                    ->
                    (more-detailed-view ?v ?v2) (:assume (take-view ?x ?v2)))
                   ;; If you take a view then you get its rules:
                   ((take-view ?x ?view)
                    ->
                    (rule-set ?view ?s)
                    (isa ?x ?s)))
           (:rules views
                   ;; A simple view of (the objects in) a set is a view of
                   ;; (the objects in) the set:
                   ((simple-view ?s ?v) -> (view ?s ?v)))))
```

Then the assertions to set up the Nixon diamond:

```
(a-assert "Taxonomy"
         '((:taxonomy (people (republicans)
                             (quakers (pacifist-quakers)
                                      (non-pacifist-quakers))))))

(a-assert "Slots"
         '((:slot pacifist (people booleans)
                  :cardinality 1
                  :comment "(pacifist p true) = p is a pacifist.")))

(a-assert "Simple view of quakers."
```

```
          '((:create ?v simple-quaker-view)
            (simple-view quakers ?v)
            (:create ?s simple-quakers)
            (rule-set ?v ?s)
            (:rules ?s
                    ;; In the simple view quakers are always pacifists:
                    ((pacifist ?x true) <-))))

(a-assert "Simple view of republicans."
          '((:create ?v simple-republican-view)
            (simple-view republicans ?v)
            (:create ?s simple-republicans)
            (rule-set ?v ?s)
            (:rules ?s
                    ;; In the simple view republicans are not pacifists:
                    ((not (pacifist ?x true)) <-))))

(a-assert "Complex view of quakers."
          '((:create ?v2 complex-quaker-view)
            (view quakers ?v2)
            (more-detailed-view simple-quaker-view ?v2)
            (:create ?s complex-quakers)
            (rule-set ?v2 ?s)
            ;; There are actually two kinds of quakers:
            (:contra-positive
             ;; (This rule actually represents the disjunction that every
             ;; quaker is either a very-quaker or a moderate-quaker.)
             (:rules ?s
                     ((isa ?x pacifist-quakers) <- (not (isa ?x non-pacifist-quakers)))))
            (:rules ?s
                    ;; By default assume pacifist-quakers:
                    ((isa ?x ?s) -> (:assume (isa ?x pacifist-quakers))))))

;; pacifist-quakers are pacifists, but we say nothing about non-pacifist-quakers:
(a-assert "pacifist-quakers"
          '((:rules pacifist-quakers
                    ((pacifist ?x true) <-))))

(a-assert "Nixon"
          '((:create ?n Nixon)
            (isa ?n republicans)
            (isa ?n quakers))))
```

Then we can ask some questions about Nixon. Notice that we initially take the simple views and do not give them up until an explicit contradiction is detected. Thus in the 'expected case' (where the defaults are correct) reasoning should be efficient since the complex view is inaccessible. For the operations which lead to contradictions we include some partial traces of contradiction resolution.

```
(a-query "What views do we take of Nixon ?"
         '((take-view Nixon ?v)))
```

```
QUERYING:   What views do we take of Nixon ?

 Result 1:
   Binding:            ?v     --- simple-quaker-view [simple-quaker-view]

  Assumption:         ((take-view nixon simple-quaker-view))

 Result 2:
   Binding:            ?v     --- simple-republican-view [simple-republican-view]

  Assumption:         ((take-view nixon simple-republican-view))

 Insertions: 0          Rule applications: 1     Iterations: 1.
 Unifications: 1        Matches: 3.
 Frame insertions: 2    Frame accesses: 41



  (a-query "Is Nixon a pacifist ?"
          '((pacifist Nixon true)))

QUERYING:   Is Nixon a pacifist ?

 Result:
  Assumption:         ((take-view nixon simple-quaker-view))

 Insertions: 1          Rule applications: 3     Iterations: 2.
 Unifications: 2        Matches: 3.
 Frame insertions: 3    Frame accesses: 82



  (a-query "Is Nixon not a pacifist ?"
          '((not (pacifist Nixon true))))

QUERYING:   Is Nixon not a pacifist ?

  ** Beginning raa Trace **
 Contradiction:       (not (pacifist nixon true))
                      (pacifist nixon true)

   (not (pacifist nixon true)) supported by assumptions:
     ((take-view nixon simple-republican-view))
   (pacifist nixon true) supported by assumptions:
     ((take-view nixon simple-quaker-view))
 View-changer recommends dropping assump: ((take-view nixon simple-quaker-view)).

 Dropping assumption: (take-view nixon simple-quaker-view).
 Asserting its negation: (not (take-view nixon simple-quaker-view)).
  ** End raa Trace **

 Result:
  Assumption:         ((take-view nixon simple-republican-view))
```

```
 Insertions: 5           Rule applications: 11    Iterations: 0.
 Unifications: 2         Matches: 28.
 Frame insertions: 22    Frame accesses: 311



  (a-query "What views do we take of Nixon ?"
          '((take-view Nixon ?v)))

QUERYING:   What views do we take of Nixon ?

 Result 1:
   Binding:             ?v     --- complex-quaker-view [complex-quaker-view]

  Assumption:         ((take-view nixon complex-quaker-view)
                       (take-view nixon simple-republican-view))

 Result 2:
   Binding:             ?v     --- simple-republican-view [simple-republican-view]

  Assumption:         ((take-view nixon simple-republican-view))

 Insertions: 0           Rule applications: 1     Iterations: 1.
 Unifications: 1         Matches: 3.
 Frame insertions: 2     Frame accesses: 56



  (a-query "Is Nixon a pacifist ?"
          '((pacifist Nixon true)))

QUERYING:   Is Nixon a pacifist ?

  ** Beginning raa Trace **
 Contradiction:       (pacifist nixon true)
                      (not (pacifist nixon true))

   (pacifist nixon true) supported by assumptions:
     ((isa nixon pacifist-quakers) (take-view nixon complex-quaker-view)
      (take-view nixon simple-republican-view))
   (not (pacifist nixon true)) supported by assumptions:
     ((take-view nixon simple-republican-view))
 View-changer recommends dropping assumps: nil.
 ...
 Dependent-assumps recommends dropping assump: ((isa nixon pacifist-quakers)).

 Dropping assumption: (isa nixon pacifist-quakers).
 Asserting its negation: (not (isa nixon pacifist-quakers)).
  ** End raa Trace **

 *Query failed.*

 Insertions: 3           Rule applications: 6     Iterations: 0.
```

```
 Unifications: 2         Matches: 13.
 Frame insertions: 9     Frame accesses: 207



  (a-query "Is Nixon not a pacifist ?"
           '((not (pacifist Nixon true)))))

QUERYING:   Is Nixon not a pacifist ?

 Result:
  Assumption:          ((take-view nixon simple-republican-view))

 Insertions: 0           Rule applications: 2     Iterations: 1.
 Unifications: 2         Matches: 1.
 Frame insertions: 2     Frame accesses: 44
```

Chapter 4

Systems

This chapter presents three examples of non-trivial systems which have been built using Algernon. This chapter is unique in that most of the work presented here is not mine. There is good reason for this. It is easy to build systems using your own knowledge representation system; It is quite another thing for others to be able to build systems using your knowledge representation system. Therefore, the work in this chapter is presented as empirical evidence in support of the claim that Algernon is a useful knowledge representation system.

The first system below is an expert system for diagnosing headaches built by Ben Kuipers. The second system is an implementation of Pearl's [Pearl, 88] theory of probability networks by Clif Day. The final system is an implementation of Qualitative Process Theory [Forbus, 84] built by Adam Farquhar and myself.

4.1 The Headache Protocol

The headache expert system is an implementation of the headache protocol of [Komaroff et al., 77]. The point of including it here is not to illustrate the details of diagnosing a headache, but to demonstrate the use of Algernon. Thus the reader is encouraged to ignore the medical details.

The headache protocol poses some definite challenges for a rule based system. The protocol consists of a series of questions which are to be posed by a nurse to a patient. Depending on the answers, sets of questions may be skipped, or, in some cases, the protocol may be immediately terminated and the patient referred to a doctor. Further, certain answers in the protocol are marked with a 'color' (literally on the sheet used by the nurse, and somewhat more figuratively in an expert system), and the final diagnosis may depend on the count of answers of a particular color. Careful control of inference is required to guarantee faithfulness to the protocol. The techniques used are discussed in the next two section.

Much of the explanatory text below is adapted from text written by Ben Kuipers. The Algernon code for the headache expert system is given in the appendix section A.1. The system has been tested on a number of synthesized cases, and successfully produced the behavior and conclusions of the protocol.

4.1.1 Controlling Inference using Set Membership

The questions in the headache protocol are organized into 'blocks' of questions. These blocks form a lattice (a loop-free transition graph). We can thus use sets, and a lattice of set-containment relations, to control the sequence of reasoning steps.

```
(a-assert "Blocks are subsets of Patients"
          '((:taxonomy (Patients
                        (Block1)
                        (Block2)
                        (Block3)
                        (Block4)
                        (Block5)
                        (Block6)
                        . . .
                        (Refer)
                        (Done)))))
```

The rules needed to trigger the questions for a block are associated with the block and thus are not accessible for a patient until the protocol has progressed far enough so that the patient becomes a 'member' of the block. Reasoning in a block is invoked by a forward-chaining rule triggered off membership in the block, after which it asserts membership in the next block.

```
((isa ?p Block1) (inference-for-block1 ?p ?val) -> (isa ?p Block2))
```

More generally, the inference within a block can determine which block membership assertion is made, and hence where the control goes. For example:

```
(:rules Block5

  ((isa ?p Block5) (refill ?p False) -> (isa ?p Block6))
  ((isa ?p Block5) (refill ?p True)  -> (isa ?p Done))

  ((refill ?p False) <- (:unp (refill ?p True)))

  ((refill ?p True)
   <-
   (definite-Dx-last-two-years ?p True)
   (on-protocol-acceptable-regiment ?p True)
   (acceptable-relief ?p True)
   (seeks-refill ?p True)))
```

4.1.2 Controlling Sequence using Access Path Order

Within a single block, we control the sequence of a set of questions using the order specified in an access path. The following rules are associated with a set named *Block6*, and the first one is invoked when membership in *Block6* is asserted. The rule's antecedent asks a sequence of questions, then asserts membership in the next block.

```
(:rules Block6
```

```
((isa ?p Block6)
 (flashes/spots-before-eyes-at-onset ?p ?tv1)
 (one-eye-red/tearful ?p ?tv2)
 (weakness-of-hand/arm/leg ?p ?tv3)
 (difficulty-speaking ?p ?tv4)
 ->
 (isa ?p Block7))

((flashes/spots-before-eyes-at-onset ?p True) -> (Dx ?p vascular-HA))
((one-eye-red/tearful ?p True)                -> (count-blues ?p True))
((weakness-of-hand/arm/leg ?p True)           -> (reds ?p True))
((difficulty-speaking ?p True)                -> (reds ?p True)))
```

4.1.3 Boolean Ask Slots

Where there are many slots that take boolean values, it is convenient to write a special set of rules to handle the user-interface for them.

If a slot is in the set named *Ask-slots*, it inherits rules for calling the `:ask` operator to query the user, and to ensure the proper linkage between the value `true` and the *non-value* `false`, and vice versa.

```
(a-assert "Ask-slots"
          '((:taxonomy (Slots (Ask-slots)))
            (:srules ask-slots
             ((?p ?x true) <- (:ask (?p ?x true)))
             ;; This next rule makes sense ONLY under depth-first search:
             ;; It is used to make sure we always ask the user only positive
             ;; questions -- e.g. "Is it true that (p frame true)?" as opposed
             ;; to "Is it true that (p frame false) ?".
             ((?p ?x false) <- (:unp (:ask (?p ?x true))))
             ;;
             ((?p ?x false) -> (not (?p ?x true)))
             ((?p ?x true) -> (not (?p ?x false)))
             ((not (?p ?x true)) -> (?p ?x false))
             ((not (?p ?x false)) -> (?p ?x true)))))
```

In the following two a-asserts, we declare the first few of a long list of boolean-valued slots, then assert that they belong to the set of *Ask-slots*.

```
(a-assert "Boolean slots, requiring :ask rules"
          '((:slot acceptable-relief      (Patients Booleans)
                   :cardinality 1)
            (:slot acutely-tender-sinuses (Patients Booleans)
                   :cardinality 1)
            (:slot appears-in-great-pain  (Patients Booleans)
                   :cardinality 1)
            (:slot appears-very-sleepy    (Patients Booleans)
                   :cardinality 1)
            . . . ))

(a-assert "The :ask slots"
          '((isa (:slot acceptable-relief)      ask-slots)
```

```
(isa (:slot acutely-tender-sinuses) ask-slots)
(isa (:slot appears-in-great-pain)  ask-slots)
(isa (:slot appears-very-sleepy)    ask-slots)
. . . ))
```

4.2 Belief Networks

Clif Day has used Algernon to implement Judea Pearl's belief networks in the tree-structured case [Pearl, 86]. Pearl's belief networks allow probabilistic data to be combined by local propagation such that each proposition will eventually be assigned a certainty measure consistent with the axioms of Bayesian probability theory. This work differs from the other systems built using Algernon in that it supports probabilistic reasoning (whereas the other systems reason propositionally). Part of the implementation is written in Lisp because Algernon does not currently support arithmetic operations on vectors and matrices.

As the algorithms involved are non-trivial, the interested reader is referred to the descriptions of probability nets given in [Pearl, 86] pp. 241-288 and [Pearl, 88] pp. 143-175. Clif Day's code implementing probability nets is given in the appendix section A.2.

4.3 QPC

QPC is a qualitative model builder based on Qualitative Process Theory [Forbus, 84], but differing from past implementation of Qualitative Process Theory in that it builds QDEs[1] for simulation by QSIM [Kuipers, 1986].

QPC is of interest to the knowledge representation community primarily because it builds a bridge between a knowledge representation system and a qualitative simulator. QPC assembles a QDE model of a physical situation by drawing on a library of *model fragments* (*e.g.*, views and processes); QSIM is then used to predict the behaviors consistent with the model.[2] It would be possible to implement qualitative simulation entirely within a knowledge representation system. However, substantial progress has been made on efficient representations and algorithms for qualitative reasoning [Weld and de Kleer, 1990]. QSIM, in particular, can be viewed as an efficient special purpose theorem prover (with formally established soundness and completeness properties) in the domain of qualitative simulation [Kuipers, 1986].

QPC should enable several types of research. First, it should enable research focusing on the representation and use of knowledge to choose the relevant objects and views of objects for answering specific question about complex situations [Falkenhainer and Forbus, 1988, Iwasaki, 90]. Second, it should enable research on the use of qualitative models to answer questions that go beyond the understanding compiled into the rules of a production system. Finally, it should enable work on building knowledge-based systems which can reason about the applicability of the various qualitative reasoning techniques [Weld and de Kleer, 1990] to complex situations.

[1] Qualitative Differential Equations

[2] This approach was originally proposed by Kuipers in his 1986 AAAI Tutorial on Qualitative Reasoning, and was explored in [Vincent,1988].

```
(objects
  (scenarios)
  (models)
  (cds (views)        ; cds = "conditionalized descriptions" - an old term
                      ; for model fragment.
      (processes))
  (constraints)
  (times ,*t0*)       ; *t0* is a variable bound to the initial QSIM time.
  (variables-magnitudes
     (magnitudes zero inf minf)
     (variables))
  (derivatives inc dec std)
  (physobs)
  (influence-types (direct-influence-types i+ i-)
                   (indirect-influence-types q+ q-))
  (relations (2-relations q= S0+)
             (3-relations add mult))
  (qpt-order-relations q-less q-greater q-equal))
```

Figure 4.1: The QPC taxonomy.

The explanatory text below is taken from a paper written jointly by Adam Farquhar, Ben Kuipers, and myself [Crawford, Farquhar, & Kuipers, 90]. The code for the Algernon portion of QPC is given in the appendix section A.3.[3]

4.3.1 The QPC Knowledge-Base

The QPC knowledge-base has three components. The first consists of a taxonomy of sets, and a collection of slot declarations and rules, which represent the ontology of Qualitative Process Theory. The QPC taxonomy is shown in figure 4.1. Declarations for some of the basic slots are shown in figure 4.2. The second component is a domain library of processes and views. In QPC, both processes and views are represented by forward-chaining rules which create their instances. We refer to both views and processes by the general term *model fragment.* Figures 4.3 and 4.4 shows rules representing the physical view of a physical object and the fluid-flow process, respectively. The third component of the knowledge-base contains instantiated model fragments for specific entities in the world.

4.3.2 The QPC Algorithm

The basic QPC algorithm consists of four steps:

1. Assemble a *view-process structure* from a description of the scenario.

[3] Parts of QPC (specifically the derivation of the QDE from the view-process structure) are currently implemented in Lisp. We expect almost all of QPC to eventually migrate into Algernon.

```
(:slot model (scenarios models)
       :comment "A model of a scenario.")
(:slot entity (models objects)
       :backlink part-of
       :comment "An entity in a model.")
(:slot cd (models cds)
       :backlink part-of
       :comment "CD links a models to its views and processes.")
(:slot variable (cds variables)
       :backlink part-of
       :comment "The variables in the cd.")
(:slot tmag (variables models times magnitudes)
       :comment "TMAG allows us to assert that a variable
                 has a magnitude in a model at a time.")
(:slot tdir (variables models times derivatives)
       :comment "TDIR allows us to assert that a variable
                 has a derivative in a model at a time.")
```

Figure 4.2: Some important QPC slots.

2. Apply the closed world assumption and build the QDE.
3. Form an initial state.
4. Simulate using QSIM.

Two kinds of complexity add iterative paths to this simple sequence (figure 4.5). First, when the initial state is formed, additional variable values are learned which may activate additional views and processes. This may necessitate re-building the QDE. Second, when simulation reaches a boundary of the QDE being simulated, control is returned to QPC so that a new model can be created.

Building the View-Process Structure

Model-building starts with a *scenario* which identifies the entities in the world one is interesting in modeling, and specifies their initial conditions. The entities in the scenario become part of the initial model of the scenario (but may or may not be part of subsequent models of the scenario, as entities can be created or destroyed by region transitions). QPC builds the view-process structure for the initial model by first adding, to the initial model, any entities needed to complete it, and then applying rules to instantiate the relevant views and processes.

We illustrate QPC with the scenario depicted in figure 4.6. It consists of two containers, `A` and `B`, connected by a fluid path. `B` has a *portal* located part way up one side. Initially there is fluid in container `A`.

The scenario is set up in QPC by creating (in the Algernon KB) frames for the containers `A` and `B`, the open fluid path connecting them, and then asserting that there is fluid in `A` and a portal in

```
((physical-view ?x ?view)
 (mass ?x ?mx) (volume ?x ?vx) (pressure ?x ?px)
 ->
   ;; link a view to its variables
   (variable ?view ?mx) (variable ?view ?vx)
   (variable ?view ?px)
   ;; cd-ineqs are inequalities indexed by view
   ;; Mass not less zero
   (not (cd-ineq ?view less ?mx zero))
   ;; Volume not less zero
   (not (cd-ineq ?view less ?vx zero))
   ;; Mass=0 <-> Volume=0
   (correspondence ?view ?mx zero ?vx zero))
```

Figure 4.3: A rule to fill in the physical view of a physical object. This rule is associated with the set of physical objects.

`B`. We also assert that `A` is an *entity* in the initial model of the scenario, but do not explicitly link `B` into the scenario (as QPC will do so automatically). The Algernon assertions to establish the scenario are shown in figure 4.6.. Our current implementation builds the process view structure for this scenario in about 31 seconds (running Austin Kyoto Common Lisp on a Sun 4).

QPC then applies rules to complete the set of entities in the initial model. For example, if a container is part of a model and it is connected, via an open connection, to another container, then the second container should be considered part of the model. Instantiated for fluid-connections this rule reads:

```
((fluid-connection ?obj1 ?path1 ?obj2)
 (open ?path1 true)
 (part-of ?obj1 ?model1)
  ->
  (entity ?model1 ?obj2))
```

where the relation `entity` links a model to its objects.

QPC deduces that `B`, the portal in `B`, and the contents of `A` must be included in the initial model. Instantiation of a fluid flow process from `A` to `B` implies the need for a frame for the contents of `B`, which is created and added to the model (along with a frame for its physical view). The initial model thus consists of the physical views of `A`, `B`, the portal, the contents of `A` and `B`, and the fluid flow process. The influences, relations, correspondences and inequalities of these views and processes are shown in figure 4.7. Notice that, as yet, no process or region transition for portal flow has been added. Neither the process nor the region transitions are set up until the relationship between the fluid level of `B` and the portal height is learned.

Applying the Closed-World Assumption and Building the QDE

At this point, QPC has created a view-process structure comprising a collection of influences, relations, inequalities, and correspondences. The next step is to convert to a QDE which consists

```
((fluid-connection ?can1 ?path ?can2)
 (part-of ?can1 ?model) (isa ?model models)
 (flow-rate ?path ?flow-rate)
 (pressure-difference ?path ?pressure-diff)
 (contents ?can1 ?liquid1)
 (isa ?liquid1 contained-liquids)
 (mass ?liquid1 ?mass1)
 (open ?path true)
 (pressure ?can1 ?pressure-can1)
 (pressure ?can2 ?pressure-can2)
 ->
 ; Find the process OR Create a new one.
 (:forc ?process
      (cd ?model ?process)
      (isa ?process fluid-flow-processes)
      (path ?process ?path))
 (variable ?process ?flow-rate)
 (variable ?process ?pressure-diff)
 (correspondence ?process ?flow-rate zero
                          ?pressure-diff zero)
 ;; pressure-diff = c1.pressure - c2.pressure
 (ADD ?process ?pressure-can2 ?pressure-diff ?pressure-can1)
 (influence ?process Q+ ?pressure-diff ?flow-rate)
 (influence ?process I- ?flow-rate ?mass1)
 (:forc ?liquid2
        (contents ?can2 ?liquid2)
        (same-material ?liquid1 ?liquid2)
        (same-state ?liquid1 ?liquid2))
 (influence ?process I+ ?flow-rate (mass ?liquid2)))
```

Figure 4.4: The rule to instantiate the fluid-flow process. The relation `fluid-connection` links a container, a path, and another container. The relation `cd` links a model to a view or process. This rule is associated with the set of physical objects.

of constraints, quantity spaces, landmarks, and corresponding values.

The key step is to transform a collection of influences into constraints. If X *influences* Y then Y will change as a result of a change in X, *all else being equal.* A *constraint* between X and Y is a universal law, limiting the possible joint behaviors of X and Y, *independent of context.* Thus, in order to transform influences into constraints, we require a Closed World Assumption, asserting that we know *all* the relevant influences.

Intuitively, the indirect influence or "qualitative proportionality" $Q^+(X_1, Y)$, means that an increase in X_1 will tend to increase Y. More formally:

$$Q^+(X_1, Y) \;\equiv\; Y = f(X_1, X_2, \ldots X_n) \text{ and } \frac{\partial f}{\partial X_1} > 0,$$

for some functional relationship f (with an indefinite number of arguments). The direct influence

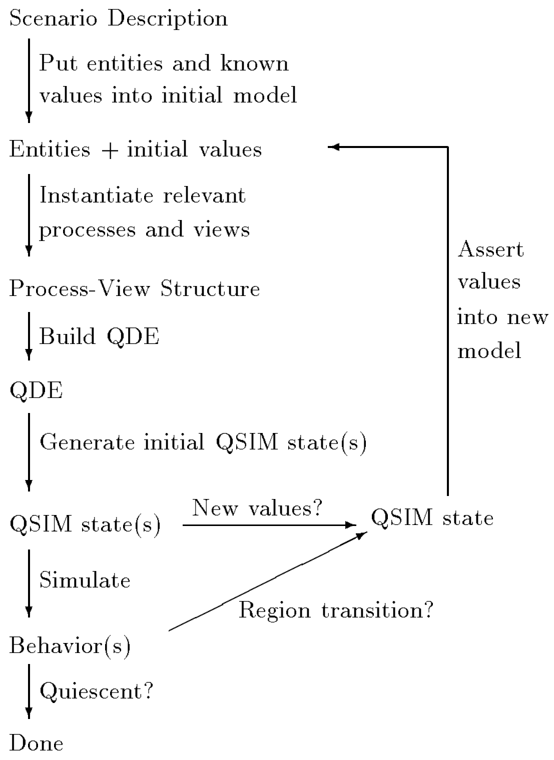


Figure 4.5: Flow of control in QPC

is similar:

$$I^{+}(X_1, Y) \equiv \frac{dY}{dt} = f(X_1, X_2, \ldots X_n) \text{ and } \frac{\partial f}{\partial X_1} > 0.$$

Influence resolution on a variable Y identifies the sets P and N of variables that positively and negatively influence Y. Based on the CWA, this determines the number of arguments to the function f. We make the additional assumption that f can be approximated by a qualitative addition of single-variable functional relationships. This allows us to assert QSIM *constraints* to capture the set of indirect influences on Y:

$$Y = \sum_{X_i \in P} M^{+}(X_i) - \sum_{X_j \in N} M^{+}(X_j).$$

Resolution of direct influences is similar.

Highlights of the initial QDE for the u-tube example are shown in figure 4.8.

Building the Initial State

At this point, we have created a QDE which reflects the current view-process structure, but we do not have initial values for all the variables in the model. We calculate initial values in three steps:

```
;; The U-Tube Scenario
(:create ?utube) (current-scenario global-context ?utube)
;; Create A, B, and the pipe between them.
(:create ?A) (isa ?A containers) (entity ?utube ?A)
(:create ?B) (isa ?B containers)
(:create ?pipe) (open ?pipe true)
(fluid-connection ?A ?pipe ?B)
;; The contents of A has mass greater than zero.
(:create ?A-contents) (contents ?A ?A-contents)
(state ?A-contents liquid-state)
(:create ?A*) (greater ?A* zero)
;; (tmag var model time mag) → var = mag at time in model.
(tmag (level ?A-contents) ?utube (initial-time ?utube) ?A*)
;; B has a portal.
(:create ?port) (isa ?port portals) (portal b ?port) (open ?port
true)
;; ...and forms to assert that the bottoms of A and B
;; are at zero, and the top heights are positive.
```

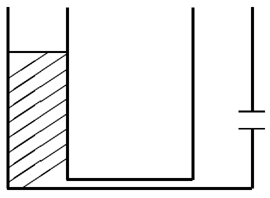


Figure 4.6: Scenario description for the u-tube with portal.

1. Propagate known values through the QDE.
2. Apply default assumptions.
3. Generate possible completions.

Propagation

Frequently, initial values are given for only some of the variables, but other values follow easily from the constraints and relations in the QDE. It would be possible to build rules into the knowledge-base to calculate such values, but this would unnecessarily duplicate the knowledge already in QSIM. Instead, we use QSIM itself as an efficient special purpose reasoning tool to propagate the known values through the QDE. In the u-tube example, propagation concludes, among other values, that the mass of the contents of `A` is greater than zero (but concludes nothing about the mass of the contents of `B`).

Default Assumptions

During automatic model building, it may be impossible to establish values for enough variables to uniquely determine an initial state. Our solution to this problem is to make default assumptions which are appropriate for the model. *E.g.*, in the u-tube no initial value is known for the mass of the contents of `B`, and it is not possible to determine a value through propagation. However, QPC assumes that the mass of any newly created liquid is zero. Such values are explicitly tagged as assumptions so that they can be withdrawn if they lead to a contradiction. In the examples we have looked at, propagating known values before making default assumptions has been sufficient to avoid such contradictions.

A Physical View:
top-height $\geq$ fluid-level $\geq$ bottom-height

A-Contents Physical View:
mass $\geq$ zero; volume $\geq$ zero
volume $\leq$ (volume A)
mass Q^+ volume Q^+ level Q^+ pressure

mass = zero $\leftrightarrow$ volume = zero $\leftrightarrow$
level = zero $\leftrightarrow$ pressure = zero

level = (bottom-height A) $\leftrightarrow$ volume = zero
level = (top-height A) $\leftrightarrow$ volume = (volume A)

Fluid Flow Process:
pressure-diff = (pressure A) - (pressure B)
pressure-diff Q^+ flow-rate
flow-rate I^- (mass (contents A))
flow-rate I^+ (mass (contents B))
flow-rate = zero $\leftrightarrow$ pressure-diff = zero

B-Portal Physical View:
(bottom-height B) < height < (top-height B)
height $\geq$ zero

Figure 4.7: Highlights of the initial views and processes for the u-tube with portal. The physical views of `B` and `B-Contents` are similar to those of `A`.

Generating All Completions

Even after propagation and the default assumptions, there may be variables which do not have known values. At this point we again use QSIM as a special purpose reasoner to construct *all* possible completions of the current state. In simple cases, such as the u-tube, there is only one possible completion. If there are multiple completions, a separate model must be created for each of them.[4]

In either case it is possible for the new values (from propagation, default assumptions or state completion) to require additional region transitions, new views, or new processes. To handle this problem, the resulting completed state information is asserted back to the knowledge-base, causing the appropriate rules to fire. In the u-tube example, the default assumptions lead QPC to assume that the fluid level in `B` is zero. This sets up a region transition that will instantiate the portal flow

[4]This is the first "choice point" in QPC (the second being the case in which a simulation produces several behaviors ending in region transitions). In such cases the possibilities are queued, and we use a simple search strategy to select the one to follow next.

```
(define-qde utube-initial-model
  (quantity-spaces
   (a-contents-level (minf 0 a* a-top inf))
   (b-contents-level (minf 0 b-top inf))
   ...)
  (constraints
   ((m+ a-contents-mass a-contents-volume) (0 0))
   ((m+ a-contents-volume a-contents-level)
        (frame23 a-top) (0 0))
   ((m+ a-contents-level a-contents-pressure)
        (0 0))
   ((= a-contents-level a-fluid-level))
   ((= a-contents-pressure a-pressure))
   ((add b-pressure pipe-ab-pressure-diff a-pressure))
   ((m+ pipe-ab-pressure-diff pipe-ab-flow-rate)
    (0 0))
   ((minus var-1 pipe-ab-flow-rate))
   ((d/dt a-contents-mass var-1))
   ((d/dt b-contents-mass pipe-ab-flow-rate))
   ((constant b-portal-height)) ...)
  ...)
```

Figure 4.8: Highlights from the initial QDE for the example. Constraints on `B` are similar to those on `A`.

process if the fluid level ever reaches the portal height and is increasing.[5]

Simulation and Region Transitions

Once a complete initial state has been created, QSIM is used to simulate the possible behaviors. In the u-tube example, QSIM predicts three behaviors: one in which equilibrium is reached below the portal-height, one in which equilibrium is reached exactly at the portal height, and a third in which the fluid level in tank `B` reaches the portal and continues to increase. The first two behaviors can be simulated using only the initial model. The third behavior, however, triggers a region transition and the building of a new model.

When a behavior ends in a region transition, QPC attempts to construct a new set of models. This is done by creating an empty model and asserting the quantity spaces and variable values of the final state of the behavior into it. The new model is then linked to its predecessor. QPC checks the previously active process and view instances to determine which remain active in the new model. QPC then determines whether new entities need to be included, and whether new views or processes need to be activated. Finally, the QDE and initial state(s) are built as before.

[5]In complex models, the additional views and processes activated at this point may invalidate the closed world assumption; new views and processes may add influences on some variable v previously assumed to be constant. In such cases, we must return to the original view-process structure and assert the new influence on v, rebuild the QDE, and recalculate the initial values.

Figure 4.9: A QSIM behavior for the u-tube example which spans two models. The time step begins again at zero for the second model. `B-portal-flow-rate` and `var-2` are not present in the initial model, but are defined by the portal-flow process. `Var-2` is the net flow into `B`, *i.e.*, `pipe-ab-flow-rate` - `B-portal-flow-rate`.

In the u-tube example, after the region transition, QPC is able to retain pointers to the old model fragments for `A`, `B`, `B-portal`, and `Fluid-flow-AB`. A new portal flow process is created, since its precondition, that the fluid level in `B` is greater than or equal to the portal height, is now satisfied. This results in an additional influence on `B-contents-mass`.

The expected behavior is, of course, that the level of liquid in `B` will increase until the flow in from `A` and the flow out of the portal equalize, and then the liquid will drain out of the portal until a final equilibrium is reached, with the level of `B` even with the portal. This type of behavior poses a problem for qualitative simulators because `B-net-flow` is the difference between two positive and decreasing values. Simple qualitative subtraction is ambiguous, and the result can become negative, zero, or positive any number of times. This behavior is known as *chatter*, and results in an infinite number of qualitatively distinct behaviors.

Fortunately, there is a solution. QSIM automatically derives constraints based on the second derivatives of the variables [Kuipers and Chiu, 1987]. It is this sort of advance in qualitative mathematics which we were hoping to take advantage of! Instead of producing an infinite tree of behaviors, QSIM produces a small number: `B-contents-level` reaches a maximum somewhere above the portal, or it reaches a maximum at the top-height, or B overflows, triggering a region transition. The first two behaviors drop down to our expected equilibrium state; the third causes a new model to be constructed. Figure 4.9 shows a QSIM plot for a behavior spanning two models and ending in the final equilibrium state in which the level in `B` is at the height of the portal.

4.3.3 The QPC Knowledge-Base (again)

Since QPC builds models incrementally, we must deal with a version of the frame problem: what must change and what remains the same after a region transition? Rather than build the new model from scratch, we have structured the representation so that chunks of the old model may be incorporated in the new model. The representation is structured in layers, as shown below, so that each layer changes more slowly than the one below it.

Individuals and their Relationships
Views and Processes
Models (QDEs)
Variable Values

- At the lowest level are the values of variables in the model. These values generally change at every step of the simulation.
- One step up are the models (QDEs) built by QPC. Models are likely to remain valid for several simulation steps, but still change whenever a region transition occurs.
- Changing more slowly, are the views and processes. When region transitions occur, they generally cause one or more view or process instances to become invalid and one or more new ones to be activated. In general, however, most of the views and processes from the previous model are still valid. For example, in the u-tube example, when the portal flow begins, a new "portal flow" process is created, but the views of the containers, and the old fluid flow process, are unchanged.
- Finally, the set of individuals and their relationships change the most slowly. For example, initiation of a boiling process would create a new individual to represent the steam produced. Our framework handles the creation or deletion of individuals naturally.

Part II

Formalizing Access Limited Logic

Chapter 5

Formal Development of ALL

This chapter presents the full formal development of ALL (without negation). It is organized into the following sections:

1. Basic Notation.
2. Formal Syntax of ALL: First the alphabet of an ALL is defined and then terms, propositions, access paths, rules, knowledge-bases and finally formulas are defined.
3. Knowledge Theory: We present the knowledge theory of ALL which defines the values of formulas (*i.e.*, the results of queries and assertions into knowledge-bases). This subsection also includes a series of basic lemmas and a theorem showing that formulas in ALL are well-defined.
4. Soundness: The soundness of ALL is shown in terms of a mapping to Predicate Calculus.
5. Completeness: We review some notation and results from logic programming and use them to prove the Socratic and Partitional Completeness of ALL.

We have placed the proofs of some of the more technical lemmas in the appendix.

5.1 Basic Notation

We use the following general conventions for variable names (the definitions of the terms used here are given in below in sections 5.2 and 5.3). Capital English letters stand for components of knowledge-bases. Greek letters from the beginning of the alphabet denote lists of propositions. Greek letters near the middle of the alphabet (*e.g.*, θ, η, and ζ) denote substitutions. ρ always stand for a rule. Calligraphic letters (*e.g.*, $\mathcal{O}$) denote operations on knowledge-bases. Small English letters denote everything else (*e.g.*, constants are $c_1, \ldots$, partitions $p_1, \ldots$ and so on).

We write quantified expressions in the form:

$$(\langle Quantifier\rangle\langle Variable\rangle : \langle Range\rangle : \langle Expression\rangle).$$

Thus, for example:

$$(\forall x : pred_1(x) : pred_2(x))$$

is read "For all x such that $pred_1(x)$, $pred_2(x)$". Similarly:

$$(\bigcup x : pred(x) : foo(x))$$

(where foo is a set valued function) denotes the union, over all x such that $pred(x)$, of $foo(x)$.

We delineate lists with the usual () and denote the empty list by nil. If α is a list then:

- $head(\alpha)$ is the first element in α.
- $rest(\alpha)$ is all but the first element in α.

We define $append(\alpha_1, \alpha_2)$ to be the result of appending the list α_1 to the beginning of the list α_2.

5.2 Formal Syntax of ALL

The syntax of ALL is quite similar to the syntax of logic programming. Accordingly we develop the syntax of ALL generally following the notation in [Apt, 88].

5.2.1 Alphabets, Terms, and Propositions

The language of access-limited logic consists of an alphabet and the formulas defined over it. The *alphabet* of an ALL consists of the following:

1. A countably infinite set of *variables* denoted $v_1, v_2, \ldots$
2. A countably infinite set of *constants* denoted $c_1, c_2, \ldots$
3. For every arity n a countably infinite set of *relations* denoted $r_1^n, r_2^n, \ldots$[1]
4. The connectives $\rightarrow$ and $\leftarrow$.
5. The operators *query* and *assert*.

A *term* is a constant or a variable. If S is a set of terms then:

- $vars(S)$ is the set of variables appearing in S.
- $constants(S)$ is the set of constants appearing in S.

$r_i^n(t_1, \ldots, t_n)$ is a *proposition* iff r_i^n is a relation and for all i $(1 \leq i \leq n)$, t_i is a term.[2] If α is a term, proposition, or list of propositions then:

- $vars(\alpha)$ is the set of variables appearing in α.
- $relations(\alpha)$ is the set of relations appearing in α.
- $constants(\alpha)$ is the set of constants appearing in α.

If $vars(\alpha) = \emptyset$ then α is *ground*. A ground propositions is referred to as a *fact*).

[1] We generally omit the superscripts from relations as they are clear from context.

[2] n place relations cause no problems in ALL. Intuitively, $r(t_1, \ldots, t_n)$ corresponds to putting the value $(t_2, \ldots, t_n)$ in the r slot of t_1.

5.2.2 Access Paths

An *access path* (or simply a *path*) is a pair $\langle V, \alpha \rangle$ where α is a list of propositions and V is a set of variables. In general, variables appearing as the first argument to propositions in α must have occurred previously in α. The variables in V are exceptions to this rule and may occur as the first argument to propositions without having occurred previously in α. The need for such exceptions will become apparent when rules are defined below.

Formally, if V is a set of variables and α is a list of propositions, then the pair $\langle V, \alpha \rangle$ is an *access path* (or a *path* for short) iff $\alpha = nil$ or both of the following hold:

1. $head(\alpha) = r_i^n(t_1, \ldots, t_n)$ where either t_1 is a constant or $t_1 \in V$.
2. $\langle V \cup vars(head(\alpha)), rest(\alpha) \rangle$ is an access path.

If V is empty we usually omit it and simply say α is an access path. A path of length one is a *primitive path*.

5.2.3 Rules

Assume $r(t_1, \ldots, t_n)$ is a proposition and α is a non-empty list of propositions. $r(t_1, \ldots, t_n) \leftarrow \alpha$ is an *if-needed rule* iff both of the following hold:

1. Either t_1 is a constant and α is a path, or t_1 is a variable and $\langle \{t_1\}, \alpha \rangle$ is a path.
2. $vars(\{t_1, \ldots, t_n\}) \subset vars(\alpha)$.

Intuitively, the first restriction ensures that once the consequent of the rule has been unified against a primitive path, the antecedent is an access path. This will be make precise after substitutions are defined in section 5.3.2. The second restriction ensures that any substitution which grounds the antecedent of a rule also grounds its consequent.

Now assume $\rho = r(t_1, \ldots, t_n) \leftarrow \alpha$ is an if-needed rule. We use the following accessor functions:

- $Key(\rho) = r(t_1, \ldots, t_n)$.
- $Conseq(\rho) = r(t_1, \ldots, t_n)$.
- $Ant(\rho) = \alpha$.

Intuitively, the Key of a rule is the proposition the rule is indexed under in the knowledge-base.

Assume $r(t_1, \ldots, t_n)$ is a proposition and α is a non-empty list of propositions. $\alpha \rightarrow r(t_1, \ldots, t_n)$ is an *if-added rule* iff both of the following hold:

1. $\langle vars(head(\alpha)), \alpha \rangle$ is a path.
2. $vars(\{t_1, \ldots, t_n\}) \subset vars(\alpha)$.

The first restriction ensures that once the head of the antecedent has been unified against a fact, the antecedent is a path. The second restriction again ensures that any substitution which grounds the antecedent of a rule also grounds its consequent.

Finally, assume $\rho = \alpha \rightarrow r(t_1, \ldots, t_n)$ is an if-added rule. We again use the accessor functions:

- $Key(\rho) = head(\alpha)$.
- $Conseq(\rho) = r(t_1, \ldots, t_n)$.
- $Ant(\rho) = \alpha$.

5.2.4 Knowledge-Bases

If S is a set then $s_1, \ldots, s_n$ is a *partitioning* of S iff:

- $(\forall i : 0 \leq i \leq n : s_i \subset S)$, and
- $(\bigcup i : 1 \leq i \leq n : s_i) = S$

A *Knowledge-Base*, K, is a six-tuple $\langle C, R, Nr, Ar, F, P\rangle$ where:

C	$=$	A set of constants.
R	$=$	A set of relations.
Nr	$=$	A set of if-needed rules such that $(\forall \rho : \rho \in Nr$: $constants(\rho) \subset C \wedge relations(\rho) \subset R)$.
Ar	$=$	A set of if-added rules such that $(\forall \rho : \rho \in Ar$: $constants(\rho) \subset C \wedge relations(\rho) \subset R)$.
F	$=$	A set of facts such that $(\forall f : f \in F$: $constants(f) \subset C \wedge relations(f) \subset R)$.
P	$=$	A partitioning of $C \times R$.

If $K = \langle C, R, Nr, Ar, F, P\rangle$ is a knowledge-base and α is a proposition or list of propositions, then α is *allowed* in K iff $constants(\alpha) \subset C \wedge relations(\alpha) \subset R$. Finally, the members of P are referred to as the *partitions* of K.

Unless otherwise specified, a knowledge-base K should be understood to have components $\langle C, R, Nr, Ar, F, P\rangle$. Similarly, for an integer $i > 0$, a knowledge-base K_i is understood to have components $\langle C, R, Nr, Ar_i, F_i, P\rangle$ (we subscript Ar and F because, as will be seen, they are the two components which change when operations are performed).

5.2.5 Operations and Formulas

If α is a non-empty path then $query(\alpha)$ is a *query*. If α is a primitive path then $query(\alpha)$ is a *primitive* query. If f is a fact then $assert(f)$ is an *assertion* (assertions of paths are not currently allowed). Any assertion or query is an *operation*, and any assertion or primitive query is a *primitive* operation. If $\mathcal{O} = query(\alpha)$ or $\mathcal{O} = assert(f)$ then $\mathcal{O}$ is *allowed* in a knowledge-base K iff α is allowed in K or f is allowed in K, respectively. If an operation $\mathcal{O}$ is allowed in a knowledge-base K then $\mathcal{O}(K)$ is an ALL *formula*.

5.3 Knowledge Theory

The knowledge theory of ALL defines the values of ALL formulas by defining the action of ALL operations (*i.e.*, queries and assertions). Intuitively, the assertion of a fact f, adds f to a knowledge-base and returns the resultant knowledge-base (*i.e.*, the knowledge-base after f is added, all applicable if-added rules are applied, and all if-added rules are closed (see section 5.3.6)). A query of q returns the substitutions needed to make q true in the knowledge-base. It also returns a new knowledge-base (since processing the query may change the knowledge-base by invoking rules). Before formally defining the knowledge theory of ALL we first define some additional notation.

5.3.1 Partitions

Intuitively, a partition of K corresponds to a part of the knowledge-base which is somehow semantically cohesive, and distinct from the rest of the knowledge-base. Facts and rules are often thought of as being 'in' partitions and operations are thought of as 'taking place' in subsets of $C \times R$ (unions of partitions). The intuition behind this comes from the frame view of ALL knowledge-bases. Recall that ALL constants can be thought of as frames, and relations as slots in these frames (*e.g.*, the fact $r(c_1, c_2)$ is equivalent to having the value c_2 in the r slot of the frame c_1). Thus a pair $\langle r, c\rangle$ can be thought of as a particular slot in a particular frame in the knowledge-base. Recall that we refer to such a pair as a *frame-slot*. Partitions are thus sets of frame-slots. Further, note that any primitive path α (by the definition of a path) must reference exactly one frame-slot and thus can be said to be 'in' a partition. In fact, since partitions can overlap, it can be in several partitions and any operation on α is performed 'in' the subset of $C \times R$ formed by taking the union of these partitions. Intuitively, this union defines the rules which are available to the operation.

More formally, if K is a knowledge-base, $\alpha = (r(c, t_1, \ldots, t_n))$ a primitive path (*i.e.*, c a constant and all t_i, $1 \leq i \leq n$, are terms), and p a subset of $C \times R$ (*e.g.*, a partition or the union of several partitions — we are often be a bit loose with variable naming and use p_i to range over both partitions and general subsets of $C \times R$) then $\alpha \in p$ iff $\langle c, r\rangle \in p$. If $P = \{p_1, \ldots, p_n\}$.[3] then:

$$par_K(\alpha) = (\bigcup i : (1 \leq i \leq n) \wedge \alpha \in p_i : p_i)$$

Intuitively, $par_k(\alpha)$ is the union of all the p_i of which α is a member.

Lemma 1 *If K is a knowledge-base and α is a primitive path allowed in K then:*

1. $par_K(\alpha) \subset C \times R$.
2. $\alpha \in par_K(\alpha)$

Proof: Both parts follow from the definitions. ▮

If K is a knowledge-base, $p \subset C \times R$, and $\mathcal{O} = query(q)$ is a primitive operation then $\mathcal{O} \in p$ iff $q \in p$. Further, $par_K(\mathcal{O}) = par_K(q)$. Similarly, if $\mathcal{O} = assert(f)$ is an operation then $\mathcal{O} \in p$ iff $f \in p$, and $par_K(\mathcal{O}) = par_K(f)$.[4]

[3] Recall that a knowledge-base K is understood to have components $\langle C, R, Nr, Ar, F, P\rangle$. Thus P is the set of partitions of K.

[4] Technical note: We often fail to distinguish between f and (f) (that is, between the fact f and the primitive path (f)), or between q and (q).

5.3.2 Substitutions

We lift our notation for substitutions almost entirely from [Apt, 88] (with some modifications due to [Palamidessi, 89]). A *substitution* is a finite mapping from variables to terms:

$$\theta = \{v_1/t_1, \ldots, v_n/t_n\}$$

where v_i $(1 \leq i \leq n)$ are distinct variables. For such a substitution θ, let $dom(\theta) = \{v_1, \ldots, v_n\}$, and $range(\theta) = \{t_1, \ldots, t_n\}$. If all t_i $(1 \leq i \leq n)$, are constants then θ is said to be *ground*. Let the variables in the alphabet be V (see section 5.2.1). A substitution θ is a *renaming* iff it is a bijection (*i.e.*, a $1:1$ onto mapping) from V to V.

If e is an expression and θ is a substitution then $e\theta$ is the result of applying θ to e (simultaneously replacing each occurrence in e of the variables in the domain of θ with the corresponding term). If θ is a renaming then $e\theta$ is a *variant* of e. For a proposition q let:

$$ground(q) = \{f \mid f \text{ is a fact } \wedge (\exists\theta : \theta \text{ is a ground substitution: } f = q\theta)\}.$$

If θ and η are substitutions then $\theta\eta$ denotes θ followed by η (formally, if $\theta = \{x_1/s_1, \ldots, x_n/s_n\}$ and $\eta = \{y_1/t_1, \ldots, y_n/t_n\}$ then $\theta\eta$ is:

$$\{x_1/s_1\eta, \ldots, x_n/s_n\eta, y_1/t_1, \ldots, y_n/t_n\}$$

minus the pairs $x_i/s_i\eta$ for which $x_i = s_i\eta$ and the pairs y_i/t_i for which $y_i \in \{x_1, \ldots, x_n\}$).

If there exists a substitution ζ such that $\eta = \theta \circ \zeta$ then θ is *more general* than η. Intuitively, if θ is more general than η then θ does 'less work' than η. A *unifier* of two primitive propositions q_1 and q_2, is a substitution θ such that $q_1\theta = q_2\theta$. The *most general unifier* of two primitive propositions q_1 and q_2 is a unifier θ of q_1 and q_2 such that for any other unifier η of q_1 and q_2, θ is more general than η. A unifier θ of q_1 and q_2 is *relevant* iff $dom(\theta) \subset vars(q_1) \cup vars(q_2)$ and $vars(range(\theta)) \subset vars(q_1) \cup vars(q_2)$ (*i.e.*, there are no irrelevant variables in θ). We denote the most general relevant unifier of q_1 and q_2 as $mgru(q_1, q_2)$.[5] It has been shown in [Apt, 88] that any propositions which are unifiable, have a most general relevant unifier.

We also use some substitution notation specialized for the formalization of ALL. For a path α let:

$$\Theta_\alpha = \{\theta \mid \theta \text{ is a ground substitution } \wedge dom(\theta) = vars(\alpha)\}.$$

Thus Θ_α is the set of ground substitutions binding all and only the variables in α. For V a set of variables and θ a substitution, let $\theta\backslash_V$ be the substitution θ with its domain restricted to only the variables in V.

The following lemmas show some basic properties of substitutions and access paths.

Lemma 2 *If θ and η are substitutions and α is a list of propositions then:*

1. *If θ and η are ground then $\theta\eta$ is ground.*

[5] The use of most general relevant unifiers (instead of just most general unifiers) is necessary for technical reasons. The basic problem with most general unifiers is that one can compose a most general unifier with an arbitrary renaming and the result is still a most general unifier.

2. *If θ ground then $vars(\alpha\theta) = vars(\alpha) - dom(\theta)$.*

Proof: From definitions. ▮

Lemma 3 *If $\alpha \neq nil$ is a path, V a set of variables and θ a substitution then:*

1. *If θ and $head(\alpha)\theta$ are ground then $rest(\alpha)\theta$ is a path.*
2. *If θ, $head(\alpha)\theta$, and $rest(\alpha)\eta$ are ground then $\alpha\theta\eta$ is ground.*
3. *If β_1 and β_2 are lists of propositions such that $vars(\beta_1) \subset vars(\beta_2)$ and $\beta_2\theta$ is ground then $\beta_1\theta$ is ground.*

Proof: The only hard part is part 1 (2 and 3 follow from the definitions).

First two technical claims. For any list of propositions β, any set of variables W, and W', and any substitution ζ:

Claim 1: If $W' \cap vars(\beta) = \emptyset$ then $\langle W, \beta \rangle$ is a path iff $\langle W - W', \beta \rangle$ is a path.

Claim 2: If ζ is ground and $\langle W, \beta \rangle$ is a path then $\langle W, \beta\zeta \rangle$ is a path.

Claim 1 says that variables not in α are 'irrelevant'. It follows by induction on the length of β. Claim 2 says that if one binds some of the variables in β to constants then β remains a path. Claim 2 also follows by induction on the length of β (the only trick is to use Claim 1 in the induction step).

Now we can prove 1. Let $q = head(\alpha)$ and $\alpha' = rest(\alpha)$. Note initially that $q\theta$ is ground implies (by lemma 2 part 2) $vars(q) \cap vars(\alpha'\theta) = \emptyset$.

α a path $\Longrightarrow$
{by definition of paths}
$\langle vars(q), \alpha' \rangle$ a path $\Longrightarrow$
{by Claim 2 above}
$\langle vars(q), \alpha'\theta \rangle$ a path $\Longrightarrow$
{ by note above and Claim 1 with $W' = W = vars(q)$ and $\beta = \alpha'\theta$. }
$\langle \emptyset, \alpha'\theta \rangle$ is a path

5.3.3 Manipulating Knowledge-bases

If K_1 and K_2 are knowledge-bases (differing only in their facts and if-added rules) then

$$K_1 \cup K_2 = \langle C, R, Nr, Ar_1 \cup Ar_2, F_1 \cup F_2, P \rangle.$$

Further, $K_1 \subset K_2$ iff $F_1 \subset F_2$ and $Ar_1 \subset Ar_2$. If K is a knowledge-base and f is a fact allowed in K then:

$$K + f = \langle C, R, Nr, Ar, F \cup \{f\}, P \rangle,$$

and $f \in K$ iff $f \in F$. Similarly, if ρ is an if-added rule then:

$$K + \rho = \langle C, R, Nr, Ar \cup \{\rho\}, F, P\rangle.$$

Finally, if $S = \{s_1, \ldots, s_n\}$ then $K + S = K + s_1 + \ldots + s_n$. Note that if K_1 and K_2 are knowledge-bases and f is a fact allowed in K_1, then $K_1 \cup K_2$ and $K_1 + f$ are knowledge-bases. Similarly, if ρ is an if-added rule allowed in a knowledge-base K, then $K + \rho$ is a knowledge-base.

In the sections which follow we often work with an ordered pair consisting of a set of substitutions and a knowledge-base. If K_1 and K_2 are knowledge-bases and Θ_1 and Θ_2 are sets of substitutions then let:

$$\langle\Theta_1, K_1\rangle \cup \langle\Theta_2, K_2\rangle = \langle\Theta_1 \cup \Theta_2, K_1 \cup K_2\rangle.$$

Further, $\langle\Theta_1, K_1\rangle \subset \langle\Theta_2, K_2\rangle$ iff $\Theta_1 \subset \Theta_2$ and $K_1 \subset K_2$. Finally, if η is a substitution and Θ a set of substitutions then,

$$\eta \circ \Theta = \{\eta\theta \mid \theta \in \Theta\}.$$

5.3.4 The Domain and Range of All Operations

Any given sets C, R, Nr, P define a finite set of possible knowledge-bases KB (differing only in facts and if-added rules), and an infinite set of ground substitutions Θ (binding variables in the alphabet to constants in C).[6] For any operation $\mathcal{O}$ allowed in the knowledge-bases in KB (note that an operation allowed in any knowledge-base in KB is allowed in all knowledge-bases in KB):

$$\mathcal{O} : KB \longrightarrow 2^{\Theta} \times KB. \tag{5.1}$$

We denote these returned values with pairs: ⟨ 'set of substitutions', 'knowledge-base' ⟩. and use *sub* and *kb* as accessors on the first and second components respectively.

5.3.5 The Values of ALL Operations — Motivation

In this subsection we motivate the definitions of ALL operations given on page 116.

Defining the values of ALL operations requires that we formalize the familiar behavior of forward and backward chaining rules in a knowledge-base. This is a non-trivial exercise, but it is necessary in order to carefully prove the theorems which follow. Further, some interesting (and not entirely expected) problems arise in formalizing our intuitive notion of rule chaining.

Consider a primitive query: $query(q)$. Intuitively, we have only to apply the if-needed rules for q, and return the new knowledge-base, and the set of substitutions which unify q with facts in the new knowledge-base. Thus, we can begin (a first try at) a formal definition of primitive queries by setting R equal to the set of applicable if-needed rules:

$$R = \{\rho\theta \mid \rho \in Nr \wedge \theta = mgru(Key(\rho), q)\}$$

But this is not quite right because we have to first rename the variables in the rule (to avoid accidental collisions between variables in the rule and variables in q). Thus, let η be a renaming

[6] Technically, we should write $KB_{C,R,Nr,P}$ and $\Theta_{C,R,Nr,P}$, but we generally omit the subscripts since they are clear from context.

which maps variables in rules in Nr to variables not used in q or K (this must be possible since the alphabet contains a countably infinite number of variables). Then:

$$R = \{\rho\eta\theta \mid \rho \in Nr \wedge \theta = mgru(Key(\rho)\eta, q)\}$$

We can then find K', the knowledge-base resulting from the application of the rules in R. If $R = \emptyset$ then:

$$K' = K$$

Otherwise, we apply the rules and union the results. Applying a rule consists of querying its antecedent and then asserting its consequent. The consequent is asserted with all substitutions under which the antecedent succeeds:

$$K' = (\bigcup \rho \in R :: (\bigcup \theta \in sub(query(Ant(\rho))(K)) :: kb(assert(Conseq(\rho\theta))(K))))$$

But there is at least one problem with this formula: querying the antecedent of the rules may add facts to the knowledge-base and we would like these facts to be in K'. So we can let:

$$\begin{aligned} K' = (\bigcup \rho \in R :: &kb(query(Ant(\rho))(K)) \cup \\ &(\bigcup \theta \in sub(query(Ant(\rho))(K)) :: kb(assert(Conseq(\rho\theta))(K)))) \end{aligned}$$

Then we can calculate the value of $query(q)$:

$$query(q) = \langle\{\theta | (\exists f \in K' :: \theta = mgru(q, f))\}, K'\rangle$$

Assertions can be defined similarly. However, there are still three problems with these definitions:

1. If-added incompleteness.
2. Recursive rules.
3. Partitions.

These problems are discussed in the next three subsections and the final definitions are given on page 116.

5.3.6 The Problem of If-Added Incompleteness

The application of an if-added rule is triggered by the assertion of a fact which matches the *key* of the rule. One problem with this approach is that, if one is not careful, it can be the case that forward chaining rules, whose antecedents are entailed by the knowledge-base, may never fire. Such a case is shown in the figure 5.1.

Our solution to this problem is to 'close' the if-added rules in the knowledge-base. Intuitively this means that for any fact f and any if-added rule $\alpha \rightarrow q$, if $\theta = mgru(head(\alpha), f)$ then we add to the knowledge-base the rule $rest(\alpha)\theta \rightarrow q\theta$. Consider the example in figure 5.1. Note that the initial knowledge-base is not closed, since it includes the rule $r_1(x, x), r_2(x, x) \rightarrow r_3(x, x)$ and the fact $r_1(c, c)$, but not the rule $r_2(c, c) \rightarrow r_3(c, c)$. Further, if we close K and then assert $r_2(c, c)$, the

Assume K is a knowledge-base such that:

$$
\begin{aligned}
C &= \{c\} \\
R &= \{r_1, r_2, r_3\} \\
Nr &= \{\} \\
Ar &= \{r_1(x,x), r_2(x,x) \rightarrow r_3(x,x)\} \\
F &= \{r_1(c,c)\} \\
P &= \{\{\langle c, r_1\rangle, \langle c, r_2\rangle, \langle c, r_3\rangle\}\}
\end{aligned}
$$

Consider $assert(r_2(c,c))(K)$. After this operation both $r_1(c,c)$ and $r_2(c,c)$ will be facts in the knowledge-base. However, $query(r_3(c,c))(K)$ will fail. The rule $r_1(x,x), r_2(x,x) \rightarrow r_3(x,x)$ never applies because $r_1(c,c)$ was added before $r_2(c,c)$.

Figure 5.1: An example of if-added incompleteness.

resultant knowledge-base does include $r_3(c,c)$. In general we will work with closed knowledge-bases and show that closure in maintained by ALL operations.

We first define formally what it means for a knowledge-base to be closed, and then define an operation which computes the closure of a knowledge-base. A knowledge-base K is *if-added closed* (or just *closed*) iff for any fact $f \in F$, and if-added rule $(\alpha \rightarrow q) \in Ar$, for all θ such that $\theta = mgru(head(\alpha), f)$:

$$
\begin{aligned}
&(length(\alpha) = 1 \quad \wedge \quad q\theta \in F) \vee \\
&(length(\alpha) > 1 \quad \wedge \quad (rest(\alpha)\theta \rightarrow q\theta) \in Ar).
\end{aligned} \tag{5.2}
$$

The key property of closed knowledge-bases is that any fact entailed by an if-added rule is present in the knowledge-base (this will be important in the completeness proofs):

Lemma 4 *For any closed knowledge-base K, and any if-added rule $((q_1, \ldots, q_n) \rightarrow r) \in Ar$:*

$$(\forall \theta : (\forall i : 1 \leq i \leq n : q_i\theta \in F) : r\theta \in F).$$

Proof: By induction on n. ▮

In order to define the closure of a knowledge-base, we first define the closure of a knowledge-base with respect to an if-added rule. If K is a knowledge-base and $\alpha \rightarrow q$ an if-added rule allowed in K then:

If $length(\alpha) = 1$ then:

$$
\begin{aligned}
closure(K, \alpha \rightarrow q) \quad = \quad & K + \alpha \rightarrow q + \\
& \{q\theta \mid (\exists f \in K :: \theta = mgru(head(\alpha), f))\}
\end{aligned} \tag{5.3}
$$

else:

$$closure(K, \alpha \to q) = K + \alpha \to q + \{rest(\alpha)\theta \to q\theta \mid (\exists f \in K :: \theta = mgru(head(\alpha), f))\} \tag{5.4}$$

Intuitively, we can now define the closure of a knowledge-base as the union over all if-added rules, $\rho \in Ar$, of $closure(K, \rho)$. There is, however, one catch: $closure(K, \rho)$ may add facts and if-added rules which we must then close over. To solve this problem we iterate:

As a shorthand, let $Ar^n_{closure}$ = the if added rules in $closure_n(K)$. Then:

$$closure_0(K) = K \tag{5.5}$$

$$closure_n(K) = (\bigcup \rho \in Ar^{n-1}_{closure} :: closure(closure_{n-1}(K), \rho)) \cup closure_{n-1}(K) \tag{5.6}$$

$$closure(K) = (\bigcup n : 0 \leq n : closure_n(K)) \tag{5.7}$$

Finally, for a set of substitutions Γ, and a knowledge-base K:

$$closure(\langle \Gamma, K \rangle) = \langle \Gamma, closure(K) \rangle.$$

The first two lemmas about closure will be needed for the proofs of the analogous properties of ALL operations.

Lemma 5 *For any knowledge-base K:*

$$K \subset closure(K).$$

Proof: From the definitions, one can show that for any rule ρ allowed in K:

$$K + \rho \subset closure(K, \rho).$$

By induction on n, one can then show that for any $n > 0$, $K \subset closure_n(K)$. The result then follows. ∎

Next we show that *closure* is monotonic.

Lemma 6 *For any knowledge-bases K_1 and K_2 such that $K_1 \subset K_2$:*

$$closure(K_1) \subset closure(K_2).$$

Assume K is a knowledge-base such that:

$$\begin{aligned} C &= \{c\} \\ R &= \{r\} \\ Nr &= \{r(x,y) \leftarrow r(x,y)\} \\ Ar &= \{\} \\ F &= \{\} \\ P &= \{\{\langle c, r\rangle\}\} \end{aligned}$$

Consider $query(r(c,c))$. Using the definitions in section 5.3.5:

$$R = \{r(c,c) \leftarrow r(c,c)\}$$

$$K' = kb(query(r(c,c))(K)) \cup (\bigcup \theta \in sub(query(r(c,c))(K)) :: kb(assert(r(c,c))(K)))$$

Thus the 'simple' definition of $query(r(c,c))$ on page 112 is infinitely recursive.

Figure 5.2: An example of a knowledge-base with a recursive rule.

Proof: As in lemma 5, one can show that for any if-added rule ρ allowed in K:

$$closure(K_1, \rho) \subset closure(K_2, \rho).$$

One can then show by induction that for any $n > 0$:

$$closure_n(K_1) \subset closure_n(K_2).$$

The result then follows. ∎

Finally, we have the key property of *closure*:

Lemma 7 *For any knowledge-base K, $closure(K)$ is closed.*

Proof: Straightforward from the definitions. ∎

5.3.7 The Problem of Recursive Rules

A second problem with the definitions given above (page 111) is that it is not well founded; if the knowledge-base contains recursive rules then the definition is infinitely recursive. An example of a knowledge-base with a recursive rule is shown in figure 5.2.

Intuitively, our solution to this problem is to take the fixed point of the recursion; instead of backchaining (or forward chaining) infinitely, we chain just far enough so that chaining any further would have no affect on the resultant knowledge-base. In order to define (the operation) $\mathcal{O}$ as a fixed point we introduce $\mathcal{O}_n$. $\mathcal{O}_n$ will be the result of the operation $\mathcal{O}$ with rule chaining cut off at depth n. We then define $\mathcal{O}$ as the infinite union over all n of $\mathcal{O}_n$. We show (in section 7) that there is always an n after which increasing the depth of rule chaining does not affect the result.

5.3.8 Partitions (again)

The astute reader may have noticed that the definitions given in section 5.3.5 do not force operations to take place in partitions — rule backchaining across the entire knowledge-base is allowed. We enforce the partitioning by allowing $\mathcal{O}_n$ to take two arguments: a knowledge-base and a partition. Within the definition of $\mathcal{O}_n$ we then disallow backchaining out of the partition.

5.3.9 The Values of ALL Operations — Definitions

Consider any operation $\mathcal{O}$ allowed in a knowledge-base K.

Base Case: If $\mathcal{O}$ is a primitive operation then we take the union over all n of $\mathcal{O}_n$ within the partition of $\mathcal{O}$:

$$\mathcal{O}(K) = closure((\bigcup n : n > 0 : \mathcal{O}_n(K, par_K(\mathcal{O})))) \tag{5.8}$$

Recursion: If $\mathcal{O}$ is non-primitive then it must be a query (see section 5.2.5). Thus $\mathcal{O} = query(\alpha)$, where α a path of length greater than 1. Let $head(\alpha) = q$, and $rest(\alpha) = \alpha'$. We query q and then, branching on all resultant substitutions, query α':

If $sub(query(q)(K)) = \{\}$ (*i.e.*, $query(q)$ 'failed'),

$$\mathcal{O}(K) = \langle \{\}, K \rangle \tag{5.9}$$

else ($query(q)$ succeeded so we branch on all resultant substitutions and union the results):

$$\begin{aligned}\mathcal{O}(K) = closure((\bigcup \theta \in sub(query(q)(K)) :: \langle\, \theta \circ sub(query(\alpha'\theta)(K)), \\ kb(query(q)(K)) \cup kb(query(\alpha'\theta)(K))\rangle\,))\end{aligned} \tag{5.10}$$

Figure 5.3 shows a query on a simple knowledge-base.

It remains only to formally define $\mathcal{O}_n$. For any n, and any operation $\mathcal{O}$ allowed in the knowledge-bases of KB:

$$\mathcal{O}_n : KB \times 2^{C \times R} \Longrightarrow 2^{\Theta} \times KB. \tag{5.11}$$

We define $\mathcal{O}_n$ in the following 3 cases. In all cases assume K is a knowledge-base, α a (non-empty) path allowed in K, q a primitive path allowed in K, f a fact allowed in K, and p a subset of $C \times R$. We use the shorthand:

$$lookup(q)(K) = \{\theta | (\exists f \in K :: \theta = mgru(q, f))\}. \tag{5.12}$$

Assume K is a knowledge-base such that:

$$\begin{array}{rcl} C & = & \{c\} \\ R & = & \{r_1, r_2\} \\ Nr & = & \{r_1(c,x) \leftarrow r_2(c,x)\} \\ Ar & = & \{\} \\ F & = & \{r_2(c,c)\} \\ P & = & \{\{\langle c, r_1\rangle, \langle c, r_2\rangle\}\} \end{array}$$

Consider $query(r_1(c,x))(K)$ (where x is a variable). This is a primitive operation and $par_K(r_1(c,x)) = \{\langle c, r_1\rangle, \langle c, r_2\rangle\}$, so we must first compute:

$$query_0(r_1(c,x))(K, \{\langle c, r_1\rangle, \langle c, r_2\rangle\})$$

Rule back-chaining is cut off at depth zero so no rules apply and

$$query_0(r_1(c,x))(K, \{\langle c, r_1\rangle, \langle c, r_2\rangle\}) = \langle\{\}, K\rangle$$

(an empty list of substitutions is returned since there is no known value of x for which the query succeeds). However, when we calculate

$$query_1(r_1(c,x))(K, \{\langle c, r_1\rangle, \langle c, r_2\rangle\}),$$

the if-needed rule applies and

$$query_1(r_1(c,x))(K, \{\langle c, r_1\rangle, \langle c, r_2\rangle\}) = \langle\{\{x/c\}\}, K + r_1(c,c)\rangle$$

($\{x/c\}$ binds x to c). As n is increased further there are no other rules to apply so

$$query(r_1(c,x))(K) = \langle\{\{x/c\}\}, K + r_1(c,c)\rangle.$$

Figure 5.3: A query on a simple knowledge-base.

Case 1: Base case: $\mathcal{O}$ is a primitive operation, and $\mathcal{O} \notin p$ or $n = 0$.

If $\mathcal{O} = query(q)$ then

$$\mathcal{O}_n(K, p) = \langle lookup(q)(K), K\rangle \tag{5.13}$$

else $\mathcal{O} = assert(f)$ and

$$\mathcal{O}_n(K, p) = \langle\{\{\}\}, closure(K + f)\rangle. \tag{5.14}$$

Case 2: $\mathcal{O}$ is a primitive operation, $n > 0$, and $\mathcal{O} \in p$.

First we find the rules which apply. Let η be a renaming which maps variables in rules in Nr to variables not used in $\mathcal{O}$ or K (this must be possible since the alphabet contains a countably infinite number of variables).[7]

If $\mathcal{O} = query(q)$ then:

$$R = \{\rho\eta\theta \mid \rho \in Nr \wedge \theta = mgru(Key(\rho)\eta, q)\} \tag{5.15}$$

Else, $\mathcal{O} = assert(f)$ and:

$$R = \{\rho\theta \mid \rho \in Ar \wedge \theta = mgru(Key(\rho), f)\} \tag{5.16}$$

If $R = \emptyset$ then let:

$$K' = kb(\mathcal{O}_{n-1}(K, p)) \tag{5.17}$$

Otherwise, we apply the rules and union the results. Applying a rule consists of querying its antecedent and then asserting its consequent. The consequent is asserted with all substitutions under which the antecedent succeeds:

If $\mathcal{O} = query(q)$ then:

$$\begin{aligned} K' = closure(\bigcup \rho \in R :: kb(query_{n-1}(Ant(\rho))(K, p)) \cup \\ (\bigcup \theta \in sub(query_{n-1}(Ant(\rho))(K, p)) :: \\ kb(assert_{n-1}(Conseq(\rho\theta))(K, p)))) \end{aligned} \tag{5.18}$$

$$\mathcal{O}_n(K, p) = \langle lookup(q)(K'), K'\rangle \tag{5.19}$$

Else, $\mathcal{O} = assert(f)$ and:

$$\begin{aligned} K' = closure(\bigcup \rho \in R :: kb(query_{n-1}(Ant(\rho))(K + f, p)) \cup \\ (\bigcup \theta \in sub(query_{n-1}(Ant(\rho))(K + f, p)) :: \\ kb(assert_{n-1}(Conseq(\rho\theta))(K + f, p)))) \end{aligned} \tag{5.20}$$

$$\mathcal{O}_n(K, p) = \langle\{\{\}\}, K'\rangle \tag{5.21}$$

[7]Technical note: It will be convenient to regard this renaming as a function of $\mathcal{O}$ and K (rather than as a randomly chosen renaming). Such a function could be defined by imposing an ordering on the variables in the language (details left to the reader).

Case 3: Non-primitive queries. $\mathcal{O} = query(\alpha)$, where α a path of length greater than 1. Assume $head(\alpha) = q$, and $rest(\alpha) = \alpha'$. If $sub(query_n(q)(K,p)) = \{\}$ (*i.e.*, $query_n(q)$ 'failed'),

$$\mathcal{O}_n(K,p) = \langle \{\}, K \rangle \tag{5.22}$$

else:

$$\begin{aligned} \mathcal{O}_n(K,p) = closure((\bigcup \theta \in\ & sub(query_n(q)(K,p)) \\ & :: \langle \theta \circ sub(query_n(\alpha'\theta)(K,p)), \\ & \quad kb(query_n(q)(K,p)) \cup kb(query_n(\alpha'\theta)(K,p)) \,\rangle\,)) \end{aligned} \tag{5.23}$$

5.3.10 Implementation Note

There are three important differences between the formal definitions of ALL operations given here and our Lisp implementation (Algernon). First, in the formalism, when an operation branches (*e.g.*, when several rules are applied or when the evaluation of a path branches on several possible instantiations of its variables) the branches are computed separately ('in parallel') and the results are unioned together. In our implementation the branches are computed serially (*i.e.*, one rule is applied and then the next rule is applied in the resultant knowledge-base).[8] There are two advantages of the formalization presented here over a 'serial' formalization. First, the complexity analysis is considerably simplified, and second, the formalism given here would also apply to a parallel implementation of ALL.

The second difference is that our implementation supports limits on the accessibility of rules, which have been omitted (for simplicity) from the current formalization. In our implementation, a rule can be 'associated' with a frame f_0, and only accessed from frames known to be in an *isa* relation with f_0. Intuitively, such rules apply only to members of the set f_0.[9]

Our implementation of *closure* demonstrates a useful application of rules associated with sets. One might worry that closing a knowledge-base might add a large number of if-added rules and thus slow the system (since we have to try to unify against all of them). However, we associate these if-added rules with very small sets (sets of size one) and they are thus ignored except when they are needed. Consider a rule added in the closure of a knowledge-base. It must be of form $rest(\alpha)\theta \rightarrow q\theta$. It follows from the definition of if-added rules (the fact that $\alpha \rightarrow q$ is an if-added rule, and the fact that θ is a ground substitution binding all variables in $head(\alpha)$) that $rest(\alpha)\theta$ must be a path. This implies that $head(rest(\alpha))$ must be of form $r(c, t_1, \ldots, t_n)$. Thus (in our implementation) we simply associate this rule with a set consisting of the single element (frame) c (creating such a set if it does not exist).

The third difference is also related to our implementation of *closure*. Consider an assertion of a fact f. This assertion may trigger an if-added rule which asserts a fact f' into a partition p' which is disjoint from the partitions of f. In our implementation, f' is queued in p' and any if-added

[8]Since ALL operations are monotonic, the serial implementation returns knowledge-bases which are supersets of those given by the formalism. Hence, our completeness results carry over to the serial case.

[9]Such access limitations can be formalized. The key idea is that when a rule associated with a set is translated to predicate calculus (see section 5.4.1) one must prepend an appropriate *isa* relation to its antecedent. It should be possible to show that the completeness results carry over (some care must be taken, however, when defining the *closure* of a knowledge-base with respect to an *isa* relation).

rules for f' are not applied until 'attention' is drawn to p' by some operation in p'. This ensures that the complexity of an operation is a function only of the rules in its partitions. We have not yet incorporated the notion of 'queuing' assertions into our formalism. Thus facts are closed with respect to all if-added rules in the knowledge-base, and the complexity of an operation (as will be seen in section 7) is a function of the set of all if-added rules in the knowledge-base.

5.3.11 Basic Results

In this subsection we prove some basic results about ALL operations which we will need in the proofs of the later theorems.

Theorem 1 *ALL formulas are well-defined.*

Proof: We have to prove that for any $\mathcal{O}$ allowed in KB, $\mathcal{O}$ (as defined above) is a total function:

$$\mathcal{O} : KB \Longrightarrow 2^{\Theta} \times KB.$$

as asserted in equation 5.1. For the necessary inductions to work we also prove that if $\mathcal{O} = query(\alpha)$ (for some path α), and $\mathcal{O}$ allowed in K then:

$$(\forall \theta \in sub(\mathcal{O}(K)) :: \theta\ is\ ground \wedge \alpha\theta\ is\ ground) \tag{5.24}$$

First we assume that $\mathcal{O}_n$ is well defined (as asserted in equation 5.11) and that for all $n \geq 0$ and all $p \subset C \times \mathrm{R}$, if $\mathcal{O} = query(\alpha)$:

$$(\forall \theta \in sub(\mathcal{O}_n(K, p)) :: \theta\ is\ ground \wedge \alpha\theta\ is\ ground), \tag{5.25}$$

(which is just a modified version of 5.24) and prove the result for $\mathcal{O}$. If $\mathcal{O}$ is primitive then well-definedness follows directly from the observation that the union of knowledge-bases is a knowledge-base and the union of sets of substitutions is a set of substitutions. 5.24 follows easily from 5.25. For a non-primitive operation, $\mathcal{O} = query(\alpha)$, we induct on the length of α.

It remains to show that $\mathcal{O}_n$ is a total function:

$$\mathcal{O}_n : KB \times 2^{C \times R} \Longrightarrow 2^{\Theta} \times KB.$$

as asserted in equation 5.11, and that 5.25 holds. To do this we induct on n. The actual induction is just a rather tedious application of Lemmas 2 and 3 to the three cases of the definition of $\mathcal{O}_n$. The only trick is that the induction is first on n and then on the length of α. Details are left to the reader. ∎

Finally, we show the following technical properties of ALL operations:

1. The substitutions returned by a query bind all and only the variables in the query to constants.
2. ALL operations can only increase the size of the knowledge-base.
3. Increasing the depth of rule chaining can only increase the size of the resultant knowledge-base.

4. ALL operations are monotonic.

We prove these results first for $\mathcal{O}_n$ and then (for all but part 3) for $\mathcal{O}$.

Lemma 8 *For any knowledge-base K, any $p \subset C \times R$[10], any operation $\mathcal{O}$ allowed in K, and any integer $n \geq 0$:*

1. *If $\mathcal{O} = query(\alpha)$ then:*

$$(\forall \theta \in sub(\mathcal{O}_n(K, p)) :: \theta\, ground \wedge dom(\theta) = vars(\alpha)).$$

2. *$K \subset kb(\mathcal{O}_n(K, p))$.*

3. *$\mathcal{O}_n(K, p) \subset \mathcal{O}_{n+1}(K, p)$.*

4. *$\mathcal{O}_n$ is monotonic (in its first argument).*

Proof: All parts are shown by induction on n and then on the length of α (part 4 also relies on part 3). ∎

Lemma 9 *For any knowledge-base K and any operation $\mathcal{O}$ allowed in K:*

1. *If $\mathcal{O} = query(\alpha)$ then:*

$$(\forall \theta \in sub(\mathcal{O}(K)) :: \theta\, ground \wedge dom(\theta) = vars(\alpha)).$$

2. *$K \subset kb(\mathcal{O}(K))$.*

3. *$\mathcal{O}$ is monotonic.*

Proof: If $\mathcal{O}$ is primitive the results follow directly from lemma 8. If $\mathcal{O} = query(\alpha)$ is not primitive then the results follow by inductions on the length of α.

5.4 Soundness

In this subsection we show that inference in ALL is consistent. That is, all inferences made in ALL are justified by the contents of the knowledge-base. To define soundness in ALL we first define the semantics of ALL by showing a mapping from ALL to (first order) Predicate Calculus (PC).

Mapping ALL to predicate calculus is straightforward. Propositions do not change. Paths become conjunctions. Rules become implications with all variables universally quantified, and knowledge-bases become the conjunction of their rules and facts.

[10] Any partition of a knowledge-base is a subset of $C \times R$, but not vice-verse. We prove our lemmas for any subset of $C \times R$ so that they will apply to partitions, unions of partitions, partitions that no one has thought of yet, and so on.

5.4.1 Mapping ALL to Predicate Calculus

We map ALL to Predicate Calculus by the $\mathcal{PC}$ function. We overload $\mathcal{PC}$ in that it maps ALL propositions, rules and knowledge-bases to Predicate Calculus. An overview of the notation of predicate calculus, as used in this paper, is provided in the appendix (section B.1).

For any ALL proposition, q, $\mathcal{PC}(q) = q$. For a list of propositions, α, if $\alpha = nil$ then $\mathcal{PC}(\alpha) =$ **true** else:

$$\mathcal{PC}(\alpha) = \mathcal{PC}(head(\alpha)) \wedge \mathcal{PC}(rest(\alpha)).$$

Consider any rule ρ. Let $\{v_1, \ldots, v_n\} = vars(\rho)$.

$$\mathcal{PC}(\rho) = (\forall v_1 : (\forall v_2 : \ldots (\forall v_n : \mathcal{PC}(Ant(\rho)) \rightarrow \mathcal{PC}(Conseq(\rho))) \ldots)). \tag{5.26}$$

If S is a set of ALL propositions or rules then $\mathcal{PC}(S) = (\bigwedge s \in S :: \mathcal{PC}(s))$. For K a knowledge-base, $\mathcal{PC}(K) = \mathcal{PC}(Nr) \wedge \mathcal{PC}(Ar) \wedge \mathcal{PC}(F)$. Note that $\mathcal{PC}$ maps knowledge-bases to sentences in predicate calculus.

5.4.2 Soundness Theorem.

Soundness is often intuitively thought of as "You can't derive a contradiction." Soundness requires that the substitutions returned by query must be semantic consequences of the old knowledge-base.[11] The requirements on the new knowledge-base are more subtle. Soundness requires that propositions do not suddenly become true, or, in model theoretic terms, that models are not suddenly lost. Thus, in a query, any model of the old knowledge-base must still be a model of the new knowledge-base, and in an assertion any model of the old knowledge-base and of the fact asserted must still be a model of the new knowledge-base:

Theorem 2 (Soundness of ALL) *For any knowledge-base K, any path α allowed in K, and any fact f allowed in K:*

1. $(\forall \theta \in \Theta : \theta \in sub(query(\alpha)(K)) : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta))$
2. $\mathcal{PC}(K) \models \mathcal{PC}(kb(query(\alpha)(K)))$
3. $(\mathcal{PC}(K) \wedge \mathcal{PC}(f)) \models \mathcal{PC}(kb(assert(f)(K)))$

Proof: Since inference in ALL only involves the application of rules, soundness is not a surprise. However the formal proof of soundness is an important check on the definition of ALL operations. The proof is not really difficult, but it involves keeping careful track of details while working through the definition of $\mathcal{O}$. Details can be found in the appendix, section B.2.

[11] More precisely, for any query of a path α, if θ is returned then $\mathcal{PC}(\alpha\theta)$ must be a consequence of the knowledge-base.

Assume $K = \langle C, R, Nr, Ar, F, P\rangle$ is a knowledge-base such that:

$$\begin{aligned}
C &= \{c\} \\
R &= \{r_1, r_2, r_3\} \\
Nr &= \{r_1(c, x) \leftarrow r_2(c, x)\} \\
Ar &= \{r_1(c, x) \rightarrow r_3(c, x)\} \\
F &= \{r_2(c, c)\} \\
P &= \{\{\langle c, r_1\rangle, \langle c, r_2\rangle, \langle c, r_3\rangle\}\}
\end{aligned}$$

Consider $query(r_3(c, x))(K)$. This query must fail since it matches no facts in F and there are no *if-needed* rules for $r_3(c, x)$. But, any model of $\mathcal{PC}(K)$ must be a model of $\mathcal{PC}(r_3(c, c))$ (by the two rules and the fact that $r_2(c, c)$ is in F). Hence, inference in ALL is not complete.

Figure 5.4: A form of incompleteness in ALL.

5.5 Completeness

Completeness can be thought of as "Any fact which is a model theoretic consequence is provable." Thus completeness requires that all substitutions which are semantic consequences of the old knowledge-base are returned by query. Completeness also requires that true facts do not suddenly become false. In model theoretic terms this means that we do not gain models. Thus any model of the new knowledge-base must also have been a model of the old knowledge-base. Note that the requirements for completeness are essentially the requirements for soundness with their implications reversed:

Conjecture 1 (Completeness of ALL) *For any knowledge-base K, any path α allowed in K, and any fact f allowed in K:*

1. $(\forall \theta \in \Theta_\alpha : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta) : \theta \in sub(query(\alpha)(K)))$
2. $\mathcal{PC}(kb(query(\alpha)(K)) \models \mathcal{PC}(K)$
3. $\mathcal{PC}(kb(assert(f)(K))) \models (\mathcal{PC}(K) \wedge \mathcal{PC}(f))$

(Recall that Θ_α, defined in section 5.3.2, is the set of all ground substitutions binding all and only variables in α.) Unfortunately, part one of this conjecture is false. In some cases, rules necessary for a query to succeed cannot be accessed. Two such cases are shown in the figures 5.4 and 5.5.

Notice, however, that in the example in figure 5.4:

$$query(r_3(c, x))(kb(query(r_1(c, x))(K)))$$

Assume $K = \langle C, R, Nr, Ar, F, P \rangle$ is a knowledge-base such that:

$$\begin{aligned}
C &= \{c\} \\
R &= \{r_1, r_2, r_3\} \\
Nr &= \{r_1(c,x) \leftarrow r_2(c,x), r_2(c,x) \leftarrow r_3(c,x)\} \\
Ar &= \{\} \\
F &= \{r_3(c,c)\} \\
P &= \{\{\langle c, r_1 \rangle\}, \\
&\quad \{\langle c, r_2 \rangle, \langle c, r_3 \rangle\}\}
\end{aligned}$$

Consider $query(r_1(c,x))(K)$. This query must fail since the only rule for $r_1(c,x)$ depends on $r_2(c,x)$, which matches no facts in F and is not in $par_K(r_1(c,x))$ (so no rules for $r_2(c,x)$ fire). But, any model of $\mathcal{PC}(K)$ must be a model of $\mathcal{PC}(r_1(c,c))$ (by the two rules and the fact that $r_3(c,c)$ is in F).

Figure 5.5: Another form of incompleteness in ALL.

would succeed, since $r_3(c,c)$ is added to $kb(query(r_1(c,x))(K))$ by the if-added rule $r_1(c,x) \rightarrow r_3(c,x)$. Similarly, in figure 5.5,

$$query(r_1(c,x))(kb(query(r_2(c,x))(K)))$$

succeeds. This suggests the idea behind *Socratic Completeness.* Informally, the Socratic Completeness Theorem says that for any query of α which 'should' succeed in a knowledge-base, there exists a series of preliminary queries Γ, after which a query of α will succeed. We also show a second type of weakened completeness result, *Partitional Completeness.* Partitional Completeness says that if all information needed to process a query can be located by the if-needed rules in the partitions of the query, then the query will succeed.

An alternative approach would be to define a model theory for ALL, in terms of which ALL is complete. This could be done, but we believe that (since the model theory of predicate calculus is well understood), mapping to predicate calculus and appropriately weakening the notion of completeness gives a more perspicuous picture of the semantics of ALL. Further, we believe that soundness and Socratic Completeness relative to predicate calculus (or perhaps an appropriate non-monotonic logic) are necessary properties for any knowledge representation system.

Intuitively, the proof of the Socratic Completeness Theorem breaks down into two steps:

1. Show that for any rule whose antecedent is currently known (*i.e.*, is in the knowledge-base) there exists a query which will cause the rule to fire (adding its consequent to the knowledge-base).

2. Show that the repeated applications of rules is sufficient to eventually cause any fact which is a model theoretic consequence of the knowledge-base to be added to the knowledge-base.

Part 2 is a known result from the study of Logic Programming. Thus before we prove Socratic Completeness we review some of the notation and results from Logic Programming.

5.5.1 Some Results From the Study of Logic Programming

This section is essentially a summary of the relevant material in [Apt, 88].

A *logic program* consists of a set of *program clauses* of form:

$$a \leftarrow b_1, \ldots, b_n$$

where a and b_i $(1 \leq i \leq n)$ are propositions[12]. Such a clause is understood to be equivalent to the predicate calculus statement:

$$(\forall x_1 : \ldots (\forall x_m : a \leftarrow b_1, \ldots, b_n))$$

where $x_1, \ldots, x_m$ are all variables appearing in a and $b_1, \ldots, b_n$.

Let L be a first order language (with alphabet as given in section B.1) whose set of constants is not empty. Let the *Herbrand Universe* U_L for L be the set of all ground terms (*i.e.*, constants — since ALL does not allow functions) in L. Let the *Herbrand Base* B_L for L be the set of all ground propositions (*i.e.*, facts) in L. By a *Herbrand Model* for L we mean a model $\mathcal{H}$ such that:

1. The domain of $\mathcal{H}$ is the Herbrand universe U_L.
2. Each constant $c \in L$ is assigned to itself ($c_{\mathcal{H}} = c$).
3. If r is an n-ary relation in L then it is assigned to a set of n-tuples of ground terms of L.

Thus each Herbrand model for L is uniquely determined by the assignment to relations of sets of n-tuples of ground terms of L. Such an assignment can be equated with a subset I of the Herbrand base by the mapping:

$$\langle t_1, \ldots, t_n \rangle \in r_{\mathcal{H}} \Longleftrightarrow r(t_1, \ldots, t_n) \in I.$$

Thus we can identify Herbrand models with subsets of the Herbrand base. Further, if K is a knowledge-base, then $\mathcal{PC}(F)$ is a subset of the Herbrand base. Thus *we can identify Herbrand models with knowledge-bases.* Finally, observe that for any fact f: K (viewed as a Herbrand model) $\models f$ iff $f \in K$.

For a logic program P, the *immediate consequence operator*, T_P, maps Herbrand models to Herbrand models by:

$$f \in T_P(I) \Longleftrightarrow (\exists \theta \in \Theta, a \leftarrow b_1, \ldots, b_n \in P : f = a\theta : I \models (b_1 \wedge \ldots \wedge b_n)\theta).$$

Intuitively, T_P applies each rule in P once. It is shown in [Apt, 88] that T_P is monotonic. We can define powers of T_P by:

$$\begin{aligned} T_P \uparrow 0(I) &= I \\ T_P \uparrow (n+1)(I) &= T_P(T_P \uparrow n(I)) \\ T_P \uparrow \omega(I) &= (\bigcup n : n > 0 : T \uparrow n(I)) \end{aligned}$$

[12] Actually logic programs allow terms to be variables, constants or functions applied to terms. Thus a and the b_i can have a somewhat more general form than ALL propositions.

We abbreviate $T_P \uparrow n(\emptyset)$ by $T_P \uparrow n$.

Finally, the result we need for our completeness results below is found in the proof of theorem 3.25 in [Apt, 88]. For a ground proposition a and a logic program P:

$$P \models a \Longleftrightarrow a \in T_P \uparrow \omega. \tag{5.27}$$

5.5.2 Mapping ALL Knowledge-Bases to Logic Programs

In order to use this result we need a way to map ALL knowledge-bases to logic programs. To do this we define the $\mathcal{LP}$ operator. $\mathcal{LP}$ (like $\mathcal{PC}$) maps ALL facts, rules, and knowledge-bases in an ALL language, to logic programs in a first order language with the same alphabet as the ALL language. For any fact f, $\mathcal{LP}(f) = f \leftarrow$. For a rule ρ, $\mathcal{LP}(\rho) = \rho$. If S is a set of facts or rules then $\mathcal{LP}(S) = \{\mathcal{LP}(s) \mid s \in S\}$. Finally, if K is a knowledge-base then $\mathcal{LP}(K) = \mathcal{LP}(F) \cup \mathcal{LP}(Ar) \cup \mathcal{LP}(Nr)$. Note that it follows from the constructions of $\mathcal{PC}$ and $\mathcal{LP}$ that $\mathcal{PC}(K)$ is the predicate calculus equivalent of $\mathcal{LP}(K)$.

5.5.3 Socratic Completeness

We first prove Socratic Completeness for primitive queries and then generalize to paths.

The proofs below rely on the following lemmas. The first lemma says simply that asserting a fact into a knowledge-base adds the fact to the knowledge-base:

Lemma 10 *If f is a fact allowed in a knowledge-base K, p is a subset of $C \times R$, and $n \geq 0$:*

1. $f \in kb(assert_n(f)(K, p))$.
2. $f \in kb(assert(f)(K))$.

Proof: Part 1 is shown by induction on n. Part 2 follows from part 1. ∎

The second lemma says that there always exists a series of queries which 'simulates' the immediate consequence operator. We first extend our notation to represent queries of a series of paths. If Γ is a series of paths allowed in a knowledge-base K then let:

$$\begin{aligned} query(nil)(K) &= K && (5.28)\\ query(\Gamma)(K) &= query(head(\Gamma))(kb(query(rest(\Gamma)(K)))) && (5.29) \end{aligned}$$

The proof of this lemma is somewhat technical and is given in the appendix (section B.3).

Lemma 11 *For any knowledge-base K there exists a series of variable free paths Γ (allowed in K) such that $T_{\mathcal{LP}(K)}(K) \subset kb(query(\Gamma)(K))$.*

Finally, we can show Socratic Completeness for primitive queries, and then for all queries:

Lemma 12 (Socratic Completeness for Primitive Queries) *For any knowledge-base K and any primitive path α allowed in K:*

$$\begin{array}{lll}(\forall\theta \in \Theta_\alpha & : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta) & \\ & : (\exists\Gamma & : \Gamma \textit{ a series of variable free paths allowed in } K \\ & & : \theta \in sub(query(\alpha)(kb(query(\Gamma)(K))))))\end{array}$$

Proof: By the result above from logic programming (5.27) we know for any fact f:

$$f \in T_{\mathcal{LP}(K)} \uparrow \omega \Longleftrightarrow \mathcal{PC}(K) \models f.$$

Thus it suffices to show:

$$\begin{array}{lll}(\forall\theta \in \Theta_\alpha & : \mathcal{PC}(\alpha\theta) \in T_{\mathcal{LP}(K)} \uparrow \omega & \\ & : (\exists\Gamma & : \Gamma \text{ a series of variable free paths allowed in } K \\ & & : \theta \in sub(query(\alpha)(kb(query(\Gamma)(K))))))\end{array}$$

But this follows from:

$$\begin{array}{ll}(\forall f & : f \in T_{\mathcal{LP}(K)} \uparrow \omega \\ & : (\exists\Gamma : \Gamma \text{ a series of variable free paths allowed in } K : f \in kb(query(\Gamma)(K))))\end{array}$$

By definition of $T_P \uparrow \omega$, $f \in T_{\mathcal{LP}(K)} \uparrow \omega$ implies that for some k, $f \in T_{\mathcal{LP}(K)} \uparrow k$. Thus it suffices to show for arbitrary k that there exists some series of variable free paths Γ (allowed in K) such that:

$$T_{\mathcal{LP}(K)} \uparrow k \subset kb(query(\Gamma)(K)) \tag{5.30}$$

(note that we are again using the identification of knowledge-bases with Herbrand models discussed above in section 5.5.1). This we prove by induction on k. The base case of this induction is trivial since $T_{\mathcal{LP}(K)} \uparrow 0 = \emptyset$. In the inductive case we know by induction that for any knowledge-base K, there exists some series of variable free paths δ such that:

$$\begin{array}{l}T_{\mathcal{LP}(K)} \uparrow (k-1) \subset kb(query(\delta)(K)) \Longrightarrow \\ \{ \text{ monotonicity of } T_{\mathcal{LP}(K)} \} \\ T_{\mathcal{LP}(K)} \uparrow k \subset T_{\mathcal{LP}(K)}(kb(query(\delta)(K))) \Longrightarrow \\ \{ \text{ Lemma 11 with } K = kb(query(\delta)(K))\} \\ T_{\mathcal{LP}(K)} \uparrow k \subset kb(query(\gamma)(kb(query(\delta)(K)))) \Longrightarrow \\ \{ \text{ Let } \Gamma = append(\gamma, \delta) \} \\ T_{\mathcal{LP}(K)} \uparrow k \subset kb(query(\Gamma)(K))\end{array} \qquad \blacksquare$$

Theorem 3 (Socratic Completeness) *For any knowledge-base K, any path α allowed in K, and any fact f allowed in K:*

1. $(\forall\theta \in \Theta_\alpha : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta)$

$$: (\exists\Gamma : \Gamma \textit{ a series of paths allowed in } K : \theta \in sub(query(\alpha)(kb(query(\Gamma)(K))))))$$

2. $\mathcal{PC}(kb(query(\alpha)(K))) \models \mathcal{PC}(K)$

3. $\mathcal{PC}(kb(assert(f)(K))) \models (\mathcal{PC}(K) \wedge \mathcal{PC}(f))$

Proof: Observe first that $K_1 \subset K_2 \implies F_1 \subset F_2$ and $Ar_1 \subset Ar_2$. Thus (by definitions of $\models$ and $\mathcal{PC}$) $K_1 \subset K_2 \implies \mathcal{PC}(K_2) \models \mathcal{PC}(K_1)$. 2 follows from this observation and the observation that $query(\alpha)$ cannot remove facts from K (by lemma 9 part 2). A part of 3 (specifically $\mathcal{PC}(kb(assert(f)(K))) \models (\mathcal{PC}(K)))$ follows from the similar observation that $assert(\alpha)$ cannot remove facts from K (again by lemma 9 part 2). The rest of 3 follows from lemma 10 part 2. For part 1 we induct on the length of α. The base case is just lemma 12. In the inductive case the trick is to apply the induction hypothesis to $head(\alpha)$, then to $rest(\alpha)\theta\backslash_{vars(head(\alpha))}$, and finally to note that (since θ is ground):

$$\theta = \theta\backslash_{vars(head(\alpha))} \circ \theta\backslash_{vars(rest(\alpha))}.$$

∎

5.5.4 Partitional Completeness

Intuitively, Partitional Completeness says that if all information needed to prove a query is located 'close enough' to the query in the knowledge-base, then the query will succeed. 'Close enough' will mean that the information is reachable using only if-needed rules in the partitions of the query. Thus, to state the Partitional Completeness Theorem, we first have to define carefully which rules in the knowledge-base are considered 'part' of which partitions. Intuitively, a rule is considered a part of a partition if it could be triggered by queries to, and assertions into, the frame-slots in that partition. Counterintuitively, partitions limit access to *rules* not facts. When we speak of a fact as being in a partition, we mean that the fact is queried (or asserted) using the rules in that partition. Theoretically a rule could access facts in every partition of the knowledge-base; it is the use of access paths, not partitions, which limits access to facts.

If the head of a rule is of form $r(c, t_1, \ldots, t_n)$ then clearly the rule is in a partition iff $\langle c, r\rangle$ is in the partition. If the head of the rule is of form $r(v, t_1, \ldots, t_n)$ then for every frame-slot in the partition which is of form $\langle c, r\rangle$ (for any $c \in C$), we instantiate the rule by binding v to c. Formally, if K is a knowledge-base, $p \subset C \times R$, and S is a set of rules from K, then $S\backslash_p$, the restriction of S to p, is given by:

$$\begin{aligned} S\backslash_p = \{\rho\theta \mid\ & \rho \in S \wedge Key(\rho) = r(t_1, \ldots, t_n) \wedge \\ & ((t_1 \textit{ a constant} \wedge \theta = \emptyset \wedge \langle t_1, r\rangle \in p) \vee \\ & \ (t_1 \textit{ a variable} \wedge (\exists c \in C : \langle c, r\rangle \in p : \theta = \{t_1/c\})))\}. \end{aligned}$$

The knowledge-base K with its rules restricted to only the if-needed rules in $p \subset C \times R$ is given by:

$$K\backslash_p = \langle C, R, Nr\backslash_p, \emptyset, F, \{p\}\rangle. \tag{5.31}$$

Note that $K\backslash_p$ is never computed (in the definition of ALL formulas or in our Lisp implementation); It is only a formal object used to state the Partitional Completeness Theorem.

In order to prove Partitional Completeness we use the following key lemma. Intuitively, this lemma says that if a fact is an instantiation of q, and is in the knowledge-base produced by n iterations of the immediate consequence operator, then it is in the knowledge-base produced by $query_n(q)$. The proof is fairly technical and is given in the appendix (section B.3).

Lemma 13 *For any knowledge-base K, any primitive path q allowed in K, any $p \subset C \times R$:*

$$ground(q) \cap T_{\mathcal{LP}(K\backslash_p)} \uparrow n \subset kb(query_n(q)(K,p)).$$

Theorem 4 (Partitional Completeness) *For any knowledge-base K and any primitive path q allowed in K:*

$$(\forall \theta \in \Theta_q : \mathcal{PC}(K\backslash_{par_K(q)}) \models \mathcal{PC}(q\theta) : \theta \in sub(query(q)(K)))$$

Proof: From the result above from logic programming (equation 5.27) it suffices to show for all K,q:

$$(\forall \theta \in \Theta_q : \mathcal{PC}(q\theta) \in T_{\mathcal{LP}(K\backslash_{par_K(q)})} \uparrow \omega : \theta \in sub(query(q)(K)))$$

But this follows (from the definitions, and the observation that $q\theta$ is ground) from:

$$(\forall f \in ground(q) : f \in T_{\mathcal{LP}(K\backslash_{par_k(q)})} \uparrow \omega : f \in kb(query(q)(K)))$$

By definition of $T_P \uparrow \omega$ and $\mathcal{O}$ it thus suffices to prove (for all $n > 0$):

$$(\forall f \in ground(q) : f \in T_{\mathcal{LP}(K\backslash_{par_k(q)})} \uparrow n : f \in kb(query_n(q)(K, par_K(q))))$$

Which is a special case of lemma 13. ∎

Chapter 6

Adding Negation to ALL

Ultimately we are working towards a formal theory which has the expressive power of predicate calculus, and is sound and Socratically Complete, but is still computationally tractable. It is straightforward to add to ALL the ability to express full classic negation (*i.e.*, not negation by failure), but then inference in ALL (using rules alone) is no longer Socratically Complete. In this chapter we formalize in ALL the notion of reasoning by *reductio ad absurdum*. Reasoning by *reductio ad absurdum* involves adding assumptions to the knowledge-base and then reasoning about their consequences (and if the consequences of an assumption include 'false' concluding the negation of the assumption). We show that ALL with negation is Socratically Complete. Further, the queries determine what assumptions are made, so the complexity of ALL is still polynomial (though some hard problems may require an exponential amount of work to determine what queries and assumptions to make).

The formalism here uses *query* and *assert*, as defined in chapter five, as 'subroutines' to build a more complex logic capable or representing and reasoning with negation. We refer to the previous formalism as ALL and the formalism introduced in this chapter as ALL(+neg).

6.1 Two Examples

We begin with a pair of examples. The first example simply involves reasoning by *modus ponens*, and the desired inference can be made easily. The second example is a bit more complex and the desired inference requires some preliminary queries in ALL and some thought for a person, but would follow as easily as the inference in the first example using resolution. We would like to eventually argue that ALL reasons efficiently in the cases people find easy and needs help in the cases people find hard. At least in these examples this is true; in example 1, ALL can conclude immediately that Mary is happy, but requires several leading questions to conclude that the flowers in example 2 do not bloom:

Example 1:

1. If Mary waters the flowers then the flowers bloom.
2. If the flowers bloom then Mary is happy.
3. Mary waters the flowers.

Example 2:

1. If Mary waters the flowers and the flowers bloom then Mary is happy.
2. If Mary does not water the flowers then they do not bloom.
3. Mary is not happy.

Both examples are fairly small and simple. Yet (at least to me), it is obvious that in example 1 Mary is happy, but it is not obvious that in example 2 the flowers do not bloom.

Formally, we can let:

p = Mary waters the flowers
q = The flowers bloom
r = Mary is happy

The first example becomes:

1. $p \to q$.

2. $q \to r$.

3. p.

From which r easily follows (with two rule applications or a two step resolution proof).

The second example becomes:

1. $p \land q \to r$.

2. $\neg p \to \neg q$.

3. $\neg r$.

In this case $\neg q$ follows by a two step resolution proof. In ALL one must make two assumptions. One must first assume q, and then (within the context of this assumption) assume p. Together these assumptions imply r, which is a contradiction. By *reductio ad absurdum*, this contradiction gives us $\neg p$. But $\neg p$ implies $\neg q$ and this contradicts the assumption q. Thus, by *reductio ad absurdum* again, we get $\neg q$. More formally we have:

4. $assume(q)$.

5. $assume(p)$.

6. r by 1,4, and 5.

7. $\neg p$ by 3,6, and *reductio ad absurdum*.

8. $\neg q$ by 2 and 7.

9. $\neg q$ by 4, 8 and *reductio ad absurdum*.

Note that though the example is fairly short, the proof is non-trivial. This may partially explain why the result is not immediately obvious.

6.2 Formal Development

This section presents the full formal development of ALL with negation. It is organized into the following subsections:

1. Syntax of ALL(+neg).
2. Knowledge Theory.
3. Consistency.
4. Socratic Completeness.

6.2.1 Syntax of ALL(+neg)

ALL(+neg) differs from ALL primarily in that it allows one to make assumptions and then reason by *reductio ad absurdum.* When an assumption is made the current state of the knowledge-base is saved so that if the assumption leads to a contradiction it can be restored. Thus, in general ALL(+neg) operations operate on stacks of knowledge-bases which we refer to as *knowledge-base structures.*

Basic Notation and Definitions

Conventions for variable names are the same as in ALL except that $\mathcal{K}$ will be used to denote a knowledge-base structure.

We use the function *negate* to return the negation of a relation or proposition in the obvious way: $negate(r) = \neg r$, $negate(\neg r) = r$, $negate(r(t_1, \ldots, t_n)) = \neg r(t_1, \ldots, t_n)$ and $negate(\neg r(t_1, \ldots, t_n)) = r(t_1, \ldots, t_n)$. The distinction between $\neg$ and *negate* is important: $\neg$ is simply a character which can occur in names of relations, while *negate* is a function; $\neg$ is a part of ALL while *negate* is a part of the mathematical language used to define ALL.

For any set of facts F, F is *inconsistent* iff there exists a fact f such that $f \in F$ and $negate(f) \in F$. A knowledge-base is *f-inconsistent* iff its set of facts is inconsistent.[1]

Knowledge-Base Structures

A *knowledge-base structure*, or *kbs* for short, $\mathcal{K}$, is a stack of pairs:

$$\langle a_n, K_n \rangle, \ldots, \langle a_1, K_1 \rangle$$

such that, for $1 \leq i \leq n$, K_i is a knowledge-base and a_i is *nil* or a fact allowed in K_i. As for list, we access the top element of such a stack using the function *head* and rest of the stack using *rest*. Further, $height(\mathcal{K})$ will be the number of pairs in the structure. For any pair $\langle a, K \rangle$ we access the first and second components using the functions *assump* and *kb* respectively. We also use *kb* and *assump* to access the 'top' knowledge-base and assumption in a structure; for any kbs $\mathcal{K}$:

$$kb(\mathcal{K}) = kb(head(\mathcal{K}))$$

[1] Since knowledge-bases are not deductively closed, a knowledge-base may be inconsistent (*i.e.*, it may not have a model – see section 6.2.3), but not be f-inconsistent.

$$assump(\mathcal{K}) = assump(head(\mathcal{K})).$$

We denote the structure formed by adding the pair $\langle a, K\rangle$ to the top of the the structure $\mathcal{K}$ by:

$$push(\langle a, K\rangle, \mathcal{K}).$$

There are several invariants which will hold for knowledge-base structures in which we will be interested. First, all knowledge-bases in the stack should have the same constants, relations, if-needed rules, and partitions. Second, if any relation is allowed in knowledge-base then its negation must also be allowed. Third, going up the stack, the facts and if-added rules should only increase. Fourth, for each pair, the assumption in the pair should be a fact in the corresponding knowledge-base. Finally, all knowledge-bases in the structure should be f-consistent. We refer to a kbs satisfying these invariants as *well-formed.*

Formally, a kbs:

$$\mathcal{K} = \langle a_n, K_n\rangle, \ldots, \langle a_1, K_1\rangle$$

is *well-formed* iff:

1. $C_1 = C_2 = \ldots = C_n$.
2. $R_1 = R_2 = \ldots = R_n$.
3. $Nr_1 = Nr_2 = \ldots = Nr_n$.
4. $P_1 = P_2 = \ldots = P_n$.
5. $(\forall i : 1 \leq i \leq n : (\forall r \in R_i :: negate(r) \in R_i))$.
6. $(\forall i : 1 \leq i < n : F_{i+1} \supset F_i)$.
7. $(\forall i : 1 \leq i < n : Ar_{i+1} \supset Ar_i)$.
8. $(\forall i : 1 \leq i \leq n : a_i \in K_i)$.
9. $(\forall i : q \leq i \leq n : K_i$ f-consistent $)$.

Unless otherwise noted all knowledge-base structures can be assumed to be well-formed. Note that if $\mathcal{K}$ is a well-formed kbs then $rest(\mathcal{K})$ is also a well-formed kbs.

The well-formedness conditions guarantee that any proposition, list of propositions, or rule allowed in any knowledge-base in a structure is allowed in all knowledge-bases in the structure. Thus we say a proposition, list of propositions or rule is allowed in a kbs iff it is allowed in the knowledge-bases of the kbs.

ALL(+neg) Operations and Formula

To differentiate them from ALL operations, we subscript ALL(+neg) operations by *neg*. Thus $query(q)$ is an ALL operation and operates on a knowledge-base, while $query_{neg}(q)$ is an ALL(+neg) operation which operates on a kbs. The same holds for variables ranging over operations. If $\mathcal{O} = query(q)$ then $\mathcal{O}_{neg}$ is the corresponds ALL(+neg) operation, $query_{neg}(q)$. This distinction is often clear from context (since any operation on a knowledge-base is an ALL operation and any operation on a kbs is an ALL(+neg) operation) so we may omit the subscripts.

If α is a non-empty path then $query_{neg}(\alpha)$ is a *query*. If α is a primitive path then $query_{neg}(\alpha)$ is a *primitive* query. If f is a fact then $assert_{neg}(f)$ is an *assertion* (assertions of paths are not currently allowed). Finally, if a is a fact then $assume_{neg}(a)$ is an *assumption*. Any query, assertion or assumption is an *operation*, and any assertion, assumption, or primitive query is a *primitive* operation. If $\mathcal{O}_{neg} = query_{neg}(\alpha)$, $\mathcal{O}_{neg} = assert_{neg}(f)$, or $\mathcal{O}_{neg} = assume_{neg}(a)$ then $\mathcal{O}_{neg}$ is *allowed* in a kbs $\mathcal{K}$ iff α is allowed in $\mathcal{K}$, f is allowed in $\mathcal{K}$, or a is allowed in $\mathcal{K}$, respectively. If an operation $\mathcal{O}_{neg}$ is allowed in a kbs $\mathcal{K}$ then $\mathcal{O}_{neg}(\mathcal{K})$ is an ALL(+neg) *formula*.

6.2.2 Knowledge Theory

The knowledge theory defines the values of ALL(+neg) formulas by defining the action of operations (*i.e.*, queries, assertions, and assumptions). Intuitively, the knowledge theory of ALL(+neg) differs from ALL only in that assumptions are supported and that contradictions are recognized and dealt with.

The Domain and Range of All Operations

Any given sets C, R, Nr, P define a set of possible knowledge-base structures $\mathcal{KB}$ (differing only in assumptions, facts, and if-added rules), and a set of ground substitutions Θ (binding variables in the alphabet to constants in C). For any operation $\mathcal{O}_{neg}$ allowed in the knowledge-base structures in $\mathcal{KB}$:

$$\mathcal{O}_{neg} : \mathcal{KB} \longrightarrow 2^{\Theta} \times \mathcal{KB}. \tag{6.1}$$

We denote these returned values with pairs: $\langle$ 'set of substitutions', 'knowledge-base structure' $\rangle$. and use *sub* and *kbs* as accessors on the first and second components respectively.

The Values of ALL Operations

We first define $query_{neg}$ and $assert_{neg}$ and then define *assume* in terms of $assert_{neg}$.

Assume that $\mathcal{K}$ is a kbs and $\mathcal{O}_{neg}$ is a query or assertion allowed in $\mathcal{K}$. Queries and assertions in ALL(+neg) consist of two steps. First the operation is performed as in ALL. If the resultant knowledge-base is f-consistent then it replaces the knowledge-base in the top pair on the stack. If, however, the result is an f-inconsistent knowledge-base, then the most recent assumption is retracted. The retraction of an assumption consists of popping the stack and asserting the negation of the assumption (unless the stack has only one element in which case there are no assumptions to drop and the knowledge-base is inconsistent).

Formally, we use the shorthand:

$$retract\_assump(\mathcal{K}) = kbs(assert_{neg}(negate(assump(\mathcal{K})))(rest(\mathcal{K})))$$

For any query or assertion, $\mathcal{O}_{neg}$ allowed in a kbs $\mathcal{K}$:

If $kb(\mathcal{O}(kb(\mathcal{K}))$ f-consistent or $height(\mathcal{K}) = 1$ then:

$$\mathcal{O}_{neg}(\mathcal{K}) = \langle sub(\mathcal{O}(kb(\mathcal{K})), push(\langle assump(\mathcal{K}), kb(\mathcal{O}(kb(\mathcal{K})))\rangle, rest(\mathcal{K}))\rangle$$

else (an inconsistency has been found and there is an assumption to drop):

$$\mathcal{O}_{neg}(\mathcal{K}) = \langle \emptyset, retract\_assump(\mathcal{K}) \rangle$$

Finally, making an assumption consists of adding a new pair to the stack and then asserting the assumption. For any fact a allowed in a kbs $\mathcal{K}$:

$$assume(a)(\mathcal{K}) = assert_{neg}(a)(push(\langle a, kb(\mathcal{K}) \rangle, \mathcal{K}))$$

Figure 6.1 shows a simple example of a proof by contradiction. The example in the introduction is shown in figure 6.2.

Implementation Note — Chronological vs. Dependency Directed Backtracking

The assumption management techniques used in this formalism have been chosen to make the formal development as transparent as possible. More efficient techniques are used in the implementation of ALL. The major difference between the formalism and the implementation is that the formalism uses *chronological* backtracking while the implementation uses *dependency directed* backtracking. In chronological backtracking, when a contradiction is found the most recent assumption is retracted. In dependency directed backtracking, the dependencies of facts in the knowledge-base on assumptions are explicitly maintained (often as labels on the facts [Stallman & Sussman, 77]). When a contradiction is found, an assumption which the fact depends on is retracted. Dependency directed backtracking can greatly reduce the length of proofs, but its formalization is more complex.

Basic Results

The first theorem simply says that well-formedness is preserved by ALL(+neg) operations, and that queries return ground substitutions binding the variables in the path queried:

Theorem 5 *For any well-formed kbs $\mathcal{K}$ and any ALL(+neg) operation $\mathcal{O}$ allowed in $\mathcal{K}$:*

1. $height(\mathcal{K}) \geq height(kbs(\mathcal{O}(\mathcal{K})))$ *unless $\mathcal{O}$ is an assumption.*
2. $kbs(\mathcal{O}(\mathcal{K}))$ *is a well-formed kbs.*
3. *If $\mathcal{O}_{neg} = query_{neg}(\alpha)$ (for some path α) then:*

$$(\forall \theta \in sub(\mathcal{O}_{neg}(K)) :: \theta \; is \; ground \wedge \alpha\theta \; is \; ground)$$

Proof: We will need one additional definition: A kbs is *almost well-formed* iff conditions 1-7 and 9, in the definition of well-formed hold, and:

$$(\forall i : 1 \leq i < n : a_i \in K_i).$$

Informally, a kbs is almost well-formed if all conditions hold except condition 8 with $i = n$.

Consider the knowledge-base structure $\mathcal{K} = \langle nil, K\rangle$ where K is a knowledge-base such that:

$$\begin{aligned}
C &= \{c\} \\
R &= \{r_1, r_2\} \\
Nr &= \{r_1(c,c) \leftarrow r_2(c,c), \neg r_1(c,c) \leftarrow r_2(c,c)\} \\
Ar &= \{\} \\
F &= \{\} \\
P &= \{\{\langle c, r_1\rangle, \langle c, r_2\rangle\}\}
\end{aligned}$$

Note that logically this knowledge-base implies $\neg r_2(c,c)$, but that no possible sequence of rule applications will ever make this deduction. However, consider what happens if we assume $r_2(c,c)$, and then query $r_1(c,c)$ and $\neg r_1(c,c)$. First, let $\mathcal{K}_1 = kbs(assume(r_2(c,c))(\mathcal{K}))$. From the definitions above:

$$\begin{aligned}
\mathcal{K}_1 &= kbs(assert_{neg}(r_2(c,c))(\langle r_2(c,c), K\rangle, \langle nil, K\rangle)) \\
&= \langle r_2(c,c), K + r_2(c,c)\rangle, \langle nil, K\rangle
\end{aligned} \tag{6.2}$$

Then let $\mathcal{K}_2 = kbs(query_{neg}(r_1(c,c))(\mathcal{K}_1))$. The query succeeds so:

$$\mathcal{K}_2 = \langle r_2(c,c), K + r_2(c,c) + r_1(c,c)\rangle, \langle nil, K\rangle$$

Finally, let $\mathcal{K}_3 = kbs(query_{neg}(\neg r_1(c,c))(\mathcal{K}_2))$. In this case $query(\neg r_1(c,c))(kb(\mathcal{K}_2))$ results in an inconsistent knowledge-base. Thus:

$$\begin{aligned}
\mathcal{K}_3 &= retract_assump(\mathcal{K}_2) \\
&= \langle nil, K + \neg r_2(c,c)\rangle
\end{aligned} \tag{6.3}$$

Thus we have deduced $\neg r_2(c,c)$ and $query_{neg}(\neg r_2(c,c))(\mathcal{K}_3)$ would succeed.

Figure 6.1: A simple proof by contradiction.

We express the situation in example 2 in the introduction using the kbs $\mathcal{K} = \langle nil, K\rangle$, where K is:

$$
\begin{aligned}
C &= \{Mary, Flowers\} \\
R &= \{waters, blooms, happy\} \\
Nr &= \{(happy(Mary) \leftarrow waters(Mary), blooms(Flowers)), \\
&\quad (\neg blooms(Flowers) \leftarrow \neg waters(Mary))\} \\
Ar &= \{\} \\
F &= \{\neg happy(Mary)\} \\
P &= \{\{\langle Mary, waters\rangle, \langle Mary, blooms\rangle, \langle Mary, happy\rangle, \\
&\quad \langle Flowers, waters\rangle, \langle Flowers, blooms\rangle, \langle Flowers, happy\rangle\}\}
\end{aligned}
$$

Let $\mathcal{K}_1 = kbs(assume(blooms(Flowers))(\mathcal{K}))$. Then:

$$
\begin{aligned}
\mathcal{K}_1 &= kbs(assert_{neg}(blooms(Flowers))(\langle blooms(Flowers), K\rangle, \langle nil, K\rangle)) \\
&= \langle blooms(Flowers), K + blooms(Flowers)\rangle, \langle nil, K\rangle
\end{aligned}
$$

Similarly, let $\mathcal{K}_2 = kbs(assume(waters(Mary))(\mathcal{K}_1))$. Then:

$$
\begin{aligned}
\mathcal{K}_2 &= \langle waters(Mary), K + blooms(Flowers) + waters(Mary)\rangle, \\
&\quad \langle blooms(Flowers), K + blooms(Flowers)\rangle, \\
&\quad \langle nil, K\rangle
\end{aligned}
$$

Let $\mathcal{K}_3 = kbs(query_{neg}(happy(Mary))(\mathcal{K}_2))$. $query(happy(Mary))(kb(\mathcal{K}_2))$ results in an inconsistent knowledge-base. Thus:

$$
\begin{aligned}
\mathcal{K}_3 &= retract\_assump(\mathcal{K}_2) \\
&= \langle blooms(Flowers), K + blooms(Flowers) + \neg waters(Mary)\rangle, \langle nil, K\rangle
\end{aligned}
$$

Let $\mathcal{K}_4 = kbs(query_{neg}(\neg blooms(Flowers))(\mathcal{K}_3))$.
Again $query(\neg blooms(Flowers))(kb(\mathcal{K}_3))$ results in an inconsistent knowledge-base, and:

$$
\begin{aligned}
\mathcal{K}_4 &= retract\_assump(\mathcal{K}_3) \\
&= \langle nil, K + \neg blooms(Flowers)\rangle
\end{aligned}
$$

Figure 6.2: Proving that the flowers do not bloom.

Consider first the case in which $\mathcal{O}$ is a query or an assertion. In this case we show that 1-3 hold, and we also show that if $\mathcal{K}$ is merely almost well-formed then 1 and 3 still hold, and $kbs(\mathcal{O}(\mathcal{K}))$ is almost well-formed.

The proof is by induction on the height of $\mathcal{K}$. In the base case either the height of $\mathcal{K}$ is 1, or $kb(\mathcal{O}(kb(\mathcal{K})))$ is f-consistent. In this case:

$$\mathcal{O}_{neg}(\mathcal{K}) = \langle sub(\mathcal{O}(kb(\mathcal{K})), push(\langle assump(\mathcal{K}), kb(\mathcal{O}(kb(\mathcal{K})))\rangle, rest(\mathcal{K}))\rangle$$

and the necessary results all follow easily (using similar results for ALL from chapter 5). In the inductive case $height(\mathcal{K}) > 1$ and $kb(\mathcal{O}(kb(\mathcal{K})))$ is not f-consistent. This implies:

$$\mathcal{O}_{neg}(\mathcal{K}) = \langle \emptyset, retract\_assump(\mathcal{K})\rangle$$

Thus 3 is trivially true. Further, $\mathcal{K}$ well-formed (or almost well-formed) implies that $rest(\mathcal{K})$ is well formed. By definition:

$$retract\_assump(\mathcal{K}) = kbs(assert_{neg}(negate(assump(\mathcal{K})))(rest(\mathcal{K}))).$$

so 1 and 2 follow by induction.

Finally, consider the case in which $\mathcal{O}$ is an assumption. The only difficulty in this case is that:

$$assume(a)(\mathcal{K}) = assert_{neg}(a)(push(\langle a, kb(\mathcal{K})\rangle, \mathcal{K}))$$

and $push(\langle a, kb(\mathcal{K})$ is not necessarily well-formed. However, it is almost well-formed and so (by case 1 above):

$$assert_{neg}(a)(push(\langle a, kb(\mathcal{K})\rangle, \mathcal{K}))$$

must be almost well-formed and 1 and 3 must hold. To show it is well-formed it only remains to show:

$$assump(kbs(assume(a)(\mathcal{K}))) \in kb(kbs(assume(a)(\mathcal{K}))).$$

If $assert(a)(kb(\mathcal{K}))$ is f-consistent then:

$$assump(kbs(assume(a)(\mathcal{K}))) = a$$

and

$$kb(kbs(assume(a)(\mathcal{K}))) = assert(a)(kb(\mathcal{K}))$$

but we know by lemma 10 that $a \in assert(a)(kb(\mathcal{K}))$. If $assert(a)(kb(\mathcal{K}))$ is f-inconsistent then:

$$assume(a)(\mathcal{K}) = \langle \emptyset, retract\_assump(push(\langle a, kb(\mathcal{K})\rangle, \mathcal{K}))\rangle$$

and so:

$$assume(a)(\mathcal{K}) = \langle \emptyset, kbs(assert_{neg}(negate(a))(\mathcal{K}))\rangle$$

which must be well-formed by case 1. ∎

6.2.3 Soundness

Mapping to Predicate Calculus

Mapping ALL(+neg) to predicate calculus is straightforward; each knowledge-base in the stack is believed under its assumption and all assumptions 'under' it. For any kbs $\mathcal{K}$ such that $\mathcal{K} = \langle a_n, K_n\rangle, \ldots, \langle a_1, K_1\rangle$:

$$\mathcal{PC}(\mathcal{K}) = (\bigwedge i : 1 \leq i \leq n : (\bigwedge j : 1 \leq j \leq i : \mathcal{PC}(a_j)) \Rightarrow \mathcal{PC}(K_i))$$

A kbs K is *consistent* iff there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \mathcal{PC}(\mathcal{K})$. Note that if a kbs $\mathcal{K}$ is f-inconsistent then it is inconsistent, but not necessarily vice-versa.

Soundness Theorem

Theorem 6 (Soundness of ALL(+neg)) *Consider any well-formed kbs $\mathcal{K}$, any path α allowed in $\mathcal{K}$, and any fact f allowed in $\mathcal{K}$. Assume $\mathcal{K} = \langle a_n, K_n\rangle, \ldots, \langle a_1, K_1\rangle$, then:*

1. $(\forall \theta \in \Theta : \theta \in sub(query_{neg}(\alpha)(\mathcal{K})) : (\mathcal{PC}(\mathcal{K}) \wedge (\bigwedge i : 1 \leq i \leq n : a_i)) \models \mathcal{PC}(\alpha\theta))$
2. $\mathcal{PC}(\mathcal{K}) \models \mathcal{PC}(kbs(query_{neg}(\alpha)(\mathcal{K})))$
3. $(\mathcal{PC}(\mathcal{K}) \wedge ((\bigwedge i : 1 \leq i \leq n : \mathcal{PC}(a_i)) \rightarrow \mathcal{PC}(f))) \models \mathcal{PC}(kbs(assert_{neg}(f)(\mathcal{K})))$
4. $\mathcal{PC}(\mathcal{K}) \models \mathcal{PC}(kbs(assume(f)(\mathcal{K})))$

Proof: We first show 3 by induction on the height of $\mathcal{K}$ (as in the proof of theorem 5). The base case is straightforward from the definitions (and theorem 2 part 3). In the inductive case $kb(assert(kb(\mathcal{K})))$ is f-inconsistent and:

$$kbs(assert_{neg}(f)(\mathcal{K})) = kbs(assert_{neg}(negate(a_n))(rest(\mathcal{K})))$$

Consider any model $\mathcal{M}$ such that:

$$\mathcal{M} \models (\mathcal{PC}(\mathcal{K}) \wedge ((\bigwedge i : 1 \leq i \leq n : \mathcal{PC}(a_i)) \rightarrow \mathcal{PC}(f))))$$

Clearly $\mathcal{M} \models \mathcal{PC}(rest(\mathcal{K}))$ so by induction need only show:

$$\mathcal{M} \models ((\bigwedge i : 1 \leq i < n : \mathcal{PC}(a_i)) \rightarrow \mathcal{PC}(negate(a_n))).$$

To see this, assume its contrapositive:

$$\mathcal{M} \models (\bigwedge i : 1 \leq i \leq n : \mathcal{PC}(a_i)).$$

This implies $\mathcal{M} \models \mathcal{PC}(\mathcal{K})$ and $\mathcal{M} \models \mathcal{PC}(f)$, which implies, by theorem 2 part 3, that:

$$\mathcal{M} \models kb(assert(f)(kb(\mathcal{K}))$$

which is impossible since $kb(assert(kb(\mathcal{K})))$ is f-inconsistent.

The proof for 2 is similar, and the proof for 1 is straightforward (by induction on the height of $\mathcal{K}$ using theorem 2 part 1). Finally, 4 is straightforward from 3 (using the definitions and a little predicate calculus). ∎

6.2.4 Completeness

In this section we show two theorems. For knowledge-base structures we show the inverse of the requirements for soundness. These requirements are fairly weak, only saying that models cannot be gained and thus that information is not lost. We then show Socratic Completeness.

Requirements for Knowledge-base Structures

Theorem 7 *Consider any well-formed kbs $\mathcal{K}$, any path α allowed in $\mathcal{K}$, and any fact f allowed in $\mathcal{K}$. Assume $\mathcal{K} = \langle a_n, K_n \rangle, \ldots, \langle a_1, K_1 \rangle$, then:*

1. $\mathcal{PC}(kbs(query_{neg}(\alpha)(\mathcal{K}))) \models \mathcal{PC}(\mathcal{K})$
2. $\mathcal{PC}(kbs(assert_{neg}(f)(\mathcal{K}))) \models (\mathcal{PC}(\mathcal{K}) \wedge ((\bigwedge i : 1 \leq i \leq n : \mathcal{PC}(a_i)) \rightarrow \mathcal{PC}(f)))$
3. $\mathcal{PC}(kbs(assume(f)(\mathcal{K}))) \models \mathcal{PC}(\mathcal{K})$

Proof: We first show 2 by induction on the height of $\mathcal{K}$ (as in the proof of theorem 5). The base case is straightforward from the definitions (and theorem 3 part 3). In the inductive case $kb(assert(kb(\mathcal{K})))$ is f-inconsistent and:

$$kbs(assert_{neg}(f)(\mathcal{K})) = kbs(assert_{neg}(negate(a_n))(rest(\mathcal{K})))$$

Consider any model $\mathcal{M}$ such that

$$\mathcal{M} \models \mathcal{PC}(kbs(assert_{neg}(f)(\mathcal{K}))).$$

This implies:

$$\mathcal{M} \models \mathcal{PC}(kbs(assert_{neg}(negate(a_n))(rest(\mathcal{K})))).$$

Which implies, by induction, that $\mathcal{M} \models \mathcal{PC}(rest(\mathcal{K})$, and that:

$$\mathcal{M} \models ((\bigwedge i : 1 \leq i < n : \mathcal{PC}(a_i)) \rightarrow \mathcal{PC}(negate(a_n)))$$

which implies that:

$$\mathcal{M} \not\models (\bigwedge i : 1 \leq i \leq n : \mathcal{PC}(a_i)).$$

From which the result follows.

1 is similar, and 3 is straightforward from the definitions (and 2). ∎

Ordering Knowledge-base Structures and Other Technical Results

It will prove important in the next section to have an ordering on knowledge-base structures. We first define the 'natural' lexicographic metric, $\leq$, show that it is a partial order, and show that all ALL(+neg) operations can only return larger knowledge-base structures. However, ALL(+neg) operations are not monotonic under the ordering $\leq$. We, therefore, define a second metric, $\trianglelefteq$, under which operations are monotonic. The proofs in the next sections will make use of both metrics.

Intuitively, $\mathcal{K}_1 < \mathcal{K}_2$ if we can match up the knowledge-bases in the two stacks and they are equal below some height j, but at height j the knowledge-base in $\mathcal{K}_2$ is larger than the knowledge-base in $\mathcal{K}_1$. However, there is one complication. It is useful to think of a kbs with height greater than one as only an intermediate state in a proof by contradiction; assumptions are only made so that contradictions can be derived and the negation of the assumptions proven. Thus assumptions in a kbs can be thought of as 'assertions waiting to happen'. So, when we compare two knowledge-base structures we compare the knowledge-bases at level j plus the assumptions at level $j+1$ (see definition below). One final twist is that a shorter kbs which matches up along its entire height to a taller one is considered to be $>$ the taller kbs (this is like alphabetic order in which "air" comes before "airplane").

Def 1 *For any integer j, and any knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ such that:*

$$\begin{aligned}\mathcal{K}_1 &= \langle a_n^1, K_n^1\rangle, \ldots, \langle a_1^1, K_1^1\rangle \\ \mathcal{K}_2 &= \langle a_m^2, K_m^2\rangle, \ldots, \langle a_1^2, K_1^2\rangle\end{aligned}$$

(to simplify the notation below, for any $i > n$ let $a_i^1 = nil$ and for any $i > m$ let $a_i^2 = nil$), $\mathcal{K}_1 \leq_j \mathcal{K}_2$ iff all of the following hold:

1. $j \leq n$ *and* $j \leq m$.
2. $(\forall i : 1 \leq i < j : a_i^1 = a_i^2 \wedge K_i^1 + negate(a_{i+1}^1) = K_i^2 + negate(a_{i+1}^2))$.
3. $K_j^1 + negate(a_{j+1}^1) \leq K_j^2 + negate(a_{j+1}^2)$.
4. $a_j^1 = a_j^2$.
5. $(K_j^1 + negate(a_{j+1}^1) = K_j^2 + negate(a_{j+1}^2)) \rightarrow j = m$.

Def 2 *For any knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$, $\mathcal{K}_1 \leq \mathcal{K}_2$ iff $(\exists j :: \mathcal{K}_1 \leq_j \mathcal{K}_2)$.*

Lemma 14 *$\leq$ is a partial order over the set of well-formed knowledge-base structures.*

Proof: We need to show that $\leq$ is transitive, reflexive, and antisymmetric. Transitivity is relatively straightforward. Reflexivity is almost immediate from the definitions.

For antisymmetry, consider two knowledge-base structures, $\mathcal{K}_1$ and $\mathcal{K}_2$ as in the definition of $\leq$. Assume $\mathcal{K}_1 \leq_{j_1} \mathcal{K}_2$ and $\mathcal{K}_2 \leq_{j_2} \mathcal{K}_1$. One can show (from the definitions) that this implies $height(\mathcal{K}_1) = height(\mathcal{K}_2)$ and that:

$$(\forall i : 1 \leq i \leq height(\mathcal{K}_1) : a_i^1 = a_i^2 \wedge K_i^1 + negate(a_{i+1}^1) = K_i^2 + negate(a_{i+1}^2)).$$

Finally, using the well-formedness of $\mathcal{K}_1$ and $\mathcal{K}_2$ one can show this implies:

$$(\forall i : 1 \leq i \leq height(\mathcal{K}_1) : a_i^1 = a_i^2 \wedge K_i^1 = K_i^2).$$

∎

Lemma 15 *For any well-formed kbs $\mathcal{K}$ and any operation $\mathcal{O}$ allowed in $\mathcal{K}$:*

1. *$\mathcal{K} \leq_j kbs(\mathcal{O}(\mathcal{K}))$ (and unless $\mathcal{O}$ an assumption $j = height(kbs(\mathcal{O}(\mathcal{K})))$).*
2. *If $height(\mathcal{K}) > 1$ then $\mathcal{K} <_j retract\_assump(\mathcal{K})$ with $j = height(retract\_assump(\mathcal{K}))$.*

Proof: First assume $\mathcal{O}$ a query or assertion. We induct on the height of $\mathcal{K}$ (as in the proof of theorem 5). The base case is straightforward. In the inductive case we first show 2. By induction (and the definition of *retract_assump*):

$$rest(\mathcal{K}) \leq_j retract\_assump(\mathcal{K}).$$

with $j = height(retract\_assump(\mathcal{K})$. Let $\mathcal{K}_1 = \mathcal{K}$ and $\mathcal{K}_2 = retract\_assump(\mathcal{K})$. Need only show $\mathcal{K}_1 \leq \mathcal{K}_2$. Parts 1,3,4, and 5 (in the definition of $\leq$) are straightforward. In part 2 need show $K_j^1 + n(a_{j+1}^1) \subseteq K_j^2$. This can be shown for any $\mathcal{K}_1$ and $\mathcal{K}_2$ such that $\mathcal{K}_2 = retract\_assump(\mathcal{K}_1)$ by induction on the height of $\mathcal{K}_1$. Part 1 of the lemma then follows easily.

Now assume $\mathcal{O} = assume(f)$. Here there are two cases:

Case 1: $kb(assert(f)(kb(\mathcal{K})))$ f-consistent. In this case:

$$assume(f)(\mathcal{K}) = push(\langle f, assert(f)(kb(\mathcal{K}))\rangle, \mathcal{K}).$$

Further, $negate(f) \notin kb(\mathcal{K})$ (else $kb(assert(f)(kb(\mathcal{K})))$ would be f-inconsistent), from which it follows that $\mathcal{K} \leq assume(f)(\mathcal{K})$.

Case 2: $kb(assert(f)(kb(\mathcal{K})))$ f-inconsistent. In this case the assumption immediately leads to a contradiction and:

$$assume(f)(\mathcal{K}) = assert(negate(f))(\mathcal{K}).$$

From which it follows by the result for *assert* above that $\mathcal{K} \leq assume(f)(\mathcal{K})$.

∎

We can extend this result to hold for any series of ALL(+neg) operations. Assume $\mathcal{K}$ is a kbs and Γ is a series of operations allowed in $\mathcal{K}$. If $length(\Gamma) = 0$ then let $\Gamma(\mathcal{K}) = \mathcal{K}$ else let:

$$\Gamma(\mathcal{K}) = kbs(head(\Gamma)(rest(\Gamma)(\mathcal{K}))).$$

Notice that this means that:

$$(\mathcal{O}_1, \ldots, \mathcal{O}_n)(\mathcal{K}) = kbs(\mathcal{O}_1(\ldots kbs(\mathcal{O}_n(\mathcal{K}))\ldots))$$

Corollary 1 *For any well-formed kbs $\mathcal{K}$ and any series of operations Γ allowed in $\mathcal{K}$:*

$$\mathcal{K} \leq \Gamma(\mathcal{K}).$$

Then we have the second metric for ordering knowledge-base structures. It is similar to $\leq$, but is carefully chosen so that it will be preserved under ALL(+neg) operations (*i.e.*, to guarantee monotonicity). One way to remember the intuitive meaning of $\trianglelefteq$ is that a 'tall thin kbs' is $\trianglelefteq$ a 'short fat kbs':

Def 3 *For any knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ such that:*

$$\begin{aligned}\mathcal{K}_1 &= \langle a^1_n, K^1_n\rangle, \ldots, \langle a^1_1, K^1_1\rangle \\ \mathcal{K}_2 &= \langle a^2_m, K^2_m\rangle, \ldots, \langle a^2_1, K^2_1\rangle\end{aligned}$$

(for any $i > n$ let $a^1_i = nil$ and for any $i > m$ let $a^2_i = nil$), $\mathcal{K}_1 \trianglelefteq \mathcal{K}_2$ iff all of the following hold:

1. *$n \geq m$.*
2. *$(\forall i : 1 \leq i \leq m : a^1_i = a^2_i \wedge K^1_i = K^2_i)$.*
3. *$assert(negate(a^1_{m+1})(K^1_m)) \leq K^2_m$.*

The proof of the following two lemmas are straightforward and are left to the reader.

Lemma 16 *$\trianglelefteq$ is a partial order over the set of well-formed knowledge-base structures.*

Lemma 17 *For any well-formed knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$, if*

$$height(\mathcal{K}_1) = height(\mathcal{K}_2) = 1$$

then

$$\mathcal{K}_1 \leq \mathcal{K}_2 \Leftrightarrow \mathcal{K}_1 \trianglelefteq \mathcal{K}_2.$$

One property of $\trianglelefteq$ is that one can perform ALL operations on the 'top' of the 'tall thin kbs' without effecting the ordering:

Lemma 18 *For any well-formed knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ and any operation $\mathcal{O}$ allowed in $\mathcal{K}_1$, if $\mathcal{K}_1 \trianglelefteq \mathcal{K}_2$ and $height(\mathcal{K}_1) > height(\mathcal{K}_2)$ then:*

$$kbs(\mathcal{O}(\mathcal{K}_1)) \trianglelefteq \mathcal{K}_2.$$

Proof: Induct on the height of $\mathcal{K}_1$ (as in the proof of theorem 5). The only trick is to use part 3 of the definition of $\trianglelefteq$ in the induction step. ∎

Lemma 19 *For any well-formed knowledge-base structure $\mathcal{K}$, if $height(\mathcal{K}) > 1$ then:*

$$\mathcal{K} \trianglelefteq retract\_assump(\mathcal{K}).$$

Proof: Part 1 (of the definition of $\trianglelefteq$) is easy (using theorem 5 part 1). For parts 2 and 3, induct on the height of $rest(\mathcal{K})$. ∎

The key lemma for $\trianglelefteq$ is monotonicity. It only holds, however, for two knowledge-base structures of equal height:

Lemma 20 *For any well-formed knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ and any operation $\mathcal{O}$ allowed in $\mathcal{K}_1$ and $\mathcal{K}_2$, if $\mathcal{K}_1 \trianglelefteq \mathcal{K}_2$ and $height(\mathcal{K}_1) = height(\mathcal{K}_2)$ then:*

$$kbs(\mathcal{O}(\mathcal{K}_1)) \trianglelefteq kbs(\mathcal{O}(\mathcal{K}_2)).$$

Proof: First consider the case in which $\mathcal{O}$ is a query or assertion. We induct on the height of $\mathcal{K}_1$ (as in the proof of theorem 5). In the base case, if $height(\mathcal{K}_1) = 1$ then $height(\mathcal{K}_2) = 1$ and the result follows from the definitions (and the monotonicity of ALL operations). If $height(\mathcal{K}_1) > 1$ and $kb(\mathcal{O}(kb(\mathcal{K}_1)))$ f-consistent then we induct on the height of $\mathcal{K}_2$. The base case (of the second induction) is straightforward. In the inductive case (for the second induction):

$$kbs(\mathcal{O}(\mathcal{K}_2)) = retract\_assump(\mathcal{K}_2),$$

and the result follows by application of lemma 19 and then lemma 18. In the inductive case (for the induction on the height of $\mathcal{K}_1$), $kb(\mathcal{O}(kb(\mathcal{K}_1)))$ f-inconsistent implies $kb(\mathcal{O}(kb(\mathcal{K}_2)))$ f-inconsistent so both operations result in an assumption being retracted and the result follows by induction.

The case in which $\mathcal{O}$ is an assumption follows easily from the assertion case. ∎

Finally, we show a weak version of monotonicity for serieses of operations. This will be the key lemma in the proof of the next section:

Lemma 21 *For any well-formed knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ and any series of operations Γ_1 allowed in $\mathcal{K}_1$, if $\mathcal{K}_1 \trianglelefteq \mathcal{K}_2$ then there exists a series of operations allowed in $\mathcal{K}_2$, Γ_2, such that:*

$$\Gamma_1(\mathcal{K}_1) \trianglelefteq \Gamma_2(\mathcal{K}_2).$$

Proof: Induct on the length of Γ_1. The base case and the inductive case both follow by application of lemma 20 (if the heights of the knowledge-base structures are equal) or lemma 18 (if $height(\mathcal{K}_1)$ is greater than $height(\mathcal{K}_2)$). ∎

Provability in ALL(+neg)

As discussed in the introduction, reasoning in ALL(+neg) is not complete, but it is Socratically Complete; for any query which is a logical consequence of a knowledge-base there exists a series of queries and assumptions, after which the query will succeed. In order to formally prove this we first define provability in ALL(+neg). Notice that, in keeping with the intuition that knowledge-base structures with height greater than one are only intermediate steps in a proof, we define provability for structures of height one:

Def 4 *Given a kbs $\mathcal{K} = \langle a, K \rangle$, and any ground path $f_1, \ldots, f_n$ allowed in $\mathcal{K}$:*

$$\mathcal{K} \vdash_{ALL} f_1, \ldots, f_n$$

iff there exists a series of queries and assumptions allowed in $\mathcal{K}$, Γ, such that:

$$\Gamma(\mathcal{K}) = \langle a, K' \rangle \wedge (\forall i : 1 \leq i \leq n : f_i \in K').$$

In the remainder of this section we show that the inference rules *reductio ad absurdum* and *modus ponens* hold for $\vdash_{ALL}$. In the next section we use these results to prove Socratic Completeness.

We first extend the definition of $\vdash_{ALL}$ to the case where the height of the kbs is greater than one; a kbs proves a fact iff the top knowledge-base in the kbs proves the fact:

Def 5 *Given a kbs $\mathcal{K}$, and a ground path $f_1, \ldots, f_n$ allowed in $\mathcal{K}$:*

$$\mathcal{K} \vdash_{ALL} f_1, \ldots, f_n \leftrightarrow \langle nil, kb(\mathcal{K})\rangle \vdash_{ALL} f_1, \ldots, f_n.$$

The primary technical problem in working with provability in ALL(+neg) is that a series of operations which derives a fact in a kbs may not be sufficient to derive the fact in a larger kbs. An example of this problem is shown in figure 6.3. However, we can show (using lemma 21) that a series of operation for the larger kbs does exist (in fact it is equal to or shorter than the series for the smaller kbs, though we do not prove this):

Lemma 22 *For any well-formed knowledge-base structures $\mathcal{K}_1$ and $\mathcal{K}_2$ and any fact f allowed in $\mathcal{K}_1$ and $\mathcal{K}_2$, if $\mathcal{K}_1 \vdash_{ALL} f$, $\mathcal{K}_1 \trianglelefteq \mathcal{K}_2$ and $height(\mathcal{K}_1) = height(\mathcal{K}_2)$ then:*

$$\mathcal{K}_2 \vdash_{ALL} f.$$

Proof: Straightforward using lemma 21 and the definition of $\trianglelefteq$. ∎

Using this lemma we can show that if $\mathcal{K}$ proves f_1 and f_2 then it proves the path f_1, f_2.

Lemma 23 *For any well-formed kbs $\mathcal{K}$ and any facts f_1 and f_2 allowed in $\mathcal{K}$, if $\mathcal{K} \vdash_{ALL} f_1$ and $\mathcal{K} \vdash_{ALL} f_2$ then:*

$$\mathcal{K} \vdash_{ALL} f_1, f_2.$$

Proof: Note first that for any path α, $\mathcal{K} \vdash_{ALL} \alpha$ iff $\langle nil, kb(\mathcal{K})\rangle \vdash_{ALL} \alpha$. Thus it suffices to show result for knowledge-base structures of height one.

Let Γ_1 be series for $\mathcal{K} \vdash_{ALL} f_1$. By corollary 1, $\mathcal{K} \leq \Gamma_1(\mathcal{K})$ which implies by lemma 17 that $\mathcal{K} \trianglelefteq \Gamma_1(\mathcal{K})$. Thus, by lemma 22, $\Gamma_1(\mathcal{K}) \vdash_{ALL} f_2$. Let Γ_2 be the appropriate series. One can then easily verify that $\mathcal{K} \vdash_{ALL} f_1, f_2$ using the series $append(\Gamma_2, \Gamma_1)$. ∎

Consider any series of operations on a kbs. Intuitively, the operations must either only effect the 'top' of the kbs, or must cause assumptions to be retracted all the way down to the bottom of the kbs. We show this formally in the next two lemmas. These two lemmas are a bit technical and their proofs are largely omitted.

Lemma 24 *Consider any well-formed kbs $\mathcal{K}$ such that:*

$$\mathcal{K} = \langle a_n, K_n\rangle, \ldots, \langle a_1, K_1\rangle$$

such that $n \geq 2$. For any operation $\mathcal{O}$ allowed in $\mathcal{K}$, one of the following must hold:

Consider the example from figure 6.2. Recall that the initial kbs was $\mathcal{K} = \langle nil, K\rangle$ where K was:

$$\begin{aligned}
C &= \{Mary, Flowers\}\\
R &= \{waters, blooms, happy\}\\
Nr &= \{(happy(Mary) \leftarrow waters(Mary), blooms(Flowers)),\\
&\quad\ (\neg blooms(Flowers) \leftarrow \neg waters(Mary))\}\\
Ar &= \{\}\\
F &= \{\neg happy(Mary)\}\\
P &= \{\{\langle Mary, waters\rangle, \langle Mary, blooms\rangle, \langle Mary, happy\rangle,\\
&\qquad \langle Flowers, waters\rangle, \langle Flowers, blooms\rangle, \langle Flowers, happy\rangle\}\}
\end{aligned}$$

Recall, further, that to derive $\neg blooms(Flowers)$ we used the series:

$$\begin{aligned}
\Gamma &= query_{neg}(\neg blooms(Flowers), query_{neg}(happy(Mary)),\\
&\quad assume(waters(Mary)), assume(blooms(Flowers))
\end{aligned}$$

and that

$$\Gamma(\mathcal{K}) = \langle nil, K + \neg blooms(Flowers)\rangle.$$

However, consider the kbs $\mathcal{K}' = \langle nil, K + \neg blooms(Flowers)\rangle$. Notice that $\mathcal{K} \trianglelefteq \mathcal{K}'$. Clearly no operations are required to derive $\neg blooms(Flowers)$, but consider what would happen if we applied Γ:

$$\begin{aligned}
\Gamma(\mathcal{K}') &= \langle waters(Mary), K + \neg blooms(Flowers) + waters(Mary)\rangle,\\
&\quad \langle nil, K + \neg blooms(Flowers)\rangle
\end{aligned}$$

Which is not even of height one (and no future query or assumption will be able to force the assumption to be retracted).

Figure 6.3: Provability in ALL(+neg) — larger knowledge-base structures require smallerserieses of operations.

1. $kbs(\mathcal{O}(\mathcal{K})) = append(kbs(\mathcal{O}(\langle a_n, K_n\rangle, \ldots, \langle a_2, K_2\rangle)), \langle a_1, K_1\rangle)$.

2. $kbs(\mathcal{O}(\mathcal{K})) = kbs(assert(negate(a_2))(\langle a_1, K_1\rangle))$.

Proof: By induction on the height of $\mathcal{K}$. ▮

Lemma 25 *Consider any well-formed kbs $\mathcal{K}$ such that:*

$$\mathcal{K} = \langle a_n, K_n\rangle, \ldots, \langle a_1, K_1\rangle$$

such that $n \geq 2$. For any series of operation Γ allowed in $\mathcal{K}$, one of the following must hold:

1. $\Gamma(\mathcal{K}) = append(\Gamma(\langle a_n, K_n\rangle, \ldots, \langle a_2, K_2\rangle), \langle a_1, K_1\rangle)$.

2. *There exists some serieses of operations Γ_1 and Γ_2 such that $\Gamma = append(\Gamma_2, \Gamma_1)$ and:*

$$\Gamma_1(\mathcal{K}) = kbs(assert(negate(a_2))(\langle a_1, K_1\rangle))$$

Proof: By induction on n using lemma 24 in the induction step. ▮

Finally we can prove the two main lemmas for this subsection. For a kbs $\mathcal{K}$ and a fact f (allowed in $\mathcal{K}$) we use the shorthand:

$$\mathcal{K} + f = push(\langle assump(\mathcal{K}), kb(\mathcal{K}) + f\rangle, rest(\mathcal{K})).$$

Lemma 26 (*Reductio Ad Absurdum) For any well-formed kbs $\mathcal{K}$ and any fact a allowed in $\mathcal{K}$, if there exists a fact f allowed in $\mathcal{K}$ such that:*

$$(\mathcal{K} + a \vdash_{ALL} f) \wedge (\mathcal{K} + a \vdash_{ALL} negate(f))$$

then:

$$\mathcal{K} \vdash_{ALL} negate(a).$$

Proof: Note first that, as in the proof of lemma 23, we may assume wlog that the height of $\mathcal{K}$ is one. Let $\mathcal{K}' = assume(a)(\mathcal{K})$. If $kb(assert(a)(kb(\mathcal{K})))$ f-inconsistent then $\mathcal{K}' = \langle nil, assert(negate(a))(kb(\mathcal{K}))\rangle$ and trivially $\mathcal{K} \vdash_{ALL} negate(a)$. Thus assume $kb(assert(a)(kb(\mathcal{K})))$ f-consistent. So:

$$\mathcal{K}' = \langle a, assert(a)(kb(\mathcal{K}))\rangle, \langle nil, kb(\mathcal{K})\rangle.$$

Note that $\mathcal{K} + a \trianglelefteq \langle nil, assert(a)(kb(\mathcal{K}))\rangle$. Thus by lemma 22:

$$\langle nil, assert(a)(kb(\mathcal{K}))\rangle \vdash_{ALL} f.$$

Let Γ_1 be the necessary series of operations. Consider $\Gamma_1(\mathcal{K}')$. By lemma 25 there are only two possibilities. The simple case (possibility 2 in the lemma) is when a contradiction is found and $negate(a)$ asserted by some sub-series of Γ_1. In this case one can easily show that $\mathcal{K} \vdash_{ALL} negate(a)$. The other possibility is that Γ_1 completes (without finding a contradiction). In this case there is some resultant knowledge-base K_f such that:

1. $\Gamma_1(\mathcal{K}') = \langle a, K_f\rangle, \langle nil, kb(\mathcal{K})\rangle$.

2. $assert(a)(kb(\mathcal{K})) \subseteq K_f$.

3. $f \in K_f$.

Notice that $\mathcal{K} + a \trianglelefteq \langle nil, K_f\rangle$, and so:

$$\langle nil, K_f\rangle \vdash_{ALL} negate(f).$$

Let Γ_2 be the necessary series of operations and consider $\Gamma_2(\langle a, K_f\rangle, \langle nil, kb(\mathcal{K})\rangle)$. Again, by lemma 25, there are two possibilities, but in this case a contradiction is necessarily drawn (otherwise we would end up with a knowledge-base containing both f and $negate(f)$), and $negate(a)$ is asserted. ▮

Lemma 27 (*Modus Ponens*) *For any well-formed kbs $\mathcal{K}$, any (if-added or if-needed) rule $\rho \in kb(\mathcal{K})$ such that $Ant(\theta) = b_1, \ldots, b_n$, and any ground substitution θ such that $vars(\rho) \subset domain(\theta)$, if:*

$$(\forall i : 1 \leq i \leq n : \mathcal{K} \vdash_{ALL} b_i\theta)$$

then:

$$\mathcal{K} \vdash_{ALL} Conseq(\rho)\theta.$$

Proof: Assume wolg that the height of $\mathcal{K}$ is one. Note that by lemma 23 and induction on n:

$$\mathcal{K} \vdash_{ALL} Ant(\rho)\theta.$$

Let Γ be the necessary path, and let $\mathcal{K}' = \Gamma(\mathcal{K})$. Let $K' = kb(\mathcal{K}')$. Note that:

$$(\forall i : 1 \leq i \leq n : b_i\theta \in K')$$

Thus, by definition of the immediate consequence operator (see page 125):

$$Conseq(\rho)\theta \in T_{\mathcal{LP}(K')}(K').$$

So, by lemma 11 there exists a series of queries allowed in $\mathcal{K}'$, Γ', such that:

$$Conseq(\rho)\theta \in \Gamma'(\mathcal{K}').$$

Thus $\mathcal{K} \vdash_{ALL} Conseq(\rho)\theta$ using the series of operations $append(\Gamma', \Gamma)$. ▮

Socratic Completeness

Proving Socratic Completeness is now only a matter of using lemmas 26 and 27 to show that $\mathcal{PC}(\mathcal{K}) \models \mathcal{PC}(f)$ implies $\mathcal{K} \vdash_{ALL} f$. This can be shown using a standard Henkin style proof. A brief sketch is given below (a detailed and readable Henkin style proof is given in [Hunter, 71] pp. 105 - 114).

Def 6 *A kbs $\mathcal{K}$ is* p-consistent *in ALL iff there is no fact f allowed in $\mathcal{K}$ such that $\mathcal{K} \vdash_{ALL} f$ and $\mathcal{K} \vdash_{ALL} negate(f)$.*

Lemma 28 *For any well-formed kbs $\mathcal{K}$ and any fact f allowed in $\mathcal{K}$:*

$$\mathcal{K} + negate(a) \textit{ p-inconsistent} \Longleftrightarrow \mathcal{K} \vdash_{ALL} a.$$

Proof: $\Rightarrow$ follows using lemma 26. For $\Leftarrow$, note that $\mathcal{K} \vdash_{ALL} a$ implies (by lemma 22) $\mathcal{K} + negate(a) \vdash_{ALL} a$. Thus $\mathcal{K} + negate(a)$ p-inconsistent. ∎

Def 7 *A kbs $\mathcal{K}$ is* maximally *p-consistent in ALL iff $\mathcal{K}$ is p-consistent in ALL and for any fact f allowed in $\mathcal{K}$: either $f \in \mathcal{K}$ or $\mathcal{K} + f$ is p-inconsistent.*

Lemma 29 *If $\mathcal{K}$ is a well-formed maximally p-consistent kbs and f is a fact allowed in $\mathcal{K}$ then exactly one of f and $negate(f)$ is in $\mathcal{K}$.*

Proof: If both f and $negate(f)$ are in $\mathcal{K}$ then $\mathcal{K}$ is not p-consistent. If neither are in $\mathcal{K}$ then, by definition of maximal p-consistency and lemma 28, $\mathcal{K} \vdash_{ALL} f$ and $\mathcal{K} \vdash_{ALL} negate(f)$, and $\mathcal{K}$ is not p-consistent. ∎

Lemma 30 *If $\mathcal{K}$ is a well-formed maximally p-consistent kbs and f is a fact allowed in $\mathcal{K}$ then $\mathcal{K} \vdash_{ALL} f \Rightarrow f \in \mathcal{K}$.*

Proof: If $\mathcal{K} \vdash_{ALL} f$ and $f \notin \mathcal{K}$ then, by lemma 29, $negate(f) \in \mathcal{K}$. Thus $\mathcal{K}$ not p-consistent. ∎

Lemma 31 (Lindenbaum's Lemma for ALL) *If $\mathcal{K}$ is a well-formed p-consistent kbs then there exists a well-formed maximally p-consistent kbs, $\mathcal{K}'$ such that $\mathcal{K} \trianglelefteq \mathcal{K}'$ and $height(\mathcal{K}) = height(\mathcal{K}')$.*

Proof: Let $f1, \ldots, f_n$ be the facts allowed in $\mathcal{K}$ (this list must be finite because only relations in R and constants in C are allowed). Let $\mathcal{K}_0 = \mathcal{K}$. For any $1 \leq i \leq n$, if $\mathcal{K}_i + f_i$ p-consistent then let $\mathcal{K}_{i+1} = \mathcal{K}_i + f_i$, otherwise let $\mathcal{K}_{i+1} = \mathcal{K}$. Let $\mathcal{K}' = \mathcal{K}_n$. By induction on n, $\mathcal{K}'$ p-consistent, $\mathcal{K} \trianglelefteq \mathcal{K}'$, and $height(\mathcal{K}) = height(\mathcal{K}')$. Need only show $\mathcal{K}'$ maximal. Consider any $f_i \notin \mathcal{K}'$. By construction of $\mathcal{K}'$, $\mathcal{K}_{i-1} + f_i$ p-inconsistent. But by lemma 22 this implies that $\mathcal{K}' + f_i$ p-inconsistent. ∎

Lemma 32 *Any well-formed p-consistent kbs of height one is consistent.*

Proof: Let $\mathcal{K}$ be a well-formed p-consistent kbs of height one. By lemma 31 there exists a well-formed maximally p-consistent kbs, $\mathcal{K}'$, of height one such that $\mathcal{K} \trianglelefteq \mathcal{K}'$. Note that $\mathcal{K}$ must be consistent if $\mathcal{K}'$ is consistent. We thus have only to show there exists a model for $\mathcal{PC}(\mathcal{K}')$. Let $K' = kb(\mathcal{K}')$, and let $\mathcal{M}$ be the Herbrand model for K' (see section 5.5.1). Clearly $\mathcal{M} \models F'$, thus need only show $\mathcal{M} \models Nr'$ and $\mathcal{M} \models Ar'$. But this follows easily from lemmas 27 and 30. ∎

Finally,

Theorem 8 (Socratic Completeness for ALL(+neg)) *If $\mathcal{K}$ is a well-formed kbs of height one and f is a fact allowed in $\mathcal{K}$ then:*

$$\mathcal{PC}(\mathcal{K}) \models \mathcal{PC}(f) \Rightarrow \mathcal{K} \vdash_{ALL} f.$$

Proof: $\mathcal{PC}(\mathcal{K}) \models \mathcal{PC}(f)$ implies that $\mathcal{K} + negate(f)$ is inconsistent. Thus, by lemma 32, $\mathcal{K} + negate(f)$ is p-inconsistent. So, by lemma 28, $\mathcal{K} \vdash_{ALL} f$. ∎

Chapter 7

Time Complexity of ALL

In this chapter we show that a primitive ALL operation can be computed in time polynomial in the size of the portion of the knowledge-base accessible to it. We focus on primitive operations since non-primitive operations are defined in terms of (short) sequences of primitive operations (equation 5.10). Similarly, we focus on primitive operations in ALL without negation since ALL(+neg) operations are defined as (short) sequences of ALL operations (see section 6.2.2).

To discuss the complexity of ALL it is useful to return to the view of an ALL knowledge-base as a collection of frames and slots. This view was introduced in section 1.2 and discussed further in section 5.3.1. The basic idea is that constants are thought of as *frames*, and relations are thought of as *slots*. The fact $r(c_1, c_2)$ is equivalent to putting c_2 in the r slot of the frame c_1. Recall that if c is a frame and r is a slot then we refer to the pair $\langle c, r\rangle$ as a *frame-slot*. Recall also that partitions are simply collections of frame-slots (and the rules for a partition are the rules which could possibly be triggered by queries to, and assertions into, the frame-slots in the partition).

To examine the complexity of ALL operations we define some (fairly complex) functions which return the portion of the knowledge-base an operation can access, and the portion an operation can affect. In most cases these portions are much smaller than the entire knowledge-base. Recall that an ALL operation, $\mathcal{O}$, is defined as the infinite union over n of $\mathcal{O}_n$. Thus it is not obvious that $\mathcal{O}$ can be efficiently computed. However, we show that there is always an n such that $\mathcal{O} = \mathcal{O}_n$, and that this n is polynomial in the portion of the knowledge-base accessible to $\mathcal{O}$. Finally, we show that $\mathcal{O}_n$ can also be computed in polynomial time.

Computational Model

In order to discuss the computational complexity of ALL it is necessary to make some assumptions about the complexity of basic operations. We assume that a value can be added to a frame-slot in constant time (*i.e.*, in time independent of the size of the knowledge-base, the frame, or the number of values in the frame-slot), while the values in a frame-slot can be accessed in time proportional to the number of values in the frame-slot. In some cases we need to union knowledge-bases. In all such cases the knowledge-bases to be unioned are of the form:

$$K + S, K + S'$$

(where K is a knowledge-base and S and S' are sets of facts and if-added rules allowed in K) We assume that such unions can be computed in time proportional to $max(\mid S \mid, \mid S' \mid)$[1] (*i.e.*, independent of the size of K).

Defining Accessibility

The definition of the accessible portion of a knowledge-base is fairly technical and is given in appendix, section B.4. In this section we simply give the intuitions behind the definitions and the key lemmas which are used in the proofs which follow.

For a knowledge-base K, and $p \subset C \times R$ we define $rules(p) = Ar \cup Nr\backslash_p$ ($\backslash_p$ is defined on page 128). Now, consider a primitive operation $\mathcal{O}$ allowed in a knowledge-base K, and assume $\mathcal{O} = query(q)$ or $\mathcal{O} = assert(q)$. We define $reach_n(q,p)$ to include all frame-slots which can be accessed in the calculation of $\mathcal{O}_n$ in p, with rule backchaining cut off at depth n, and $change_n(q,p)$ to include all frame-slots which can be changed in the calculation of $\mathcal{O}_n$ in p (with rule backchaining cut off at depth n). Finally, we define $frames_n(q,p)$ to include all *frames* which $\mathcal{O}_n$ can access (with rule backchaining cut off at depth n). $frames_n(q,p)$ includes the frames appearing in frame-slots in $reach_n(q,p)$, plus the frames appearing explicitly in rules in $rules(p)$, and the frames in q itself.

We define $ops_n(\mathcal{O})$ to include all operations that $\mathcal{O}_n$ potentially 'depends on'. These include all queries of frame-slots in $reach_n(q, par_K(\mathcal{O}))$ and all assertions (of frames in $frames_n(q, par_K(\mathcal{O})))$ into frame-slots in $change_n(q, par_K(\mathcal{O}))$ (some amount of care is required to prove that ops includes all the assertions and queries which $\mathcal{O}_n$ depends on — for details, and a formal definition of 'the set of all operations which $\mathcal{O}$ depends on', see the appendix, section B.4).

We then define:

$$\begin{aligned} reach(q,p) &= (\bigcup n : n > 0 : reach_n(q,p)) && (7.1)\\ change(q,p) &= (\bigcup n : n > 0 : change_n(q,p)) && (7.2)\\ frames(q,p) &= (\bigcup n : n > 0 : frames_n(q,p)) && (7.3)\\ ops(\mathcal{O}) &= (\bigcup n : n > 0 : ops_n(\mathcal{O})). && (7.4) \end{aligned}$$

Figures 7.1 and 7.2 show $reach$, $change$, and ops for two simple queries.

If $\mathcal{O} = query(q)$ or $\mathcal{O} = assert(q)$ is a primitive operation allowed in a knowledge-base K then let:

$$\begin{aligned} nops(\mathcal{O}) &= \mid ops(\mathcal{O}) \mid && (7.5)\\ nfr(\mathcal{O}) &= \mid frames(q, par_K(\mathcal{O})) \mid && (7.6)\\ nrules(\mathcal{O}) &= \mid rules(par_K(\mathcal{O})) \mid && (7.7) \end{aligned}$$

Further, let ma be the maximum arity of any relation in R, $mvars(\mathcal{O})$ be the maximum number of variables in any rule in $rules(par_k(\mathcal{O}))$, and len be the maximum length of the antecedent of any rule in $rules(par_K(\mathcal{O}))$.

$\mathcal{O}$ is defined as an infinite union of $\mathcal{O}_n$'s thus it is not obvious how it can be computed. The key lemma for computing $\mathcal{O}$ is that if there is an n at which all the operations which $\mathcal{O}$ 'depends

[1] Where $\mid S \mid$ is the cardinality of the set S.

Recall the knowledge-base, K, from figure 5.3:

$$\begin{aligned} C &= \{c\} \\ R &= \{r_1, r_2\} \\ Nr &= \{r_1(c,x) \leftarrow r_2(c,x)\} \\ Ar &= \{\} \\ F &= \{r_2(c,c)\} \\ P &= \{\{\langle c, r_1\rangle, \langle c, r_2\rangle\}\} \end{aligned}$$

Consider $\mathcal{O} = query(r_1(c,x))$ (where x is a variable). Let $q = r_1(c,x)$ and $p = par_K(\mathcal{O})$, then:

$$\begin{aligned} reach_0(q,p) &= \{\langle c, r_1\rangle\} \\ reach_1(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\ reach_2(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\ reach_n(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\ reach(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \end{aligned}$$

Thus:

$$\begin{aligned} change(q,p) &= \{\langle c, r_1\rangle\} \\ ops(\mathcal{O}) &= \{query(r_1(c,x)), query(r_2(c,x)), query(r_1(c,c)), \\ &\quad\; query(r_2(c,c)), assert(r_1(c,c))\} \end{aligned}$$

Figure 7.1: The accessible frame-slots and dependent operations for a simple query.

Recall the knowledge-base, K, from figure 5.5:

$$\begin{aligned}
C &= \{c\} \\
R &= \{r_1, r_2, r_3\} \\
Nr &= \{r_1(c,x) \leftarrow r_2(c,x), r_2(c,x) \leftarrow r_3(c,x)\} \\
Ar &= \{\} \\
F &= \{r_3(c,c)\} \\
P &= \{\{\langle c, r_1\rangle\}, \\
&\quad \{\langle c, r_2\rangle, \langle c, r_3\rangle\}\}
\end{aligned}$$

Consider $\mathcal{O} = query(r_1(c,x))$ (where x is a variable). Let $q = r_1(c,x)$ and $p = par_K(\mathcal{O})$, then:

$$\begin{aligned}
reach_0(q,p) &= \{\langle c, r_1\rangle\} \\
reach_1(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\
reach_2(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\
reach_n(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\} \\
reach(q,p) &= \{\langle c, r_1\rangle, \langle c, r_2\rangle\}
\end{aligned}$$

Thus:

$$\begin{aligned}
change(q,p) &= \{\langle c, r_1\rangle\} \\
ops(\mathcal{O}) &= \{query(r_1(c,x)), query(r_2(c,x)), query(r_1(c,c)), \\
&\quad query(r_2(c,c)), assert(r_1(c,c))\}
\end{aligned}$$

Figure 7.2: The accessible frame-slots and dependent operations for a query in a knowledge-base with two partitions.

on' return the same value they returned at $n-1$, then $\mathcal{O}(K) = \mathcal{O}_n(K, par_K(\mathcal{O}))$. The proof of this lemma is given in the appendix (section B.4).

Lemma 33 (Computation Lemma) *Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $n > 0$:*

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O}))) \tag{7.8}$$
$$\Longrightarrow$$
$$\mathcal{O}(K) = \mathcal{O}_n(K, par_K(\mathcal{O})). \tag{7.9}$$

Complexity of Closure

In order to derive the complexity of ALL operations we first characterize the complexity of the *closure* operation. We first bound the facts and rules which closing the knowledge-base can add, and then bound the time to compute the closure of a knowledge-base.

Intuitively, *closure* can only add rules which are 'shortened' versions of if-added rules in the knowledge-base (instantiated with frames in $frames(q, par_K(\mathcal{O}))$). One can also show that for any fact f added by *closure*, $assert(f)$ is in $ops(\mathcal{O})$. In the appendix (section B.4), we define $closure_{f\&r}(\mathcal{O})$ — the set of facts and rules which can be added by *closure*. The key lemma for $closure_{f\&r}(\mathcal{O})$ is that it contains any fact or rule added by an ALL operation. The proof of this lemma is given in the appendix (section B.4).

Lemma 34 ($closure_{f\&r}$ lemma) *Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $\mathcal{O}' \in ops(\mathcal{O})$:*

1. *For any rule ρ, if $\rho \in \mathcal{O}'(K)$, but $\rho \notin K$, then $\rho \in closure_{f\&r}(\mathcal{O})$.*
2. *For any fact f, if $f \in \mathcal{O}'(K)$, but $f \notin K$, then $f \in closure_{f\&r}(\mathcal{O})$.*

From this lemma we can calculate a bound on the number of potential additions to a knowledge-base, and a bound on the time to calculate the *closure* of a knowledge-base.

Lemma 35 *Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K.*

1. *$\mid closure_{f\&r}(\mathcal{O}) \mid$ is of order: $len \times nrules(\mathcal{O}) \times nfr(\mathcal{O})^{mvars(\mathcal{O})}$*
2. *For any $S \subset closure_{f\&r}(\mathcal{O})$, $closure(K + S)$ can be computed in time:*

$$len^2 \times nrules(\mathcal{O})^2 \times nfr(\mathcal{O})^{2mvars(\mathcal{O})+ma}.$$

Proof: The intuition for the first part of the lemma, is that $closure_{f\&r}(\mathcal{O})$ is bounded by the set of all shortened versions of rules in $rules(par_K(\mathcal{O}))$, instantiated with frames in $frames(q, par_K(\mathcal{O}))$. The intuition for the second part is that (by lemma 34) $closure_n(K+S)$ need only be computed for values of n less than $\mid closure_{f\&r}(\mathcal{O}) \mid$. Further, for any rule ρ, $closure(K+S, \rho)$ can be computed in time bounded by $nfr(\mathcal{O})^{ma}$ (the potential number of facts matching its key), and for any n, computing $closure_n$ may require closing with respect to no more than $\mid closure_{f\&r}(\mathcal{O}) \mid$ rules. Full definitions (of $closure_{f\&r}$ and the other functions) are given in the appendix. The full proof of this lemma can be worked out from these definitions. ▮

Complexity Theorem

We now derive the n at which all operations in $ops(\mathcal{O})$ 'stop growing'.

Lemma 36 *For any operation $\mathcal{O}$ allowed in a closed knowledge-base K if*

$$n = nops(\mathcal{O}) \times len \times nrules(\mathcal{O}) \times nfr(\mathcal{O})^{mvars(\mathcal{O})} \tag{7.10}$$

then

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_{n+1}(K, par_K(\mathcal{O})) = \mathcal{O}'_n(K, par_K(\mathcal{O}))). \tag{7.11}$$

Proof: Consider the vector of all operations in $ops(\mathcal{O})$. As we increase n the knowledge-bases returned by these operations may not shrink (by lemma 8 part 3). Further, if we ever reach a point where none of them grow then we can quit (by lemma 33). Notice that any ALL operation can only put a known frame in a known slot or complete an if-added rule (there are no operations, for example, which create an entirely new frame). Further, by lemma 34, only facts and rules in $closure_{f\&r}(\mathcal{O})$ can be added. Thus after at most $\mid closure_{f\&r}(\mathcal{O}) \mid$ facts and rules are added, a single knowledge-base must be 'full'. Finally, note that each iteration may increase at worst one knowledge-base in the set of knowledge-bases returned by the operations in $ops(\mathcal{O})$. Thus if n is greater than or equal to

$$nops(\mathcal{O}) \times len \times nrules(\mathcal{O}) \times nfr(\mathcal{O})^{mvars(\mathcal{O})}$$

then all knowledge-bases must be full and we must have:

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_{n+1}(K, par_K(\mathcal{O})) = \mathcal{O}'_n(K, par_K(\mathcal{O}))). \tag{7.12}$$

▮

Finally, we have:

Theorem 9 (Complexity Theorem) *Let nops, len, ma, nfr, nrules, and mvars be as defined above (equations 7.5 - 7.7). For any primitive operation $\mathcal{O}$ allowed in a closed knowledge-base K, $\mathcal{O}(K)$ can be computed in time of order:*

$$len^5 \times nops(\mathcal{O})^2 \times nrules(\mathcal{O})^5 \times nfr(\mathcal{O})^{5mvars(\mathcal{O})+ma}.$$

Proof: Assume that $\mathcal{O} = query(q)$ or $\mathcal{O} = assert(q)$.
Consider again the vector of all operations in $ops(\mathcal{O})$. There are

$$nops(\mathcal{O})$$

such operations. By the previous lemma we must calculate at most the result of these operations for values of n of

$$nops(\mathcal{O}) \times len \times nrules(\mathcal{O}) \times nfr(\mathcal{O})^{mvars(\mathcal{O})}$$

or less. Thus it only remains to find the time to calculate the result of a primitive operation $\mathcal{O}_n$ given the results of all operations $\mathcal{O}'_{n-1}$. We may have to apply at most

$$nrules(\mathcal{O})$$

rules. Each rule may branch on all values in $frames(q, par_K(\mathcal{O}))$ for all variables. Thus we may have

$$nfr(\mathcal{O})^{mvars(\mathcal{O})}$$

branches (the results of which must be unioned together). Finding the result of each branch involves taking at most len unions and closures[2]. There are thus order:

$$len$$

unions and closures per branch. Each union is done on a knowledge-base of form $K + S$ where S is of size $\mid closure_{f\&r}(\mathcal{O}) \mid$ or smaller (by lemma 34). Thus each union can be done in time of order:

$$len \times nrules(\mathcal{O}) \times nfr(\mathcal{O})^{mvars(\mathcal{O})}$$

Finally, by lemma 35, each closure can be computed in time:

$$len^2 \times nrules(\mathcal{O})^2 \times nfr(\mathcal{O})^{2mvars(\mathcal{O})+ma}$$

Multiplying these bounds together gives the time bound in the theorem. ∎

[2]And performing at most len retrievals from frame-slots containing at most $nfr(\mathcal{O})^{ma(K)}$ entries. However, this retrieval term is not significant because of the larger term generated by the unions.

Chapter 8

Conclusion

In this final chapter we discuss the relationship of Algernon and ALL to other work in knowledge representation, the 'meaning' of the formal results presented in the last three chapters, future work, and the primary contributions of ALL to the study of knowledge representation.

8.1 Related Work

8.1.1 Knowledge Representation Systems

Many systems for knowledge representation have been proposed. In this section we briefly discuss the relationship between Algernon and some these systems.

Algernon differs from most semantic networks based systems in that it uses a single general purpose retrieval/reasoning mechanism which is guided by the structure of the network. Past work has generally used the structure of the network only for special purpose reasoning (spreading activation, classification etc.), and has relied on a first-order logic theorem prover [Brachman et al., 83, Schubert et al., 83] or a weaker deduction system [Levesque, 84, Patel-Schneider, 85, Vilain, 85] for general reasoning.

A notable exception to this generalization is the recent work of deHaan and Schubert [Schubert, 79, deHaan & Schubert, 86]. Algernon and the networks of Schubert share several features including the use of access limitations to guide reasoning. The most obvious way to use the structure of a semantic network to limit access would be to perform deduction with facts not more than a few (say maybe two) nodes away in the network. The problem with this strategy is that some nodes (*e.g.*, the node for your spouse) may have a large number of links, many of which are irrelevant to the problem at hand. The solution used in ECOSYSTEM is to maintain a taxonomy of knowledge and use this taxonomy to guide reasoning [deHaan & Schubert, 86]. The difference in Algernon is that access is limited to known *access paths*, which access facts many nodes away in the network, but do so in a controlled fashion. Thus in Algernon it is the structure of the knowledge itself (or more specifically the structure of the access paths in the rules) which controls access and reasoning.

Another relevant line of research is the work on *vivid* knowledge-bases [Etherington et al., 89]. A vivid knowledge-base "...trades accuracy for speed ..." [Etherington et al., 89] by constructing a complete database of ground facts, from facts that may be presented in a more expressive language.

This approach has some of the same goals as Algernon — particularly in the area of efficiency — but takes a much different approach and makes different trade-offs. At a very high level, the principal differences are:

- Algernon represents all the knowledge that has been asserted (though not all of it may be accessible at a given time) while a vivid knowledge-base is an approximation of the asserted knowledge (thus weakened completeness results such as Socratic Completeness do not hold for vivid reasoning).

- To obtain increased efficiency, Algernon trades completeness for speed while a vivid approach trades both soundness and completeness for speed.

Two of the most widely used systems for knowledge representation are Prolog and KEE. Superficially Algernon resembles Prolog (and to a lesser extent KEE), but there are several important differences. Algernon differs from Prolog in (at least) the following ways:

- Algernon and Prolog use different inference algorithms. When a query is made in Algernon, all applicable rules are applied, branching on all known values for the variables in the rules. In Prolog, goals are satisfied using purely depth-first search with backtracking, and execution stops when a single set of bindings for the variables in the goal are found. This difference is purely a function of the history of Algernon; I see no reason why Algernon could not be re-implemented using Prolog style backtracking.

- Algernon imposes the syntactic restriction that rules must define access paths and queries must reference a known slot of a known frame, while Prolog has no such restrictions. This difference is mitigated by two factors: First, the access path restriction does not decrease the expressive power of Algernon (see discussion on page 9). Second, a good, efficiency minded, Prolog programmer will often simulate Algernon by writing rules which do define access paths. One advantage of the less restricted syntax of Prolog is that one can place variables in any position in queries (*e.g.*, if appropriate rules were in the knowledge-base one could query $add(X, Y, 10)$ and Prolog would return pairs of bindings for X and Y such that $X + Y = 10$). The principle advantage of the access path restrictions in Algernon is that the (worst case) time complexity of inference in Algernon is *guaranteed* to be a function of the size of the accessible portion of the knowledge-base. In Prolog the time complexity of inference is in general a function of the size of the entire knowledge-base.

- Algernon supports both forward and backward chaining, while Prolog directly supports only backward chaining.

- An Algernon rule is associated with a set and can only be applied to a query about a frame in that set. Prolog rules are global and any rule can be applied to any query. Associating rules with sets modularizes the knowledge-base and helps in the control of inference (see section 4.1.1).

- Algernon combines rules with a frame-based representation for knowledge, while a Prolog knowledge-base is unstructured (compiled Prologs gain a large part of their efficiency by structuring the Prolog knowledge-base according to a syntactic analysis of its facts and rules [Warren, 83]). Two advantages are gained by the use of a frame-based representation. First, frames have been found to be a useful way to organize knowledge (especially in large knowledge-bases). Second, during inference Algernon uses the frame structure to store (and efficiently retrieve) markers on predicates that have already been queried. The marking strategy used in Algernon allows Algernon to reason with 'recursive' rules which would cause an infinite loop in Prolog (*e.g.*, rules of form $p \leftarrow p$), and is responsible for the polynomial time complexity of Algernon (whereas the time complexity of Prolog, even without skolem functions, is exponential).

- Algernon implements 'classical' negation (*i.e.*, negation as in predicate calculus), while Prolog implements negation as failure. In Algernon, a negated fact is implied by the knowledge-base iff it is true in all models of the knowledge-base. In Prolog, the negation of a goal succeeds iff the goal fails (thus negation in Prolog is equivalent to the special form `:unp` in Algernon). Algernon's implementation of negation allows one to express disjunctions (*e.g.*, "John is the banker or the lawyer") which cannot be expressed directly in Prolog (see section 3.6).

- Existential quantification is represented in Prolog using skolem functions. Skolem functions do not fit well into a frame based representation (the value in a slot of a frame should be another frame, not something of form $f1(frame21)$). Existential quantification is currently expressed in Algernon using the special form `:forc` (see page 18), but we expect it to eventually be represented using arbitrary objects (see sections 3.7 and 8.3).

Algernon is actually more similar to KEE than to Prolog. Both are frame based and both provide forward and backward chaining rules. Algernon and KEE differ in their methods of integrating frames and rules. In KEE, a rule is a *unit* (the KEE equivalent of a frame). In Algernon, rules are associated with sets and must define paths through the network of frames. Both approaches have their advantages. By treating rules as frames, KEE is able to use its machinery for reasoning about frames to reason about rules. Associating rules with sets increases the modularity of Algernon knowledge-bases. Further, requiring that rules define access paths allows Algernon to guarantee that the time complexity of operations is independent of the size of the knowledge-base. The principle difference between Algernon and KEE, however, is that Algernon is a relatively compact system which is inherits several important formal guarantees (*e.g.*, Socratic and Partitional Completeness, and polynomial time complexity[1]) from ALL, while KEE is a conglomeration of useful techniques and tools which provides no formal guarantees.

8.1.2 Related Formalisms

ALL draws from several diverse fields. We attempt only to sketch in general terms the fields from which it draws and discuss a few particularly relevant past approaches.

ALL draws from semantic networks [Quillian, 67, Bobrow & Winograd, 77, Findler, 79, Brachman et al., 83, Vilain, 85, Shapiro, 89] the intuition that retrieval and reasoning can be guided and limited by the structure of the network. This has long been a key intuition behind semantic networks: "...the knowledge required to perform an intellectual task generally lies in the semantic vicinity of the concepts involved in the task." [Schubert, 79]. In particular, ALL draws from semantic

[1]Except where the expressive power of Algernon exceeds that of ALL.

networks its frame based data structures [Minsky, 75] and the idea of access paths. Our use of access paths is closely related to previous work on path based inference. Path based inference can be traced back (at least) to [Raphael, 68] and later to [Schwarcz et al., 70] and [Shapiro & Woodmansee, 69]. A good discussion of path based and node based inference (both of which are partially subsumed by inference in ALL and would be totally subsumed if ALL supported full quantification — see section 8.3) is given in [Shapiro, 78].

One difference between ALL and much recent careful work on knowledge representation is that ALL (along with first-order logic and the original work on semantic networks) allows the knowledge-base designer to name the relations used in the knowledge-base. After Woods' influential "What's in a Link" paper [Woods, 75], many knowledge representation languages restricted the allowable relations to a small set which were given a precise syntax and semantics [Brachman, 79, Shapiro, 89]. Our approach in ALL is to define our semantics by "borrowing" the model theory of predicate calculus (by mapping ALL knowledge-bases to statements in predicate calculus and proving soundness and weakened completeness results) and to allow relations to be given any names. The *meanings* of the relations are thus restricted only by the contents of the knowledge-base (as in predicate calculus).

The design of the inference mechanism in ALL has been heavily influenced by logic programming. The notation and results from the proof of the completeness of logic programming [Lloyd, 84, Apt, 88] have been used extensively in the completeness proofs for ALL. In a sense our use of access-paths is a strategy for ordering conjunctive queries and as such is related to the more elaborate approach given in [Smith & Genesereth, 85]. In fact, if one follows a discipline of avoiding frame-slots containing a large number of frames[2] then the use of access-paths enforces an ordering on conjunctive queries much like that discussed in [Smith & Genesereth, 85].

Recently there has been growing interest in identifying tractable inference rules — inference rules under which implication is decidable in polynomial time. One interesting result to come out of this work is that the power of tractable inference rules is dependent on the syntax of the language used [McAllester et al., 89] (see also [McAllester, 90] p. 1115). Along these line, [Shastri & Ajjanagadde, 90] shows that, using a highly parallel implementation, certain queries can be answered in time proportional to the length of the shortest derivation of the query. Term subsumption languages (the current state of which are overviewed in [Patel-Schneider et al., 90]) also support a kind of tractable inference (the computation of subsumption relations between descriptions). ALL provides an example of a tractable inference method which can be applied in a network structured knowledge-base, and which is powerful enough to guarantee Socratic Completeness.

8.2 Discussion of Results

Given a knowledge representation system with a model theory, and given a knowledge-base, one may divide the set of all possible queries in several ways. For example, one can distinguish the queries which succeed from those which fail. If this set is exactly equal to the set of all queries which are model theoretic consequences of the knowledge-base then the knowledge representation

[2] *E.g.*, if the set *things* is very large, then one would like to avoid filling the slot *member* with all the members of *things* — rather, the preferred representation in ALL for "f is a *thing*" would be to put *things* in the *isa* slot of f.

system is consistent and complete. In ALL we divide the set of all queries into three sets:

- Those which succeed immediately.
- Those which will succeed after some appropriate series of preliminary queries.
- Those which will never succeed (without additional assertions).

Socratic Completeness then gives a precise characterization of the second and third sets — the second set is equal to the set of all queries which are model theoretic consequences of the knowledge-base, and the third set is equal to the set of all queries which are not model theoretic consequences. The first set consists of the 'obvious' consequences of the knowledge-base. We have shown that this set contains, at least, any queries deducible by the application of forward chaining rules (lemma 7), and any query following by the application of backward chaining rules in the partition of the query (*i.e.*, Partitional Completeness). *Modus tolens*, which has been implemented but not yet formalized, allows additional 'obvious' deductions to be made from disjunctions (see section 3.6).

8.2.1 About Socratic Completeness

One of the questions asked about our work[3] was "What good is Socratic Completeness when Socrates is dead?" Meaning that the hard part of reasoning has simply been pushed off to the problem of posing the right questions. The first answer to this question is that in a system with the expressive power of first-order logic, the incomputability never goes away; our approach decomposes the problem of reasoning into two parts: the (tractable) problem of computing the results of queries and the (intractable) problem of deciding what queries to ask. Past work has made other divisions — *e.g.*, the T-box and A-box of [Brachman et al., 83]. The second answer is that our goal is to develop a knowledge representation system with *understandable* inferential power. To this end, Socratic Completeness is a necessary (but not yet sufficient) property. Socratic Completeness guarantees that there is some hope of eventually finding the right questions (by guaranteeing that the questions exist).

These answers suggest two directions for future work: first, encoding (in the knowledge-base) common-sense knowledge about what general types of queries are useful for solving common types of complex reasoning problems; and second, the identification of other weakened completeness properties which, like Partitional Completeness, define what queries immediately succeed.

Socratic Completeness is also a step toward a formal specification of what inferential power a knowledge representation system should provide. Due to its intractability, full logical completeness is too strong a specification, but provides an upper bound in the search for an appropriate specification. Socratic Completeness is a fairly weak specification but arguably a necessary property. Thus it provides a lower bound on the space of appropriate specifications.

8.2.2 About Partitional Completeness

Partitioning the knowledge-base is not a new idea. Minsky's original frames paper ([Minsky, 75]) envisioned a structure on the knowledge-base consisting of groups of related frames. Hayes' Naive

[3] By Rich Thomason at the 1989 Workshop on Formal Aspects of Semantic Networks.

Physics Manifesto ([Hayes, 85]) also viewed commonsense knowledge as consisting of clusters of closely related concepts, loosely related to each other. Closer to the implementation level, blackboard architectures ([Hayes-Roth, 85]) also group inference methods into weakly interacting partitions. While these intuitions about the modularity of knowledge are persuasive, it must be admitted that it has not yet been empirically demonstrated that the contents of large-scale commonsense knowledge-bases divide naturally into partitions.

If we accept the intuition that knowledge can be meaningfully divided into partitions (or perhaps before we commit to accepting this intuition), we would like to know what effect partitions have on reasoning. Intuitively, one would expect that the rules in the partition of a query would somehow be more easily accessible to the query. The Partitional Completeness Theorem gives a partial formalization of this intuition, by saying that if a query is a logical consequence of the if-needed rules in its partitions then the query will succeed immediately. The theorem also gives us an empirical way to test a partitioning of a large knowledge-base — if simple queries depend on many rules in other partitions (and thus require many preliminary queries before they succeed) then the partitions are not well chosen.

8.2.3 About the Complexity Results

The expression given in theorem 9 for the complexity of inference in ALL involves too many variables to be easily comprehended. In a 'typical' knowledge-base one might expect that:

$$nrules(\mathcal{O}) \approx nfr(\mathcal{O}) \tag{8.1}$$

$$nops(\mathcal{O}) \approx nfr(\mathcal{O})^{ma} \tag{8.2}$$

Let $n = nfr(\mathcal{O})$. If we further assume that len is small, then ALL operations can be computed in time of order:

$$n^{3ma+5+5mvars(\mathcal{O})}$$

Certainly a tighter bound could be computed (with somewhat more work). In general we have found that our examples run much faster than the worst case bound. However, the complexity analysis is still a useful exercise. Our implementation of ALL originally used an algorithm which was exponential in the worst case. Replacing it with an algorithm similar to the one given here greatly improved its run time. Further, the complexity result gives some guidance in the design of knowledge-bases. For example, it suggests that while the length of rules makes little difference, rules with many variables should be avoided.

8.3 Future Work — Expressing Full First Order Quantification

While Algernon can be used to express full quantification, ALL cannot represent mixed existential and universal quantification. We expect to be able to formalize full quantification using the arbitrary objects of [Fine, 85]. The primary theoretical problem with mixed quantification, however, is that is it not clear what sorts of inference rules to allow. With mixed quantification it would be straightforward to implement a Turing Machine in ALL (since all cells would be linked along the tape, the access limitations provide no obvious help here). Clearly, if inference in ALL is to remain decidable, the inference rules must be weak enough that a single query cannot force ALL to 'run' the Turing Machine. On the other hand, if ALL is to remain Socratically Complete, there must exist a series of queries that in some way 'step' through the 'run' of the Turing Machine. One possible approach would be to carefully control the creation of new frames. Recent work in learning and knowledge integration suggests that the creation of new frames should be separated from inference by using one (tractable) component of a knowledge representation system to perform deductions with the existing frames while another (higher level) component decides (carefully) when new frames should be created [Murray & Porter 89, Murray 90].

8.4 Contributions

We believe that ALL makes four main contributions to the study of knowledge representation. First, ALL shows how two previously disparate tools for representing knowledge, frames and rules, can efficiently work together; rules define access paths through a network of frames. Second, ALL helps formalize the intuition that reasoning in a semantic network can be more efficient than reasoning using an unstructured representation. Third, ALL shows how 'classic' negation (*i.e.*, not negation by failure) can be added to a rule based system (and thus how disjunctions can be represented, and efficiently reasoned with, in a rule based system). Fourth, ALL introduces the notion of Socratic Completeness — a weakened form of completeness that is achievable without sacrificing expressive power or tractability.

Appendix A

Three Systems Built Using Algernon

A.1 The Headache Protocol

```
; An Expert System for Diagnosis of Headache.
; Copyright (c) 1990, Benjamin Kuipers.

; The medical content is taken from the Headache Protocol (11/75)
; by A. Komaroff, et al.

;   WARNING:  This expert system is presented for tutorial
;   purposes only.  Diagnosis of a medical condition should be
;   done only under the supervision of a physician.

; Enforcing the sequence on the questions.
;
;  1.  Block-to-block sequencing.

;  The blocks are regarded as subsets of Patients.  When the patient is
; asserted to be in a given block, the rules associated with that block
; are fired, as if-added rules triggered by (isa ?p Blockn).  When the
; block is complete, (isa ?p Blockn+1) is asserted.  Thus, the patient
; accumulates membership in an increasing set of blocks.  Membership is
; never retracted.

; This control structure works as long as the blocks are organized in a
; loop-free transition-net, rather than a general finite-state machine.  The
; strategy is consistent with the monotonic nature of Algernon, where
; information is only accumulated.

;  2.  Within-block sequencing.

; Within a block, the initial rule asks all the questions.  Sequence is
; controlled by the order in an access path.  Then if-added rules
; trigger on the specific answers that indicate various actions.

; Setup the KB.

(defun setup-HA-ES ()
```

```
  (assert-objects-and-slots)
  (assert-blocks)
  (assert-ask-slots)
  (rules-for-start-and-end)
  (rules-for-block-logic))

; And test it.

(defun test-HA-ES ()
  (a-assert "John has a headache."
          '((:the ?john (name ?john (:quote (John)))
                        (isa ?john Patients))
            (complains-of ?john Headache))))


; Here are the set and slot declarations.

(defun assert-objects-and-slots ()
  (a-assert "Taxonomy"
          '((:taxonomy (Objects
                        (Patients)
                        (Symptoms headache)
                        (speed Stat)
                        (Diagnoses acute-sinusitis
                                   chronic-sinusitis
                                   vascular-HA
                                   muscular-HA)
                        (BP-attributes systolic diastolic)))))

  (a-assert "Ask-slots"
          '((:taxonomy (Slots (Ask-slots)))
            (:srules ask-slots
             ((?p ?x true) <- (:ask (?p ?x true)))
             ;; This next rule makes sense ONLY
             ;; under depth-first search:
             ((?p ?x false) <- (:unp (:ask (?p ?x true))))
             ;;
             ((?p ?x false) -> (not (?p ?x true)))
             ((?p ?x true) -> (not (?p ?x false)))
             ((not (?p ?x true)) -> (?p ?x false))
             ((not (?p ?x false)) -> (?p ?x true)))))

  (a-assert "Internal slots"
          '((:slot complains-of (Patients Symptoms))
            (:slot refer-to-MD (patients speed))
            (:slot reds  (patients booleans) :cardinality 1)
            (:slot greys (patients booleans) :cardinality 1)
            (:slot blues (patients nil) :cardinality 1)
            (:slot count-blues (patients booleans) :cardinality 1)
            (:slot Dx    (patients diagnoses))
            (:slot Rx    (patients nil))
            (:slot refill  (Patients Booleans) :cardinality 1)))

  (a-assert "Non-boolean slots, requiring :ask rules"
```

```
          '((:slot BP (Patients BP-attributes nil))
            (:slot age (patients nil) :cardinality 1)
            ;      temperature is already in AKBASE
            (:rules Patients
             ((BP ?p ?attrib ?num) <- (:ask (BP ?p ?attrib ?num)))
             ((temperature ?p ?t)  <- (:ask (temperature ?p ?t)))
             ((age ?p ?num)        <- (:ask (age ?p ?num))))))

; The following slots take boolean values, and are askable.

(a-assert "Boolean slots, requiring :ask rules"
          '((:slot acceptable-relief  (Patients Booleans)
                  :cardinality 1)
            (:slot acutely-tender-sinuses  (Patients Booleans)
                  :cardinality 1)
            (:slot appears-in-great-pain  (Patients Booleans)
                  :cardinality 1)
            (:slot appears-very-sleepy  (Patients Booleans)
                  :cardinality 1)
            (:slot arm-drift  (Patients Booleans)
                  :cardinality 1)
            (:slot asymmetric-pupils  (Patients Booleans)
                  :cardinality 1)
            (:slot awakens-congested  (Patients Booleans)
                  :cardinality 1)
            (:slot chronic-nasal-postnasal-drip  (Patients Booleans)
                  :cardinality 1)
            (:slot cloudy-corneas  (Patients Booleans)
                  :cardinality 1)
            (:slot consult-MD-for-other-reasons  (Patients Booleans)
                  :cardinality 1)
            (:slot definite-Dx-last-two-years  (Patients Booleans)
                  :cardinality 1)
            (:slot difficulty-speaking  (Patients Booleans)
                  :cardinality 1)
            (:slot difficulty-walking  (Patients Booleans)
                   :cardinality 1)
            (:slot disoriented  (Patients Booleans)
                  :cardinality 1)
            (:slot fainting  (Patients Booleans)
                  :cardinality 1)
            (:slot flashes/spots-before-eyes-at-onset  (Patients Booleans)
                  :cardinality 1)
            (:slot HA-at-this-moment  (Patients Booleans)
                  :cardinality 1)
            (:slot HA-worse-after-blow  (Patients Booleans)
                  :cardinality 1)
            (:slot HAs-worse-since-starting-pills  (Patients Booleans)
                  :cardinality 1)
            (:slot head-injury-in-last-48-hours  (Patients Booleans)
                  :cardinality 1)
            (:slot hit-hard-in-head  (Patients Booleans)
                  :cardinality 1)
            (:slot numbness-of-hand/arm/leg  (Patients Booleans)
```

```
                   :cardinality 1)
            (:slot often-when-resting-after-hard-work  (Patients Booleans)
                   :cardinality 1)
            (:slot on-protocol-acceptable-regiment  (Patients Booleans)
                   :cardinality 1)
            (:slot one-eye-red/tearful  (Patients Booleans)
                   :cardinality 1)
            (:slot one-eyelid-drooping  (Patients Booleans)
                   :cardinality 1)
            (:slot pain-deep-behind-one-eye  (Patients Booleans)
                   :cardinality 1)
            (:slot pain-over-frontal/maxillary-sinuses  (Patients Booleans)
                   :cardinality 1)
            (:slot purulent-discharge  (Patients Booleans)
                   :cardinality 1)
            (:slot red/tearful-conjunctiva  (Patients Booleans)
                   :cardinality 1)
            (:slot redness/edema-over-sinuses  (Patients Booleans)
                   :cardinality 1)
            (:slot seeks-refill  (Patients Booleans)
                   :cardinality 1)
            (:slot severe-in-first-5-min  (Patients Booleans)
                   :cardinality 1)
            (:slot similar-HA-in-past  (Patients Booleans)
                   :cardinality 1)
            (:slot stiff-neck  (Patients Booleans)
                   :cardinality 1)
            (:slot taking-birth-control-pills  (Patients Booleans)
                   :cardinality 1)
            (:slot throbbing/pounding-pain  (Patients Booleans)
                   :cardinality 1)
            (:slot trouble-talking  (Patients Booleans)
                   :cardinality 1)
            (:slot trouble-walking  (Patients Booleans)
                   :cardinality 1)
            (:slot usually-begins-on-one-side  (Patients Booleans)
                   :cardinality 1)
            (:slot usually-w-nausea-or-diarrhea  (Patients Booleans)
                   :cardinality 1)
            (:slot vomiting  (Patients Booleans)
                   :cardinality 1)
            (:slot weakness-of-hand/arm/leg  (Patients Booleans)
                   :cardinality 1)
            (:slot worst-HA-ever  (Patients Booleans)
                   :cardinality 1))))


(defun assert-blocks ()
  (a-assert "Blocks are subsets of Patients"
           '((:taxonomy (Patients
                         (Block1)
                         (Block2)
                         (Block3)
                         (Block4)
```

```
                        (Block5)
                        (Block6)
                        (Block7)
                        ;(Block8) incorporated into Block7.
                        (Block9)
                        (Block10)
                        ;(Block11) incorporated into Block10.
                        (Block12)
                        (Block13)
                        (Block14)
                        (Block15)
                        (Block16)
                        (Block16a)
                        ; refer to MD:
                        (Refer)
                        (Done ))))))


(defun assert-ask-slots ()
  (a-assert "The :ask slots"
            '((isa (:slot acceptable-relief) ask-slots)
              (isa (:slot acutely-tender-sinuses) ask-slots)
              (isa (:slot appears-in-great-pain) ask-slots)
              (isa (:slot appears-very-sleepy) ask-slots)
              (isa (:slot arm-drift) ask-slots)
              (isa (:slot asymmetric-pupils) ask-slots)
              (isa (:slot awakens-congested) ask-slots)
              (isa (:slot chronic-nasal-postnasal-drip) ask-slots)
              (isa (:slot cloudy-corneas) ask-slots)
              (isa (:slot consult-MD-for-other-reasons) ask-slots)
              (isa (:slot definite-Dx-last-two-years) ask-slots)
              (isa (:slot difficulty-speaking) ask-slots)
              (isa (:slot difficulty-walking) ask-slots)
              (isa (:slot disoriented) ask-slots)
              (isa (:slot fainting) ask-slots)
              (isa (:slot flashes/spots-before-eyes-at-onset) ask-slots)
              (isa (:slot HA-at-this-moment) ask-slots)
              (isa (:slot HA-worse-after-blow) ask-slots)
              (isa (:slot HAs-worse-since-starting-pills) ask-slots)
              (isa (:slot head-injury-in-last-48-hours) ask-slots)
              (isa (:slot hit-hard-in-head) ask-slots)
              (isa (:slot numbness-of-hand/arm/leg) ask-slots)
              (isa (:slot often-when-resting-after-hard-work) ask-slots)
              (isa (:slot on-protocol-acceptable-regiment) ask-slots)
              (isa (:slot one-eye-red/tearful) ask-slots)
              (isa (:slot one-eyelid-drooping) ask-slots)
              (isa (:slot pain-deep-behind-one-eye) ask-slots)
              (isa (:slot pain-over-frontal/maxillary-sinuses) ask-slots)
              (isa (:slot purulent-discharge) ask-slots)
              (isa (:slot red/tearful-conjunctiva) ask-slots)
              (isa (:slot redness/edema-over-sinuses) ask-slots)
              (isa (:slot seeks-refill) ask-slots)
              (isa (:slot severe-in-first-5-min) ask-slots)
              (isa (:slot similar-HA-in-past) ask-slots)
```

```
              (isa (:slot stiff-neck) ask-slots)
              (isa (:slot taking-birth-control-pills) ask-slots)
              (isa (:slot throbbing/pounding-pain) ask-slots)
              (isa (:slot trouble-talking) ask-slots)
              (isa (:slot trouble-walking) ask-slots)
              (isa (:slot usually-begins-on-one-side) ask-slots)
              (isa (:slot usually-w-nausea-or-diarrhea) ask-slots)
              (isa (:slot vomiting) ask-slots)
              (isa (:slot weakness-of-hand/arm/leg) ask-slots)
              (isa (:slot worst-HA-ever) ask-slots))))


; Start diagnosis by asserting that the patient is in Block1.

(defun rules-for-start-and-end ()
  (a-assert "Starting and ending rules"
            '(
              (:rules Patients
               ((complains-of ?p Headache) -> (isa ?p Block1))
               ; a place-holder:
               ((Rx ?p ?rx) <- (:bind ?rx 'prescription)))

              (:rules Refer
               ((isa ?p Refer)
                (name ?p ?n)
                ->
                (:lisp
                 (format t
                         "~%Refer ~a to a doctor for further evaluation."
                         '?n))
                (isa ?p Done)))

              (:rules Done
               ((isa ?p Done)
                (Dx ?p ?dx)
                (name ?dx ?n)
                ->
                (:lisp (format t "~%The patient has ~a." '?n))
                (:lisp (format t "~%The consultation is done.~%"))
                )))))

; The rules for all the various blocks.

(defun rules-for-block-logic ()
  (a-assert "Block logic"
            '(

; Block1

(:rules Block1

  ((isa ?p Block1)
   (disoriented ?p false)
   (appears-very-sleepy ?p false)
```

```
   (appears-in-great-pain ?p false)
   (trouble-talking ?p false)
   (trouble-walking ?p false)
   ->
   (isa ?p Block2))

  ((disoriented ?pt true) -> (refer-to-MD ?pt Stat))
  ((appears-very-sleepy ?pt true) -> (refer-to-MD ?pt Stat))
  ((appears-in-great-pain ?pt true) -> (refer-to-MD ?pt Stat))
  ((trouble-talking ?pt true) -> (refer-to-MD ?pt Stat))
  ((trouble-walking ?pt true) -> (refer-to-MD ?pt Stat))

  ((refer-to-MD ?p Stat) -> (:lisp (format t "~%Get patient to a doctor!  Stat!"))
                            (isa ?p Refer)))


; Block2

(:rules Block2

  ((isa ?p Block2)
   (head-injury-in-last-48-hours ?p False)
   (HA-at-this-moment ?p True)
   (worst-HA-ever ?p True)
   (severe-in-first-5-min ?p ?tv1)
   (numbness-of-hand/arm/leg ?p ?tv2)
   (weakness-of-hand/arm/leg ?p ?tv3)
   (difficulty-speaking ?p ?tv4)
   (difficulty-walking ?p ?tv5)
   (vomiting ?p ?tv6)
   (fainting ?p ?tv7)
   ->
   (isa ?p Block3))

  ((head-injury-in-last-48-hours ?p True) -> (isa ?p Refer))
  ((HA-at-this-moment ?p False)           -> (isa ?p Block3))
  ((worst-HA-ever ?p False)               -> (isa ?p Block3))

  ((severe-in-first-5-min ?p True)     -> (reds ?p True))
  ((numbness-of-hand/arm/leg ?p True)  -> (reds ?p True))
  ((weakness-of-hand/arm/leg ?p True)  -> (reds ?p True))
  ((difficulty-speaking ?p True)       -> (reds ?p True))
  ((difficulty-walking ?p True)        -> (reds ?p True))
  ((vomiting ?p True)                  -> (reds ?p True))
  ((fainting ?p True)                  -> (reds ?p True)))

; Block3

(:rules Block3

  ((isa ?p Block3) (hit-hard-in-head ?p ?tv) ->
   (isa ?p Block4))

  ((hit-hard-in-head ?p True)
```

```
   (HA-worse-after-blow ?p True) ->
   (reds ?p True)))

; Block4

(:rules Block4
  ((isa ?p Block4) (similar-HA-in-past ?p ?tv) ->
   (isa ?p Block5))

  ((similar-HA-in-past ?p False)
   (age ?p ?age)
   (:lisp (> ?age 50))
   ->
   (reds ?p True)))

; Block5

(:rules Block5

  ((isa ?p Block5) (refill ?p False) -> (isa ?p Block6))
  ((isa ?p Block5) (refill ?p True)  -> (isa ?p Done))

  ((refill ?p False) <- (:unp (refill ?p True)))

  ((refill ?p True)
   <-
   (definite-Dx-last-two-years ?p True)
   (on-protocol-acceptable-regiment ?p True)
   (acceptable-relief ?p True)
   (seeks-refill ?p True)))

; Block6

(:rules Block6

  ((isa ?p Block6)
   (flashes/spots-before-eyes-at-onset ?p ?tv1)
   (one-eye-red/tearful ?p ?tv2)
   (weakness-of-hand/arm/leg ?p ?tv3)
   (difficulty-speaking ?p ?tv4)
   ->
   (isa ?p Block7))

  ((flashes/spots-before-eyes-at-onset ?p True) -> (Dx ?p vascular-HA))
  ((one-eye-red/tearful ?p True)                -> (count-blues ?p True))
  ((weakness-of-hand/arm/leg ?p True)           -> (reds ?p True))
  ((difficulty-speaking ?p True)                -> (reds ?p True)))

; Block7 (+ Block8)

(:rules Block7

  ((isa ?p Block7)
   (pain-deep-behind-one-eye ?p ?tv1)
```

```
   (throbbing/pounding-pain ?p ?tv2)
   (often-when-resting-after-hard-work ?p ?tv3)
   (usually-w-nausea-or-diarrhea ?p ?tv4)
   (usually-begins-on-one-side ?p ?tv5)
   ->
   (isa ?p Block9))

  ((pain-deep-behind-one-eye ?p True)           -> (count-blues ?p True))
  ((throbbing/pounding-pain ?p True)            -> (count-blues ?p True))
  ((often-when-resting-after-hard-work ?p True) -> (count-blues ?p True))
  ((usually-w-nausea-or-diarrhea ?p True)       -> (count-blues ?p True))
  ((usually-begins-on-one-side ?p True)         -> (count-blues ?p True)))

; Block8  (incorporated into Block 7)

; Block9

(:rules Block9

  ((isa ?p Block9)
   (pain-over-frontal/maxillary-sinuses ?p True)
   (chronic-nasal-postnasal-drip ?p ?tv1)
   (awakens-congested ?p ?tv2)
   ->
   (isa ?p Block10))

  ((pain-over-frontal/maxillary-sinuses ?p False)
   ->
   (isa ?p Block10))

  ((chronic-nasal-postnasal-drip ?p True) -> (greys ?p True))
  ((awakens-congested ?p True)            -> (greys ?p True)))

; PHYSICAL EXAMINATION

; Block 10 (+ Block11).

(:rules Block10

  ((isa ?p Block10)
   (temperature ?p ?t)
   (BP ?p Systolic ?sys)
   (BP ?p Diastolic ?dias)
   (one-eyelid-drooping ?p ?tv1)
   (cloudy-corneas ?p ?tv2)
   (red/tearful-conjunctiva ?p ?tv3)
   (asymmetric-pupils ?p ?tv4)
   ->
   (isa ?p Block12))

  ((temperature ?p ?t)     (:lisp (> ?t 100.0))  -> (reds ?p True))
  ((BP ?p Systolic ?sys)   (:lisp (> ?sys 200))  -> (reds ?p True))
  ((BP ?p Diastolic ?dias) (:lisp (> ?dias 105)) -> (reds ?p True))
  ((one-eyelid-drooping ?p True)                 -> (reds ?p True))
```

```
  ((cloudy-corneas ?p True)                     -> (reds ?p True))
  ((red/tearful-conjunctiva ?p True)            -> (reds ?p True))
  ((asymmetric-pupils ?p True)                  -> (reds ?p True)))

; Block11 (incorporated into Block 10).

; Block12

(:rules Block12

  ((isa ?p Block12)
   (HA-at-this-moment ?p True)
   (acutely-tender-sinuses ?p ?tv1)
   (redness/edema-over-sinuses ?p ?tv2)
   (purulent-discharge ?p ?tv3)
   ->
   (isa ?p Block13))

  ((isa ?p Block12) (HA-at-this-moment ?p False) -> (isa ?p Block13))

  ((acutely-tender-sinuses ?p True)      -> (Dx ?p acute-sinusitis))
  ((redness/edema-over-sinuses ?p True) -> (Dx ?p acute-sinusitis))
  ((purulent-discharge ?p True)         -> (Dx ?p acute-sinusitis)))

; Block13

(:rules Block13

  ((isa ?p Block13)
   (stiff-neck ?p ?tv1)
   (arm-drift ?p ?tv2)
   ->
   (isa ?p Block14))

  ((stiff-neck ?p True) -> (reds ?p True))
  ((arm-drift ?p True)  -> (reds ?p True)))

; To make the last few blocks look like the others we add
; some slots so that we can trigger everything with forward-chaining:

(:slot any-reds (patients booleans) :cardinality 1)
(:slot any-greys (patients booleans) :cardinality 1)
(:slot multiple-blues (patients booleans) :cardinality 1)

(:rules Patients
  ((any-reds ?p true) <- (reds ?p true))
  ((any-greys ?p true) <- (greys ?p true))
  ((multiple-blues ?p true) <- (blues ?p ?nb) (:lisp (> ?nb 1)))

  ;; By the time we query these slots we can use negation by failure:
  ((any-reds ?p false) <- (:unp (any-reds ?p true)))
  ((any-greys ?p false) <- (:unp (any-greys ?p true)))
  ((multiple-blues ?p false) <- (:unp (multiple-blues ?p true))))
```

```
; Block14

(:rules Block14

  ((isa ?p Block14)
   (any-reds ?p false)
   (consult-MD-for-other-reasons ?p False)
   ->
   (isa ?p Block15))

  ((any-reds ?p True)
   ->
   (isa ?p Refer))

  ((consult-MD-for-other-reasons ?p True)
   ->
   (isa ?p Refer)))

(:rules Block15

  ((isa ?p Block15)
   (:unp (Dx ?p acute-sinusitis))
   (any-greys ?p ?v)
   (multiple-blues ?p ?v2)
   ->
   (isa ?p Block16))

  ((Dx ?p acute-sinusitis) (Rx ?p ?rx) -> (isa ?p Block16))

  ((any-greys ?p True) -> (Dx ?p chronic-sinusitis) (Rx ?p ?rx))

  ((multiple-blues ?p true) -> (Dx ?p vascular-HA)))

(:rules Block16

  ((isa ?p Block16)
   (Dx ?p vascular-HA)
   (Rx ?p ?rx)
   (taking-birth-control-pills ?p true)
   (HAs-worse-since-starting-pills ?p false)
   ->
   (isa ?p Done))

  ((isa ?p Block16)
   (:unp (Dx ?p vascular-HA))
   ->
   (Dx ?p muscular-HA)
   (Rx ?p ?rx)
   (isa ?p Done))

  ((taking-birth-control-pills ?p false)     -> (isa ?p Done))
  ((HAs-worse-since-starting-pills ?p true) -> (isa ?p Refer)))

)))
```

A.2 Belief Networks

```
;;; BELIEF.LISP
;;; Copyright (c) 1990 by Coy C. Day, Jr.
;;; Implementation of tree-structured subset of
;;; Judea Pearl's belief networks.

;;; To initialize:
;;; Nodes are structured, according to dependency, by parent
;;; and child relations.  Then, simply assert the conditional
;;; probability matrices between links, the initial pi value
;;; for the root node and initial lambda values for the leaf
;;; nodes.  The initial pi value for the root indicates the
;;; prior probabilities of the root values.  The initial
;;; lambda values for the leaves indicate that they are
;;; anticipatory nodes, i.e. they have no support yet from
;;; below.

;;; Architecture:
;;; Each variable is represented by an object called a "node".
;;; Its parent is given by (parent node ?p) and its children
;;; by (child node ?c).  Each node contains slots for its
;;; belief values and the messages it has received from other
;;; nodes.  These messages are "sent" by asserting the
;;; appropriate slot value in the receiving node.

;;; note:
;;; This code is written for understandability
;;; rather than efficiency.  Specific improvements to the
;;; basic algorithm are mentioned on p. 260 of Pearl.

;;; Reference:
;;; Pearl, Judea.  Fusion, propagation, and structuring in
;;; belief networks.  Artificial Intelligence 29 (1986) 241-288.


(defun bel-frames ()
  (a-assert "Taxonomy"
    '((:taxonomy
        (things (objects (contexts current-context)
                         (nodes))))))

  (a-assert "slots"
    '((:slot root (contexts nodes) :cardinality 1)
      (:slot leaf (contexts nodes))
      (:slot description (nodes nil) :cardinality 1)
      (:slot values (nodes nil) :cardinality 1)
      (:slot child (nodes nodes))
      (:slot parent (nodes nodes) :cardinality 1)
      (:slot dummy-child (nodes nodes) :cardinality 1)
      (:slot matrix (nodes nil) :cardinality 1)
      (:slot lambda (nodes nil) :cardinality 1)
      (:slot pi (nodes nil) :cardinality 1)
```

```
      (:slot belief (nodes nil) :cardinality 1)
      (:slot lambda-message (nodes nodes nil) :cardinality 1)
      (:slot pi-message (nodes nil) :cardinality 1)
      (:slot delete-lambda-msg (nodes nodes) :cardinality 1)
      (:slot delete-pi-msg (nodes nodes) :cardinality 1))))


(defun bel-rules ()

  (a-assert "general rules"
    '((:rules nodes

        ((child ?node1 ?node2) -> (parent ?node2 ?node1))

        ((delete-lambda-msg ?a ?b)
         ->
         (:unp (lambda-message ?a ?b ?msg))
         (:clear-slot ?a delete-lambda-msg))

        ((delete-lambda-msg ?a ?b)
         ->
         (lambda-message ?a ?b ?msg)
         (:clear-slot ?a delete-lambda-msg)
         (:delete (lambda-message ?a ?b ?msg)))

        ((delete-pi-msg ?b)
         ->
         (:unp (pi-message ?b ?msg))
         (:clear-slot ?b delete-pi-msg))

        ((delete-pi-msg ?b)
         ->
         (pi-message ?b ?msg)
         (:clear-slot ?b delete-pi-msg)
         (:delete (pi-message ?b ?msg))))))


  (a-assert "update rules"
    '((:rules nodes

        ;; update lambda

        ((lambda-message ?a ?b ?lambda-msg)
         ->
         (:unp (lambda ?a ?old-lambda))
         (:bind-to-values ?lambda-list ?a lambda-message)
         (:bind ?lambda (compute-lambda '?lambda-list))
         (lambda ?a ?lambda))

        ((lambda-message ?a ?b ?lambda-msg)
         ->
         (lambda ?a ?old-lambda)
         (:bind-to-values ?lambda-list ?a lambda-message)
         (:bind ?lambda (compute-lambda '?lambda-list))
```

```
 (:lisp (vector-/= '?lambda '?old-lambda))
 (:delete (lambda ?a ?old-lambda))
 (lambda ?a ?lambda))


;; update pi

((pi-message ?b ?pi-msg)
 ->
 (:unp (pi ?b ?old-pi))
 (matrix ?b ?mat)
 (:bind ?pi (compute-pi '?mat '?pi-msg))
 (pi ?b ?pi))

((pi-message ?b ?pi-msg)
 ->
 (pi ?b ?old-pi)
 (matrix ?b ?mat)
 (:bind ?pi (compute-pi '?mat '?pi-msg))
 (:lisp (vector-/= '?pi '?old-pi))
 (:delete (pi ?b ?old-pi))
 (pi ?b ?pi))


;; update lambda-message

((lambda ?b ?lambda)
 ->
  (parent ?b ?a)
 (matrix ?b ?mat)
 (:bind ?lambda-msg (compute-lambda-msg '?mat '?lambda))
 (delete-lambda-msg ?a ?b)
 (lambda-message ?a ?b ?lambda-msg))


;; update pi-message

((lambda ?b ?lambda)
 ->
 (pi ?b ?pi)
 (child ?b ?c)
 (:bind-to-values ?lambda-list ?b lambda-message)
 (:bind ?pi-msg (compute-pi-msg '?pi '?c '?lambda-list))
 (delete-pi-msg ?c)
 (pi-message ?c ?pi-msg))

((pi ?b ?pi)
 ->
 (child ?b ?c)
 (:bind-to-values ?lambda-list ?b lambda-message)
 (:bind ?pi-msg (compute-pi-msg '?pi '?c '?lambda-list))
 (delete-pi-msg ?c)
 (pi-message ?c ?pi-msg))
```

```
        ;; update belief

        ((lambda ?a ?lambda)
         ->
         (pi ?a ?pi)
         (:bind ?bel (compute-belief '?lambda '?pi))
         (:clear-slot ?a belief)
         (belief ?a ?bel))

        ((pi ?a ?pi)
         ->
         (lambda ?a ?lambda)
         (:bind ?bel (compute-belief '?lambda '?pi))
         (:clear-slot ?a belief)
         (belief ?a ?bel))))))


;;; Supporting LISP functions.

(defun compute-lambda (lambda-list)
;;; Lambda is the product of the received lambda messages.

  (reduce #'vector-* (mapcar #'cadr lambda-list)))


(defun compute-pi (matrix pi-vec)
;;; Multiply the link matrix by the received pi message.

  (matrix-trans-* matrix pi-vec))


(defun compute-belief (lambda pi-vec)
;;; Belief is alpha * lambda * pi.

  (let* ((prod (vector-* lambda pi-vec))
         (alpha (/ 1 (apply #'+ prod))))
    (scale prod alpha)))


(defun compute-lambda-msg (matrix lambda)
;;; Lambda message is the link matrix times lambda.

  (matrix-* matrix lambda))


(defun compute-pi-msg (pi-vec dest lambda-list)
  ;;; The pi message is pi times the product of all lambda messages
  ;;; except for that of the destination node.
  ;;; NEED TO CHANGE!

  (let* ((lambda-prod
          (mapcar #'(lambda (pair)
                      (if (eq (car pair) dest)
```

```
                           (mapcar #'(lambda (value) 1.0) pi-vec)
                           (cadr pair)))
                  lambda-list))
         (prod
          (vector-* pi-vec
                    (cond ((null lambda-prod)
                           '(1.0 1.0 1.0))
                          ((null (cdr lambda-prod))
                           (car lambda-prod))
                          (t
                           (reduce #'vector-* lambda-prod))))))

    (scale prod (/ 1 (apply #'+ prod)))))


(defun matrix-* (matrix vector)
  ;;; Multiply a matrix by a vector.

  (mapcar #'(lambda (vec)
              (sdot vec vector))
          matrix))


(defun matrix-trans-* (matrix vector)
  ;;; Multiply transpose of matrix by vector.

  (let ((result (mapcar #'(lambda (x) 0.0) (car matrix))))
    (mapc #'(lambda (vec x)
              (setf result (vector-+ result (scale vec x))))
          matrix vector)
    result))


(defun sdot (vec1 vec2)
  ;;; Dot product of two vectors.

  (apply #'+ (vector-* vec1 vec2)))


(defun scale (vector scalar)
  ;;; Multiply elements of one vector by a scalar.

  (mapcar #'(lambda (x)
              (* x scalar))
          vector))


(defun vector-* (a b)
  ;;; Multiply two vectors, item by item.

  (cond ((null a) b)
        ((null b) a)
        (t (mapcar #'* a b))))
```

```
(defun vector-+ (a b)
  ;;; Add two vectors a and b.

  (mapcar #'+ a b))


;;; Parameters for vector-/=
(defvar *threshold* 0.01
        "indication of how much the elements may differ")
(defvar *epsilon* 1e-10
        "smallest divisor allowed for normalizing elements")

(defun vector-/= (a b)
  ;;; Predicate returning nil iff the elements of vector a are
  ;;; are all "close to" the corresponding element of vector b.

  (or (not (listp a)) (not (listp b))
      (not (= (length a) (length b)))
      (progn (mapc #'(lambda (x y)
                       (let ((minimum (min x y)))
                         (when (> (abs (/ (- y x)
                                          (max minimum *epsilon*)))
                                  *threshold*)
                           (return-from vector-/= t))))
                   a b)
             nil)))
```

A.3 QPC

```
;;; Copyright: Adam Farquhar and James Crawford, 1990.
;;;
;;; The following functions are defined below:
;;;
;;;  (basic-qpt) - Basic qpt ontology.
;;;  (library) - Library of processes and views.
;;;  (portals) - Additional processes and views for portals.
;;;

(in-package 'cl-user)
(import 'qsim::t0)
(defparameter *t0* 't0 "The name for the initial time in all models.")

(defun basic-qpt ()

  (a-assert "Basic QPT Taxonomy"
           `((:taxonomy
               (objects
                 (scenarios)
                 (models)
                 (cds (views)
                      (processes))
```

```
               (constraints)
               (times ,*t0*)
               (variables-magnitudes
                 (magnitudes zero inf minf)
                 (variables))
               (derivatives inc dec std)
               (physobs)
               (influence-types (direct-influence-types i+ i-)
                                (indirect-influence-types q+ q-))
               (relations (2-relations q= S0+)
                          (3-relations add mult))
               (qpt-order-relations q-less q-greater q-equal)))
           (:taxonomy
             (sets (qspaces)))
           (:taxonomy
             (slots (considerations)
                    (quantities)))))
(a-assert "Scenarios."
          '((:slot current-scenario (contexts scenarios)
                  :comment "The current scenario of a context."
                  :cardinality 1)))
(a-assert "Models"
          '((:slot model (scenarios models)
                  :comment "A model of a scenario.")
           (:slot scenario (models scenarios)
                  :inverse model)
           (:slot initial-model (scenarios models)
                  :backlink     scenario
                  :cardinality  1)
           ;; Every scenario has an initial model:
           (:rules scenarios
            ((initial-model ?sc ?model) <-
             (:forc ?model (initial-model ?sc ?model))))

           (:slot prev-model (models models)
                  :comment "Links a model to a previous model of the
                            same scenario.")
           (:slot subseq-model (models models)
                  :inverse    prev-model
                  :comment "Links a model to a subsequent model of the
                            same scenario.")

           (:slot part-of (objects objects)
                  :comment "A generic relationship used
                            as an inverse for several different relations")
           ;; Transitively close part-of
           (isa (:slot part-of) tc-order-relations)

           ;; Copying to subseq model:
           (:slot copies-over (objects models models)
                  :comment "(copies-over x m1 m2) =
                            x can be copied from m1 to m2")
           (:rules models
            ((subseq-model ?m1 ?m2)
```

```
               (cd ?m1 ?cd)
               (copies-over ?cd ?m1 ?m2)
               ->
               (cd ?m2 ?cd))
              ((subseq-model ?m1 ?m2)
               (entity ?m1 ?po)
               (copies-over ?po ?m1 ?m2)
               ->
               (entity ?m2 ?po)))))

(a-assert "Times"
          `((:slot initial-time (models times)
                 :comment     "The time that a model starts to apply."
                 :cardinality 1
                 :backlink    part-of)
            (:rules models
             ((isa ?m models) -> (initial-time ?m ,*t0*)))))

(a-assert "Entity"
          '((:slot  entity (models objects)
                    :backlink part-of)))

(a-assert "Conditional Descriptions (CDs)"
          '((:slot cd (models cds)
                   :backlink part-of
                   :comment "A CD is a conditional description, which can~@
                   be either a view or a process.  The CD slot links~@
                   models to the cds which compose it."
                   )))

(a-assert "Considerations"
          '(
            ;; Considerations are relations that hold between entities
            ;; and views.  Any consideration is backlinked to "view-of":
            (:slot view-of (views objects))
            (:rules considerations
             ((isa ?con considerations) -> (backlink ?con (:slot view-of))))

            (:slot consider (models considerations objects)
                   :comment "Consider maps models to considerations
                             of physical objects.")
            ;;
            ;; When we lean about a consideration, then its entity is in
            ;; its model, as is the view.
            ;;
            ;; E.g., if:
            ;; (consider model1 physical-view object1) and
            ;; (physical-view object1 p-view1)
            ;; then we conclude:
            ;; (entity model1 object1) and
            ;; (cd model1 p-view1)
            ;;
            (:rules models
             ((consider ?m1 ?con1 ?po1)
```

```
              (?con1 ?po1 ?po1-con1)
              ->
              (entity ?m1 ?po1)
              (cd ?m1 ?po1-con1)))))

(a-assert "Magnitudes"
          '(;; This is a hack (since it should follow
            ;; from a more general rule):
            (not (less zero zero))))

(a-assert "Variables and qspaces."
          '((:slot variable (cds variables)
                   :backlink part-of)

            ;; variables-magnitudes = variables UNION magnitudes:
            (union-of variables-magnitudes variables)
            (union-of variables-magnitudes magnitudes)

            ;; note: Qspaces are sets of magnitudes:
            ;; (note that this is not the same as (imp-superset ?qsp ?mag) !)

            ;; If ?qsp1 is a qspace, then is a set of magnitudes.
            ;;
            (:rules qspaces
             ((isa ?qsp1 qspaces) -> (imp-superset ?qsp1 magnitudes)))

            (:slot qspace (models variables qspaces)
                   :comment "qspace links a variable in a model to its qspace.~@
                   its pseudo-inverse is qspace-inv."
                   :cardinality 1)

            ;; note, we cannot use :inverse since the arities of qspace
            ;; and qspace-inverse are different.
            ;;
            (:slot qspace-inv (qspaces variables)
                   :cardinality 1)

            (:rules models
             ((qspace ?m1 ?var1 ?qsp1)
              ->
              ;; Immediate put ?qsp1 in the main partition:
              ;;
              (frame-partition ?qsp1 main-partition)
              ;;
              ;; Backlink qspace to variables and models
              ;; (not simple :backlink since arities differ):
              (qspace-inv ?qsp1 ?var1) (part-of ?qsp1 ?m1)))

            ;; Forc a qspace in the model for each variable in its cd's:
            (:rules  models
             ((cd ?m1 ?cd1) (variable ?cd1 ?v1)
              ->
              (:forc ?qspace
               (qspace ?m1 ?v1 ?qspace))))))
```

```
(a-assert "Region transitions"
         '((:slot region-transition
                  (models variables magnitudes derivatives nil)
                  :comment "When variable is equal to magnitude
                            and moving in the specified direction
                            then a region transition occurs.")))

(a-assert "Tmags"
         '((:slot tmag (variables models times magnitudes)
                  :comment "TMAG allows us to assert that a variable~@
                              has a magnitude in a model at a time!")

           ;; If (tmag var model time mag), then that mag must be in
           ;; var.qspace.
           ;;
           (:rules variables
            ((tmag ?var1 ?m1 ?t1 ?mag1)
             (qspace ?m1 ?var1 ?qsp1)
             ->
             (member ?qsp1 ?mag1)))))

(a-assert "Tdirs"
         '((:slot tdir (variables models times derivatives)
                  :comment "TDIR allows us to assert that a variable~@
                              has a derivative in a model at a time!")
           (:rules variables
            ((tdir ?v ?m ?t std)
             ->
             (not (tdir ?v ?m ?t inc))
             (not (tdir ?v ?m ?t dec))))))

;; Inequalities are represented "twice". Once on the cds as:
;;       (cd-ineq cd26 less v1 v2)
;; and then are copied up to the models of which cd26 is a part as:
;;       (model-ineq v1 less v2 model-3)
;; The copy up to the model is necessary so we can transitively close
;; all the inequalities in the model.
;;
(a-assert "Cd inequalities"
         '((:slot cd-ineq (cds qpt-order-relations
                               variables-magnitudes variables-magnitudes)
                  :comment "The cd-ineq indexes an inequality under the CD"
                  )))

(a-assert "Model inequalities"
         '((:slot model-ineq
                  (variables-magnitudes qpt-order-relations
                   variables-magnitudes models)
                  :comment "(model-ineq ?v1 ?ineq ?v2 ?model) =
                           ?v1 ?ineq ?v2 in ?model.")

           (:slot q-inverse (qpt-order-relations qpt-order-relations)
                  :comment "(?ineq1 ?x ?y) and (q-inverse ?ineq1 ?ineq2) ->
```

```
                 (?ineq2 ?y ?x)")
(q-inverse q-less q-greater)
(q-inverse q-greater q-less)
(q-inverse q-equal q-equal)

(:rules models
 ;; COPY INEQUALITY RELATIONS from CDs up to MODELS
 ((cd ?m1 ?cd1)
  (:retrieve (cd-ineq ?cd1 ?rel ?vm1 ?vm2))
  ->
  (model-ineq ?vm1 ?rel ?vm2 ?m1))
 ((cd ?m1 ?cd1)
  (:retrieve (not (cd-ineq ?cd1 ?rel ?vm1 ?vm2)))
  ->
  (not (model-ineq ?vm1 ?rel ?vm2 ?m1)))

 ;; Copy inequalities between magnitudes in model up to model:
 ((qspace ?m1 ?var ?qsp)
  ->
  (:rules ?qsp
   ((less ?mag1 ?mag2)
    ->
    (model-ineq ?mag1 q-less ?mag2 ?m1))))
 ((qspace ?m1 ?var ?qsp)
  ->
  (:rules ?qsp
   ((equal ?mag1 ?mag2)
    ->
    (model-ineq ?mag1 q-equal ?mag2 ?m1))))
 ((qspace ?m1 ?var ?qsp)
  ->
  (:rules ?qsp
   ((not (less ?mag1 ?mag2))
    ->
    (not (model-ineq ?mag1 q-less ?mag2 ?m1)))))
 ((qspace ?m1 ?var ?qsp)
  ->
  (:rules ?qsp
   ((not (equal ?mag1 ?mag2))
    ->
    (not (model-ineq ?mag1 q-equal ?mag2 ?m1))))))

(:rules variables
 ;; QSPACE MEMBERSHIP
 ;; If a model variable is in a model-ineq with a magnitude then
 ;; that magnitude is in its quantity space (in the model):
 ;;
 ((model-ineq ?v1 ?or1 ?mag1 ?m1)
  ;; Optimization:
  (:unp (isa ?mag1 variables))
  (isa ?mag1 magnitudes)
  (:retrieve (qspace ?m1 ?v1 ?qsp1))
  ->
  (member ?qsp1 ?mag1))
```

```
 ((not (model-ineq ?v1 ?or1 ?mag1 ?m1))
  ;; Optimization:
  (:unp (isa ?mag1 variables))
  (isa ?mag1 magnitudes)
  (:retrieve (qspace ?m1 ?v1 ?qsp1))
  ->
  (member ?qsp1 ?mag1)))

(:rules variables-magnitudes
 ;; INVERSES --
 ((model-ineq ?x ?ineq1 ?y ?m1)
  (:retrieve (q-inverse ?ineq1 ?ineq2))
  ->
  (model-ineq ?y ?ineq2 ?x ?m1))

 ;; TRANSITIVELY CLOSE Q-LESS (carefully):
 ((model-ineq ?x q-less ?y ?m1)
  ;; Don't close across zero:
  (:neq ?y zero)
  (:retrieve (model-ineq ?y q-less ?z ?m1))
  ->
  (model-ineq ?x q-less ?z ?m1))

 ;; PROPAGATE Q-LESS through q-equal.
 ((model-ineq ?x q-equal ?y ?m1)
  (:neq ?x ?y) (:neq ?y zero)
  ->
  (:retrieve (model-ineq ?y q-less ?z ?m1))
  (model-ineq ?x q-less ?z ?m1))
 ((model-ineq ?y q-less ?z ?m1)
  (:neq ?y zero)
  ->
  (:retrieve (model-ineq ?y q-equal ?x ?m1))
  (:neq ?x ?y)
  (model-ineq ?x q-less ?z ?m1))

 ((model-ineq ?z q-less ?y ?m1)
  (:retrieve (model-ineq ?y q-equal ?x ?m1))
  (:neq ?x ?y) (:neq ?y zero)
  ->
  (model-ineq ?z q-less ?x ?m1))

 ;;--------------------------------------------------
 ;; not MODEL-INEQS

 ;; Backchain to conclude negation of model-ineq's.
 ((not (model-ineq ?x q-less ?y ?m1))
  <-
  (:retrieve (model-ineq ?x q-greater ?y ?m1)))
 ((not (model-ineq ?x q-greater ?y ?m1))
  <-
  (:retrieve (model-ineq ?x q-equal ?y ?m1)))
 ((not (model-ineq ?x q-greater ?y ?m1))
```

```
  <-
  (:retrieve (model-ineq ?x q-less ?y ?m1)))
 ((not (model-ineq ?x q-greater ?y ?m1))
  <-
  (:retrieve (model-ineq ?x q-equal ?y ?m1)))
 ((not (model-ineq ?x q-equal ?y ?m1))
  <-
  (:retrieve (model-ineq ?x q-greater ?y ?m1)))
 ((not (model-ineq ?x q-equal ?y ?m1))
  <-
  (:retrieve (model-ineq ?x q-less ?y ?m1)))


 ;; INVERSES
 ((not (model-ineq ?x ?ineq1 ?y ?m1))
  (:retrieve (q-inverse ?ineq1 ?ineq2))
  ->
  (not (model-ineq ?y ?ineq2 ?x ?m1)))

 ;; TRANSITIVE CLOSE (not q-less)
 ((not (model-ineq ?x q-less ?y ?m1))
  (:neq ?y zero)
  (:retrieve (not (model-ineq ?y q-less ?z ?m1)))
  ->
  (not (model-ineq ?x q-less ?z ?m1)))

 ;; PROPAGATE not Q-LESS through q-equal:
 ((model-ineq ?y q-equal ?x ?m1)
  (:neq ?y zero) (:neq ?x ?y)
  ->
  (:retrieve (not (model-ineq ?y q-less ?z ?m1)))
  (not (model-ineq ?x q-less ?z ?m1)))
 ((not (model-ineq ?y q-less ?z ?m1))
  (:neq ?y zero)
  ->
  (:retrieve (model-ineq ?y q-equal ?x ?m1))
  (:neq ?x ?y)
  (not (model-ineq ?x q-less ?z ?m1)))

 ((not (model-ineq ?z q-less ?y ?m1))
  (:retrieve (model-ineq ?y q-equal ?x ?m1))
  (:neq ?x ?y) (:neq ?y zero)
  ->
  (not (model-ineq ?z q-less ?x ?m1)))

 ;; RECOGNIZE CONTRADICTIONS
 ;;
 ((model-ineq ?x q-less ?x ?model) ->
  (not (model-ineq ?x q-less ?x ?model))))

;; Finally, some rules for deducing inequalities from tmags:
;;
(:rules variables
 ((tmag ?var ?model ?time ?mag1)
```

```
             (:retrieve (model-ineq ?var ?ineq ?mag2 ?model))
             ;; Optimization:
             (:unp (isa ?mag2 variables))
             (isa ?mag2 magnitudes)
             ->
             (model-ineq ?mag1 ?ineq ?mag2 ?model))

            ((tmag ?var1 ?model ?time ?mag1)
             (:retrieve (model-ineq ?var1 ?ineq ?var2 ?model))
             ;; Optimization:
             (:unp (isa ?var2 magnitudes))
             (isa ?var2 variables)
             (:retrieve (tmag ?var2 ?model ?time ?mag2))
             ->
             (model-ineq ?mag1 ?ineq ?mag2 ?model))

            ((tmag ?var ?model ?time ?mag1)
             (:retrieve (not (model-ineq ?var ?ineq ?mag2 ?model)))
             ;; Optimization:
             (:unp (isa ?mag2 variables))
             (isa ?mag2 magnitudes)
             ->
             (not (model-ineq ?mag1 ?ineq ?mag2 ?model)))

            ((tmag ?var1 ?model ?time ?mag1)
             (:retrieve (not (model-ineq ?var1 ?ineq ?var2 ?model)))
             ;; Optimization:
             (:unp (isa ?var2 magnitudes))
             (isa ?var2 variables)
             (:retrieve (tmag ?var2 ?model ?time ?mag2))
             ->
             (not (model-ineq ?mag1 ?ineq ?mag2 ?model))))

           (:rules magnitudes
            ((model-ineq ?mag2 ?ineq ?var ?model)
             ;; Optimization:
             (:unp (isa ?var magnitudes))
             (:retrieve (tmag ?var ?model ?time ?mag1))
             ->
             (model-ineq ?mag2 ?ineq ?mag1 ?model))
            ((not (model-ineq ?mag2 ?ineq ?var ?model))
             ;; Optimization:
             (:unp (isa ?var magnitudes))
             (:retrieve (tmag ?var ?model ?time ?mag1))
             ->
             (not (model-ineq ?mag2 ?ineq ?mag1 ?model))))))

(a-assert "Relations"
          '((:slot 2-relation
                   (cds 2-relations variables variables))
           (:slot 3-relation
                   (cds 3-relations variables variables variables))))

(a-assert "Influences"
```

```
          '((:slot influence (cds influence-types variables variables)
                   :comment "(influence inf InfluencER InfluencEE)")

            ;; influence-types =
            ;; direct-influence-types UNION indirect-influence-types
            (union-of influence-types direct-influence-types)
            (union-of influence-types indirect-influence-types)
            ))

(a-assert "(Q= v1 v2) implies (= v1 v2)"
          '((:rules cds
             ((2-relation ?cd q= ?v1 ?v2)
              (part-of ?cd ?model)
              ->
              (model-ineq ?v1 q-equal ?v2 ?model)))))

(a-assert "Correspondences"
          '((:slot correspondence
                   (cds variables-magnitudes variables-magnitudes
                        variables-magnitudes variables-magnitudes))

            ;; Correspondences add magnitudes to their variables' qspaces:
            (:srules (:slot cd)
             ((cd ?model ?cd1) (correspondence ?cd1 ?var ?mag ?vm3 ?vm4)
              (isa ?var variables) (isa ?mag magnitudes)
              (qspace ?model ?var ?var-qspace)
              ->
              (member ?var-qspace ?mag))
             ((cd ?model ?cd1) (correspondence ?cd1 ?vm1 ?vm2 ?var ?mag)
              (isa ?var variables) (isa ?mag magnitudes)
              (qspace ?model ?var ?var-qspace)
              ->
              (member ?var-qspace ?mag)))))

(a-assert "Quantities"
          ;; Quantities are relations which are associated with sets of objects.
          ;; e.g. (quantity physical-objects mass) means that physical objects
          ;; can have a variable called mass.
          ;;
          ;; Variable-of is the backlink from any variable to the object it is
          ;; a property of.
          ;;
          '((:slot  quantity (sets quantities))
            (:slot variable-of (variables objects))

            ;; When we learn ?q1 is a quantity slot:
            (:srules (:slot quantity)
             ((quantity ?set1 ?q1)
              ->
              (type-slot ?q1 ?set1 variables)
              ;; Backlink it to variable-of.
              (backlink ?q1 (:slot variable-of))
              ;; And make sure its always there when you need it:
              (:srules ?q1
```

```
              ((?q1 ?x ?var)
               <-
               (:forc ?var (?q1 ?x ?var))))
             ))))

(a-assert "Constraints"
          ;; Currently we only copy 3-relations in the cds up to constraints
          ;; in the model. Generation of constraints from other relations and from
          ;; influences in done in Lisp.
          ;;
          '((:slot constraint-rel (constraints relations)
                   :cardinality 1)
            (:slot var1 (constraints variables) :cardinality 1)
            (:slot var2 (constraints variables) :cardinality 1)
            (:slot var3 (constraints variables) :cardinality 1)

            (:slot constraint (models constraints)
                   :backlink part-of)

            ;; When we learn a 3-relation for a cd, then create a
            ;; constraint frame and attach it to the cd's model.
            (:srules (:slot cd)
             ((cd ?m1 ?cd1)
              (3-relation ?cd1 ?3rel1 ?var1 ?var2 ?var3)
              ->
              (:forc ?cons1
               (constraint ?m1 ?cons1)
               (constraint-rel ?cons1 ?3rel1)
               (var1 ?cons1 ?var1)
               (var2 ?cons1 ?var2)
               (var3 ?cons1 ?var3)))
             ((cd ?m1 ?cd1)
              (2-relation ?cd1 S0+ ?var1 ?var2)
              ->
              (:forc ?cons1
               (constraint ?m1 ?cons1)
               (constraint-rel ?cons1 S0+)
               (var1 ?cons1 ?var1)
               (var2 ?cons1 ?var2))))))

(a-assert "Index constant vars under the model"
          '((:slot constant (models variables))

            (:rules variables
             ((model-ineq ?var q-equal ?mag ?m1)
              (isa ?var variables)
              (isa ?mag magnitudes)
              ->
              (constant ?m1 ?var)))

            (:rules models
             ;; Too bad this requires three rules ...
             ((constraint ?m1 ?con1)
              (constant ?m1 (var1 ?con1))
```

```
             (constant ?m1 (var2 ?con1))
             ->
             (constant ?m1 (var3 ?con1)))
            ((constraint ?m1 ?con1)
             (constant ?m1 (var2 ?con1))
             (constant ?m1 (var3 ?con1))
             ->
             (constant ?m1 (var1 ?con1)))
            ((constraint ?m1 ?con1)
             (constant ?m1 (var1 ?con1))
             (constant ?m1 (var3 ?con1))
             ->
             (constant ?m1 (var2 ?con1)))
            ((constant ?m1 ?var1)
             (tmag ?var1 ?m1 ?1 ?mag1)
             ->
             (model-ineq ?var1 q-equal ?mag1 ?m1))
            ((constant ?m1 ?v)
             ->
             ;; It really is std at all times, but only explicitly
             ;; add std at the initial time.
             (tdir ?v ?m1 (initial-time ?m1) std))
            )))

(a-assert "Corresponding Values"
         '((:slot corresponding-value
                  (constraints magnitudes magnitudes magnitudes))

           (:rules models
            ;; m1 = m3 -> m2 = 0
            ((constraint ?model1 ?cons1) (constraint-rel ?cons1 add)
             (qspace ?model1 (var1 ?cons1) ?var1-qspace)
             (member ?var1-qspace ?mag1)
             (model-ineq ?mag1 q-equal ?mag3 ?model1)
             (qspace ?model1 (var3 ?cons1) ?var3-qspace)
             (member ?var3-qspace ?mag3)
             (qspace ?model1 (var2 ?cons1) ?var2-qspace)
             ->
             (member ?var2-qspace zero)
             (corresponding-value ?cons1 ?mag1 zero ?mag3))

            ;; The other cases (for v1 and v3) are done with
            ;; one rule (using q-inverse)  and only apply if tmags
            ;; are known for v1 and v3.
            ;;
            ((constraint ?model1 ?cons1) (constraint-rel ?cons1 add)
             (tmag (var1 ?cons1) ?model1 ?t ?mag1)
             (tmag (var3 ?cons1) ?model1 ?t ?mag3)
             (model-ineq ?mag1 ?ineq ?mag3 ?model1)
             (:neq ?ineq q-equal)
             ;; Get var2 so don't have to do it in :forc.
             (var2 ?cons1 ?var2)
             ->
             (:forc ?mag2 (tmag ?var2 ?model1 ?t ?mag2))
```

```
 (:lisp (amsg "Adding cv for the add constraint: ~a ~a"
              '(?var1 + ?var2 = ?var3)
              '(?mag1 ?mag2 ?mag3)))
 (model-ineq ?mag2 (q-inverse ?ineq) zero ?model1)
 (corresponding-value ?cons1 ?mag1 ?mag2 ?mag3))

;;
;; m2 = m3 -> m1 = 0
((constraint ?model1 ?cons1) (constraint-rel ?cons1 add)
 (qspace ?model1 (var2 ?cons1) ?var2-qspace)
 (member ?var2-qspace ?mag2)
 (model-ineq ?mag2 q-equal ?mag3 ?model1)
 (qspace ?model1 (var3 ?cons1) ?var3-qspace)
 (member ?var3-qspace ?mag3)
 (qspace ?model1 (var1 ?cons1) ?var1-qspace)
 ->
 (member ?var1-qspace zero)
 (corresponding-value ?cons1 zero ?mag2 ?mag3))
;;
;; a variant of the above rule.  We do not store
;; model-ineqs for (= x x), so if m2 is exactly the same
;; frame as m3 we need to handle it differently.
((constraint ?model1 ?cons1) (constraint-rel ?cons1 add)
 (qspace ?model1 (var2 ?cons1) ?var2-qspace)
 (member ?var2-qspace ?mag)
 (qspace ?model1 (var3 ?cons1) ?var3-qspace)
 (member ?var3-qspace ?mag)
 (qspace ?model1 (var1 ?cons1) ?var1-qspace)
 ->
 (member ?var1-qspace zero)
 (:lisp (amsg "Adding cv for the add constraint: ~a ~a"
              '?cons1
              '(zero ?mag ?mag)))
 (corresponding-value ?cons1 zero ?mag ?mag))

;; Again, the other cases (for v2 and v3) are done
;; with one rule (using q-inverse) and only apply if
;; tmags are known for v2 and v3.
;;
((constraint ?model1 ?cons1) (constraint-rel ?cons1 add)
 (tmag (var2 ?cons1) ?model1 ?t ?mag2)
 (tmag (var3 ?cons1) ?model1 ?t ?mag3)
 (model-ineq ?mag2 ?ineq ?mag3 ?model1)
 (:neq ?ineq q-equal)
 ;; Get var1 so don't have to do it in :forc.
 (var1 ?cons1 ?var1)
 ->
 (:forc ?mag1 (tmag ?var1 ?model1 ?t ?mag1))
 (:lisp (amsg "Adding cv for the add constraint: ~a ~a"
              '(?var1 + ?var2 = ?var3)
              '(?mag1 ?mag2 ?mag3)))
 (model-ineq ?mag1 (q-inverse ?ineq) zero ?model1)
 (corresponding-value ?cons1 ?mag1 ?mag2 ?mag3))))))
```

```
(defun library ()

  ;; Organization:
  ;;   Sets and relations
  ;;   Views
  ;;   Processes
  (a-assert "Containers and their contents."
           '((:taxonomy
               (physobs (containers
                          (finite-containers))
                        (contained-stuff
                          (contained-liquids)
                          (contained-gases))))))

  (a-assert "Rules & Slots for Containers and their contents."
           '((:slot container (contained-stuff containers))
             (:slot contents  (containers contained-stuff)
                    :inverse  container)
             (:slot can-contain (containers contained-stuff))

             ;; A container can-contain its contents.
             (:rules (:slot contents)
              ((contents ?can1 ?x) -> (can-contain ?can1 ?x)))

             ;; If the contained stuff is part-of a model, then so is
             ;; its container.
             (:srules (:slot container)
              ((container ?cs1 ?can1) (part-of ?cs1 ?m1)
               ->
               (entity ?m1 ?can1)))

             ;; If a container is in a model, then so are its contents.
             (:srules (:slot contents)
              ((contents ?can1 ?cs1) (part-of ?can1 ?m1)
               ->
               (entity ?m1 ?cs1)))))

  (a-assert "Paths and connections"
           '((:taxonomy (objects (paths
                                   (heat-paths)
                                   (fluid-paths))
                                 (heat-sources)
                                 (heat-sinks)))
             (:taxonomy (slots (connections)))))

  (a-assert "Rules for Paths and connections"
           ;; A path is an object which may have attributes such as
           ;; alignment or flow-rate.  A connection is a slot which
           ;; relates a source to a destination over a path.
           '(
             ;; Paths.
             (:slot aligned (paths booleans))
```

```
            (:slots flow-rate pressure-difference)
            (quantity fluid-paths (:slot flow-rate))
            (quantity fluid-paths (:slot pressure-difference))

            ;; Connections.
            (:slot heat-connection  (heat-sources heat-paths heat-sinks))
            (isa (:slot heat-connection) connections)
            (:slot fluid-connection (containers fluid-paths containers))
            (isa (:slot fluid-connection) connections)

            ;; Connections.
            ;; When there is an aligned path from a physob in a model to
            ;; another physob then put the second physob in the model.
            (:srules connections
             ((?con1 ?po1 ?path1 ?po2) (aligned ?path1 true)
              (part-of ?po1 ?m1)
              ->
              (entity ?m1 ?po2)))))

(a-assert "States and Material"
          '((:taxonomy (objects
                        (states
                          liquid-state
                          gaseous-state
                          solid-state)
                        (materials water)))))

(a-assert "Rules for States"
          '((complete states True)

            (:slot state (contained-stuff states) :cardinality 1
                   :comment "state links a contained stuff and its state")

            (:slot same-state (contained-stuff contained-stuff)
                   :inverse same-state
                   :comment "same-state links two contained
                             stuff with the same state")
            (isa (:slot same-state) equivalence-relations)

            ;; Forward and Backward rules for deducing same-state
            (:srules (:slot same-state)
             ((same-state ?cs1 ?cs2) (state ?cs1 ?s1) -> (state ?cs2 ?s1))
             ((same-state ?cs1 ?cs2) <- (state ?cs1 ?s1) (state ?cs2 ?s1)))

            ;; Contained stuff with state liquid are contained liquids:
            (:rules contained-stuff
             ((state ?po1 liquid-state)
              ->
              (isa ?po1 contained-liquids)))))

(a-assert "Rules for Materials"
          '((:slot material (contained-stuff materials)
                   :cardinality 1
                   :comment "Material links a contained stuff and its material.")
```

```
            (:slot same-material (contained-stuff contained-stuff)
                   :inverse same-material
                   :comment
                   "Same-material links two contained stuff
                    with the same material.")
            (isa (:slot same-material) equivalence-relations)

            ;; like states, need forward and backward rules for deducing
            ;; whether two stuffs have the same material.
            (:srules (:slot same-material)
             ((same-material ?cs1 ?cs2) (material ?cs1 ?mat1) ->
              (material ?cs2 ?mat1))
             ((same-material ?cs1 ?cs2) <-
              (material ?cs1 ?mat1) (material ?cs2 ?mat1)))
            ))

(a-assert "Identity of contained stuff"
          '((:rules contained-stuff
             ((container ?x ?conx)
              (material ?x ?mx) (state ?x ?sx)
              (contents ?conx ?x2) (:NEQ ?x ?x2)
              (material ?x2 ?mx) (state ?x2 ?sx)
              ->
              (coreferent ?x ?x2)))))

(a-assert "Processes and views"
          '((:taxonomy (views (physical-views) (thermal-views)))
            (:taxonomy (processes (fluid-flow-processes)
                                  (overflow-processes)))))

(a-assert "Physical and thermal views"
          '((:slot thermal-view (physobs thermal-views)
                   :cardinality 1)
            (:slot physical-view (physobs physical-views)
                   :cardinality 1)
            (isa (:slot thermal-view) considerations)
            (isa (:slot physical-view) considerations)))

(a-assert "Physical view of physical object"
          '(
            ;; physob of model -> consider its pv.
            (:rules physobs
             ((part-of ?po1 ?m1)
              (isa ?m1 models)
              ->
              (consider ?m1 (:slot physical-view) ?po1)))
            ;;
            ;; Quantities for the physical view:
            (:1-slots mass volume pressure)
            (quantity physobs (:slot mass))
            (quantity physobs (:slot volume))
            (quantity physobs (:slot pressure))
```

```
            (:rules physobs
             ;; Any physical object has a physical view:
             ((physical-view ?x ?view)
              <-
              (:forc ?view (physical-view ?x ?view)))

             ;; When you add the physical view of a physob, forc
             ;; a mass, volume, and pressure.  Index the inequalities
             ;; under the CD.
             ((physical-view ?x ?view)
              (mass ?x ?mx)
              (volume ?x ?vx)
              (pressure ?x ?px)
              ->
              (:lisp (amsg "Instantiating the physical view of ~a" '?x))
              (variable ?view ?mx)
              (variable ?view ?vx)
              (variable ?view ?px)
              ;;
              ;; Mass not less zero
              (not (cd-ineq ?view q-less ?mx zero))
              ;; Volume not less zero
              (not (cd-ineq ?view q-less ?vx zero))

              ;; And in a sharp break with tradition:
              (correspondence ?view ?mx zero ?vx zero)))

            (:rules physical-views
             ;; Copy over the physical-view if its physob copies over:
             ((copies-over ?view ?m1 ?m2)
              <-
              (copies-over (view-of ?view) ?m1 ?m2)))))

(a-assert "Copy physobs to subseq-model"
          '((:rules physobs
             ((copies-over ?po ?m1 ?m2)
              <-
              (mass ?po ?mass)
              (:unp (tmag ?mass ?m2 (initial-time ?m2) zero)
                    (tdir ?mass ?m2 (initial-time ?m2) std))))))

;; If we were concerned about overflow, then this view would put the top height
;; of the container in the qspace of the fluid-level.
(a-assert "Physical view of a container."
          '(
            ;; Quantities:
            (:1-slots fluid-level top-height bottom-height)
            (quantity containers (:slot fluid-level))
            (quantity containers (:slot top-height))
            (quantity containers (:slot bottom-height))
            ;;
            (:rules containers
             ;; If we assert the physical of view of a container, then
             ;; forc a fluid-level, top-height, bottom-height.  Index
```

```
            ;; the inequalities under the CD.
            ;;
            ;; note that this is forward, as opposed to the backward
            ;; rule for the physical view.  This is because it builds on
            ;; the physical view created by the rule for the physical view.
            ;;
            ((physical-view ?x ?view)
             (volume ?x ?vx)
             (fluid-level ?x ?flx)
             (top-height ?x ?thx)
             (bottom-height ?x ?bhx)
             (mass ?x ?massx)
             ->
             (variable ?view ?vx)
             (variable ?view ?flx)
             (variable ?view ?thx)
             (variable ?view ?bhx)
             (variable ?view ?massx)
             ;;
             ;; The the diff between top and bottom qprops the
             ;; volume.  We'll blow off the qprop as it gives no
             ;; additional inference power, and would force us to
             ;; build a local var, which we haven't allowed for.
             (3-relation ?view ADD ?vx ?bhx ?thx)
             ;;
             ;; bottom height is less than the top height.
             (cd-ineq ?view Q-LESS ?bhx ?thx)
             ;; Bottom height <= fluid level <= top height
             ;; =
             ;; (not fluid level < bottom height) & (not top height < fluid level)
             (not (cd-ineq ?view Q-LESS ?flx ?bhx))
             (not (cd-ineq ?view Q-LESS ?thx ?flx))
             (greater ?massx zero)
             (:lisp (amsg "Instantiating the physical view for
                           container ~s" '?x))))))

(a-assert "Physical view of a contained liquid."
          '(
            ;; Quantities:
            (:1-slots level)
            (quantity contained-liquids (:slot level))
            ;;
            (:rules contained-liquids
             (
              ;; Once the PV of a liquid has been instantiated, then
              ;; give the liquid a level, ((q+ mass vol) (0 0)), ((q+
              ;; vol lev)(0 0)), ((q+ lev press)(0 0)).  Index
              ;; everything under the physical view, a CD.
              (physical-view ?x ?view)
              (mass ?x ?mx) (volume ?x ?vx) (pressure ?x ?px)
              (level ?x ?lx)
              ->
              (:lisp (amsg "Defining influences for the physical view ~
                                 of contained-liquid ~s." '?x))
```

```
               (variable ?view ?lx)

               (influence ?view Q+ ?mx ?vx)

               (influence ?view Q+ ?vx ?lx)
               (correspondence ?view ?vx zero ?lx zero)
               ;;
               (influence ?view Q+ ?lx ?px)
               (correspondence ?view ?lx zero ?px zero))

              (
               ;; Establish the relations between the level and
               ;; pressure of the liquid and its container.  Then setup
               ;; the correspondences between level=bottom, vol=0,
               ;; level=top, vol=con.vol.  Index everything under the view.
               (physical-view ?x ?view)
               (volume ?x ?vx) (pressure ?x ?px) (level ?x ?lx)
               (container ?x ?cx)
               (pressure ?cx ?conx-press) (volume ?cx ?conx-vol)
               (top-height ?cx ?conx-top) (bottom-height ?cx ?conx-bottom)
               (fluid-level ?cx ?conx-level)
               ->
               (:lisp (amsg "Defining the correspondences ~
                                   for the physical view of the contained-liquid ~s."
                             '?x))
               ;;notE: since a cl cannot exist unless it has a
               ;;container, we are justified in associating these corrs
               ;;and ineqs with the cl.
               ;;
               ;; (q= p p) becomes false when head is added:
               ;;
               (2-relation ?view Q= ?px ?conx-press)
               ;;
               ;; the level of the contained liquid is the
               ;; fluid level of the container from the container view we have
               ;; the bottom-height <= container-level <= top-height
               ;; which gives us the relation between the heights
               ;; and the liquid level
               ;;
               (2-relation ?view Q= ?lx ?conx-level)
               (correspondence ?view ?lx ?conx-bottom ?vx zero)
               (correspondence ?view ?lx ?conx-top ?vx ?conx-vol)
               ;;
               ;; and the container volume is not less then the contents volume
               ;;
               (not (cd-ineq ?view Q-LESS ?conx-vol ?vx))))))

(a-assert "Fluid flow process"
          '(
             ;; A fluid-flow process has a path.
             (:slot path (processes paths))
             (:rules physobs
              (
               ;; Given an aligned path and a fluid-connection between
```

```
              ;; two containers, at least one of which has a contents,
              ;; create a process and index the variables and
              ;; influences under it.
              (fluid-connection ?c1 ?fp1 ?c2)
              (part-of ?c1 ?m1) (isa ?m1 models)
              (flow-rate ?fp1 ?flow-rate)
              (pressure-difference ?fp1 ?pressure-difference)
              (contents ?c1 ?cl1) (isa ?cl1 contained-liquids) (mass ?cl1 ?mcl1)
              (aligned ?fp1 true)
              (pressure ?c1 ?pc1)
              (pressure ?c2 ?pc2)

              ;; not requiring that the pressure in ?c1 greater than
              ;; in ?c2 Question: should the process ensure that the
              ;; entities used in it are all part-of ?m1 ?  In this
              ;; process we know that ?c1 is in th model, and due to
              ;; the fluid-connection and container rules, we are sure
              ;; that ?c2 and the cls are also there.
              ;;
              ->
              (:lisp (amsg "Instantiating the fluid flow process ~
                                  from ~a to ~a." '?c1 '?c2))

              (:forc ?process
                     (cd ?m1 ?process)
                     (isa ?process fluid-flow-processes)
                     (path ?process ?fp1)
                     (isa ?process processes))
              (variable ?process ?flow-rate)
              (variable ?process ?pressure-difference)
              (correspondence ?process ?flow-rate zero ?pressure-difference zero)
              ;; c2.pressure + flow-rate= c1.pressure or c1 - c2 =flow-rate
              (3-relation ?process ADD ?pc2 ?pressure-difference ?pc1)
              (influence ?process Q+ ?pressure-difference ?flow-rate)
              (influence ?process I- ?flow-rate ?mcl1)
              (:forc ?cl2
               (contents ?c2 ?cl2)
               (same-material ?cl1 ?cl2)
               (same-state ?cl1 ?cl2))
              (influence ?process I+ ?flow-rate (mass ?cl2))))

            (:rules fluid-flow-processes
             ((copies-over ?proc ?m1 ?m2)
              <-
              (aligned (path ?proc) true))))))

(defun portals ()
  ;; When the fluid-level in the container reaches the level
  ;; of the portal, and the portal is open, then there is flow
  ;; out of the portal.
  ;;
  ;; Portals are not physical objects.
  (a-assert "Portal taxonomy"
            '((:taxonomy (fluid-paths (portals)))
```

```
            (:taxonomy (processes (portal-flow-processes)))
            (:taxonomy (views (portal-views)))))

(a-assert "Simple Portals"
          ;; Portals have heights, and they are associated with a
          ;; container.  Inherit: aligned, flow-rate from fluid-paths.
          ;; Don't want to use fluid-connections, as that might yield
          ;; a fluid-flow process.
          '((:1-slots height)
            (quantity portals (:slot height))
            ;; Head is the pressure due to the column of fluid above a portal.
            (:slots head)
            (quantity portals (:slot head))
            ;; Don't need since no longer using s+
            ;;(quantity portals (:slot portal-pressure))
            (:slot portal (containers portals))
            (:slot container-of (portals containers)
                   :inverse portal)

            (:rules containers
             ((portal ?can ?port)
              (part-of ?can ?m) (isa ?m models)
              ->
              (entity ?m ?port)))

            (:rules portals
             ;; Simple portals don't change their height.
             ((height ?x ?height) (part-of ?x ?model) (isa ?model models)
              -> (constant ?model ?height)))))

(a-assert "View of portals"
          '((:slot portal-view (portals portal-views)
                   :cardinality 1)
            (isa (:slot portal-view) considerations)
            (:rules portals
             ((part-of ?port ?m) (isa ?m models)
              ->
              (consider ?m (:slot portal-view) ?port))

             ((portal-view ?p ?v) <- (:forc ?v (portal-view ?p ?v)))

             ((portal-view ?p ?view)
              (height ?p ?ph)

              (container-of ?p ?can)
              (top-height ?can ?th)
              (bottom-height ?can ?bh)
              ->
              (:lisp (amsg "Instantiating portal view of ~a." '?p))

              (variable ?view ?ph)

              ;; bottom height < portal height < top height
              (cd-ineq ?view q-less ?bh ?ph)
```

```
              (cd-ineq ?view q-less ?ph ?th)

              (not (cd-ineq ?view Q-LESS ?ph zero)))

             ;; The portal height is an important magnitude for the fluid level
             ;; of the container.
             ((container-of ?port ?can)

              (part-of ?port ?model)
              (part-of ?can ?model)

              (height ?port ?portal-height)
              (level (contents ?can) ?fluid-level)

              (constant ?model ?portal-height)
              (tmag ?portal-height ?model ?t1 ?portal-height-mag)

              (qspace ?model ?fluid-level ?qspace)
              ->
              ;; In general  processes and views should add
              ;; mags into qspaces.  They know what's important to them.
              (member ?qspace ?portal-height-mag)))))

(a-assert "Portal flow process"
          '((:slot process-portal (portal-flow-processes portals))
            ;;
            (:rules containers
             ;; 1. container.fluid-level < portal.height,
             ;; we don't do anything but set up for the transition.
             ;;
             ;; 2. container.fluid-level >= portal.height
             ;; regular old fluid flow.
             ((portal ?container ?portal)
              (part-of ?container ?model)
              (isa ?model models)
              (aligned ?portal true)

              (height ?portal ?portal-height)
              ;;(fluid-level ?container ?container-fluid-level)
              (level (contents ?container) ?container-fluid-level)

              (tmag ?container-fluid-level ?model ?t1 ?container-level-mag)
              (tmag ?portal-height ?model ?t1 ?portal-height-mag)
              (model-ineq ?container-level-mag q-less ?portal-height-mag ?model)

              (constant ?model ?portal-height)
              ->
              (:lisp (amsg "Setting up region transition for portal flow ~
                            from portal ~a in ~a." '?portal '?container))
              (region-transition ?model ?container-fluid-level
                                 ?portal-height-mag inc
                                 "Fluid level now above portal,
                                  so set up for flow
                                  out of portal"))
```

```
;; The following rule is for the case where
;; portal-height <= fluid-level.
((portal ?container ?portal)
 (part-of ?container ?model)
 (isa ?model models)
 (aligned ?portal true)

 (height ?portal ?portal-height)
 ;;(fluid-level ?container ?container-fluid-level)
 (level (contents ?container) ?container-fluid-level)

 (tmag ?container-fluid-level ?model ?t1 ?container-level-mag)
 (tmag ?portal-height ?model ?t1 ?portal-height-mag)

 (model-ineq ?container-level-mag q-equal
             ?portal-height-mag ?model)

 (flow-rate ?portal ?flow-rate)
 (head ?portal ?head)
 ;;(portal-pressure ?portal ?portal-pressure)

 (contents ?container ?fluid)
 (isa ?fluid contained-liquids)
 (mass ?fluid ?fluid-mass)
 ->
 (:lisp (amsg "Instantiating the Portal flow process ~
 from portal ~a in ~a." '?portal '?container))
 (:forc ?process
  (cd ?model ?process)
  (isa ?process portal-flow-processes)
  (process-portal ?process ?portal))
 (variable ?process ?flow-rate)
 (variable ?process ?head)
 ;;(variable ?process ?portal-pressure)
 (correspondence ?process ?head zero ?flow-rate zero)
 (3-relation ?process ADD
             ?head ?portal-height ?container-fluid-level)
 (influence ?process Q+ ?head ?flow-rate)
 (influence ?process I- ?flow-rate ?fluid-mass)
 (region-transition ?model ?container-fluid-level
                    ?portal-height-mag dec
                    "Fluid level now below portal, ~
                    so no more flow out of portal"))

((portal ?container ?portal)
 (part-of ?container ?model)
 (isa ?model models)
 (aligned ?portal true)

 (height ?portal ?portal-height)
 ;;(fluid-level ?container ?container-fluid-level)
 (level (contents ?container) ?container-fluid-level)
```

```
    (tmag ?container-fluid-level ?model ?t1 ?container-level-mag)
    (tmag ?portal-height ?model ?t1 ?portal-height-mag)

    (model-ineq ?portal-height-mag q-less ?container-level-mag ?model)

    (flow-rate ?portal ?flow-rate)
    (head ?portal ?head)
    ;;(portal-pressure ?portal ?portal-pressure)

    (contents ?container ?fluid)
    (isa ?fluid contained-liquids)
    (mass ?fluid ?fluid-mass)
    ->
    (:lisp (amsg "Instantiating the Portal flow process ~
                   from portal ~a in ~a." '?portal '?container))
    (:forc ?process
     (cd ?model ?process)
     (isa ?process portal-flow-processes)
     (process-portal ?process ?portal))
    (variable ?process ?flow-rate)
    (variable ?process ?head)
    (correspondence ?process ?head zero ?flow-rate zero)
    (3-relation ?process ADD
                ?head ?portal-height ?container-fluid-level)
    (influence ?process Q+ ?head ?flow-rate)
    (influence ?process I- ?flow-rate ?fluid-mass)
    (region-transition ?model ?container-fluid-level
                       ?portal-height-mag dec
                       "Fluid level now below portal, ~
                        so no more flow out of portal")))

  ;; If we wanted the portal flow process to die
  ;; when we entered a region in which the level
  ;; was below the portal then we would do it here:
  (:rules portal-flow-processes
   ((copies-over ?proc ?m1 ?m2)
    <-
    (aligned (path ?proc) true))))))
```

Then the code to test QPC on a simple utube with a portal:

```
;;; Copyright: Adam Farquhar and James Crawford, 1990.
;;;
;;; The functions below set up two test cases for QPC: A simple utube and
;;; a utube with a portal.

(in-package 'user)

(defun setup (example &optional load)
  (cond (load
         (format t "~& Resetting Algernon.~%")
```

```
        (reset-algy)
        (load-kb load))
       (t
        (format t "~& Resetting Algernon.~%")
        (reset-algy)
        (load-common-sense-kb)
        (format t "~& Background knowledge-base loaded.~%")
        (with-no-output
          (basic-qpt)
          (kb-snapshot 'basic)
          (library)
          (kb-snapshot 'lib)
          (portals)
          (kb-snapshot 'portals))))
  (with-no-output (funcall example))
  (kb-snapshot example))


;;;--------------------------------------------------------------------
;;;
;;; Factor out a few common idioms so that they don't have to be
;;; repeated in each example.
;;;

(defun create-scenario (name)
  (a-assert "Consider a scenario."
           `((:clear-slot global-context current-scenario)
             (:create ?scene ,name)
             (current-scenario global-context ?scene))))

(defun create-physob (name)
  (a-assert (format nil "Creating the physob ~a in the current scenario" name)
           `((:create ?a ,name)
             (entity (initial-model (current-scenario global-context)) ?a))))

(defun create-fluid-in (container)
  (a-assert (format nil "Creating the fluid in ~a." container)
           `((:the ?ac
              (contents ,container ?ac)
              (state ?ac liquid-state)))))

(defun create-fluid-path (src dst pipe)
  (a-assert (format nil "Creating a fluid path, ~a, from ~a to ~a." pipe src dst)
           `((:create ?pipe ,pipe)
             (aligned ?pipe true)
             (fluid-connection ,src ?pipe  ,dst))))


;;;--------------------------------------------------------------------
;;;
;;; A simple utube.  Two tanks connected via a fluid path.  Initially
;;; there is liquid in A.

(defun utube ()
```

```
  (format t "~&The utube with fluid initially in A. ~
             The tanks have bottoms at zero, and tops too.~%")
  (create-scenario 'utube)
  (create-physob 'a)
  (create-fluid-in 'a)
  (a-assert "There is a container B."
            '((:create ?b b) (isa ?b containers)))
  (create-fluid-path 'a 'b 'pipe-ab)
  ;; Now we should deduce that B is in the model, as there is an aligned
  ;; connection to A.
  (a-assert "The fluid in A has a quantity greater than zero."
            '((:create ?a* a*)
              (greater ?a* zero)
              (tmag (level (contents a))
                    (initial-model utube) (initial-time (initial-model utube))
                    ?a*)))
  (a-assert "The bottoms of A and B are at zero."
            '((initial-model utube ?model)
              (tmag (bottom-height a) ?model (initial-time ?model) zero)
              (tmag (bottom-height b) ?model (initial-time ?model) zero)))
  (a-assert "Top heights are positive."
            '((initial-model utube ?model)
              (:create ?ath a-top-height-landmark)
              (:create ?bth b-top-height-landmark)
              (greater ?ath zero) (greater ?bth zero)
              (tmag (top-height a) ?model (initial-time ?model) ?ath)
              (tmag (top-height b) ?model (initial-time ?model) ?bth)))
  (a-assert "Hack, a* < atopheight"
            '((greater a-top-height-landmark a*))))

;;;----------------------------------------------------------------------
;;;
;;; The utube with a portal in tank B.
;;;

(defun dripping-utube ()
  (utube)
  (a-assert "B has a portal"
            '((:the ?port (portal b ?port))
              (initial-model utube ?model)
              (aligned ?port true)
              (:create ?h* h*)
              (greater ?h* zero)
              (less ?h* b-top-height-landmark)
              (tmag (height (portal b)) ?model
                    (initial-time ?model) ?h*))))
```

Appendix B

Proofs of Technical Lemmas

This appendix contains the following:

- An overview of the notation of predicate calculus as used in this paper.
- The proof of the soundness theorem (theorem 2).
- The proofs the key lemmas for the Socratic and Partitional Completeness Theorems (lemmas 11 and 13).
- The proofs of the computation lemma (lemma 33), and the $closure_{f\&r}$ lemma (lemma 34).

B.1 A Dialect of Predicate Calculus

To avoid any ambiguity, we here briefly review the notation for a limited dialect of predicate calculus.

The *alphabet* of a predicate calculus consists of the following:

- *Variables*, *constants* and *relations* as in ALL.
- The *propositional constants* **true** and **false**.
- The *connectives* $\wedge$ (conjunction) and $\rightarrow$ (implication).
- The *quantifiers* $\exists$ (there exists) and $\forall$ (for all).

A (predicate calculus) *term* is a constant or a variable. A (predicate calculus) *propositions* is a relation

$$r_i^n(t_1, t_2, \ldots, t_n)$$

where all t_j $(1 \leq j \leq n)$ are terms. Predicate calculus *formulas* are defined as follows:

- Propositions are formulas.
- Propositional constants are formulas.

- If p and q are formulas then so are $(p \wedge q)$ and $(p \rightarrow q)$.
- If p is a formula and v is a variable then $(\exists v : p)$ and $(\forall v : p)$ are formulas.

A model $\mathcal{M}$ for a language L consists of the following:

1. A non-empty set M called the domain of $\mathcal{M}$.
2. An assignment for each constant c in L to an element $c_{\mathcal{M}}$ of the domain.
3. An assignment for each n-ary relation r in L to a subset $r_{\mathcal{M}}$ of M^n.

A *variable assignment*, σ, is a function mapping terms to elements of M. Assume t is a term. If t is a variable then $\sigma(t)$ is the domain element assigned to t. If t is a constant then $\sigma(t)$ is $t_{\mathcal{M}}$. If σ is a variable assignment, d an element of M and v a variable then $\sigma[v/d]$ denotes the variable assignment which differs from σ only in that it assigns v to d.

Finally we can define the semantics of predicate calculus. $\mathcal{M} \models_\sigma p$ can be read '$\mathcal{M}$ is a model of p under the variable assignment σ'. Let r_i^n be a relation, $t_1, \ldots, t_n$ be terms, and p and q be predicate calculus formulas then:

- $\mathcal{M} \models_\sigma r_i^n(t_1, \ldots, t_n)$ iff $(\sigma(t_1), \ldots, \sigma(t_n)) \in (r_i^n)_{\mathcal{M}}$
- $\mathcal{M} \models_\sigma$ **true**
- not $\mathcal{M} \models_\sigma$ **false**
- $\mathcal{M} \models_\sigma (p \wedge q)$ iff $\mathcal{M} \models_\sigma p$ and $\mathcal{M} \models_\sigma q$.
- $\mathcal{M} \models_\sigma (p \rightarrow q)$ iff not $\mathcal{M} \models_\sigma p$ or $\mathcal{M} \models_\sigma q$.
- $\mathcal{M} \models_\sigma (\forall x : p)$ iff $\mathcal{M} \models_{\sigma[v/d]} p$ for all $d \in M$.
- $\mathcal{M} \models_\sigma (\exists x : p)$ iff $\mathcal{M} \models_{\sigma[v/d]} p$ for some $d \in M$.

We say $\mathcal{M} \models p$ if for all σ, $\mathcal{M} \models_\sigma p$. If p and q are formulas we use the shorthand $p \models q$ to mean that every model of p is a model of q. If $p \models q$ then we say that q is a *semantic consequence* of p.

B.2 Proof of Soundness Theorem

In this section we supply the details of the proof of the soundness of inference in ALL. The theorem was presented, without proof, in the main text in section 5.4.2.

Theorem 2 (Soundness of ALL) For any knowledge-base K, any path α allowed in K, and any fact f allowed in K:

1. $(\forall \theta \in \Theta : \theta \in sub(query(\alpha)(K)) : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta))$
2. $\mathcal{PC}(K) \models \mathcal{PC}(kb(query(\alpha)(K)))$
3. $(\mathcal{PC}(K) \wedge \mathcal{PC}(f)) \models \mathcal{PC}(kb(assert(f)(K)))$

We first prove a series of basic lemmas about $\mathcal{PC}$.

B.2.1 Basic Lemmas About $\mathcal{PC}$

Lemma 37 *If K_1 and K_2 are knowledge-bases then:*

$$\mathcal{PC}(K_1 \cup K_2) = \mathcal{PC}(K_1) \wedge \mathcal{PC}(K_2).$$

Proof: Immediate from definition of PC. ▮

Lemma 38 *For any predicate calculus sentence s, any list of (ALL) propositions α, and any substitutions θ and η such that $head(\alpha)\theta$ and $rest(\alpha)\theta\eta$ are ground:*

$$(s \models \mathcal{PC}(head(\alpha)\theta)) \wedge (s \models \mathcal{PC}(rest(\alpha)\theta\eta)) \Longrightarrow s \models \mathcal{PC}(\alpha\theta\eta).$$

Proof: First observe that since $head(\alpha)\theta$ is ground:

$$\mathcal{PC}(\alpha\theta\eta) = \mathcal{PC}(head(\alpha)\theta) \wedge \mathcal{PC}(rest(\alpha)\theta\eta).$$

The result then follows from the definition of $\models$. ▮

Then the 'soundness' of the *lookup* function:

Lemma 39 *For any knowledge-base K and any proposition q allowed in K:*

$$(\forall\theta \in \Theta : \theta \in lookup(q)(K) : \mathcal{PC}(K) \models \mathcal{PC}(q\theta)).$$

Proof: By the definition of *lookup* the above conclusion is equivalent to:

$(\forall\theta \in \Theta : (\exists f \in K :: \theta = mgru(q, f)) : \mathcal{PC}(K) \models \mathcal{PC}(q\theta)) \Longleftrightarrow$
{ By Predicate Calculus and since $\theta = mgru(q, f) \Longrightarrow q\theta = f\theta = f$. }
$(\forall\theta \in \Theta, f \in K : \theta = mgru(q, f) : \mathcal{PC}(K) \models \mathcal{PC}(f)) \Leftarrow$
{ Predicate Calculus }
$(\forall f \in K :: \mathcal{PC}(K) \models \mathcal{PC}(f))$

Which follows immediately from the definition of $\mathcal{PC}$. ▮

Finally, the soundness of *closure*:

Lemma 40 *For any knowledge-base K:*

$$\mathcal{PC}(K) \models \mathcal{PC}(closure(K)).$$

Proof: The key lemma is that for any if-added rule ρ allowed in K:

$$\mathcal{PC}(K) \wedge \mathcal{PC}(\rho) \models \mathcal{PC}(closure(K, \rho))$$

Which can be shown from the definitions. From here one inducts on n to show (for all n):

$$\mathcal{PC}(K) \models \mathcal{PC}(closure_n(K)).$$

From which the result follows (from the definitions and lemma 37). ▮

B.2.2 Proof of Soundness of ALL

As mentioned in the text, the proof of the soundness of ALL is not really difficult, but it involves keeping careful track of details as we work through the definition of $\mathcal{O}$.

> **Theorem 2 (Soundness of ALL)** For any knowledge-base K, any path α allowed in K, and any fact f allowed in K:
>
> 1. $(\forall \theta \in \Theta : \theta \in sub(query(\alpha)(K)) : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta))$
> 2. $\mathcal{PC}(K) \models \mathcal{PC}(kb(query(\alpha)(K)))$
> 3. $(\mathcal{PC}(K) \wedge \mathcal{PC}(f)) \models \mathcal{PC}(kb(assert(f)(K)))$

Proof: We first show that it suffices to prove that $query_n$ and $assert_n$ are consistent. That is, it suffices to show that for any $n \geq 0$, for any knowledge-base K, any path α allowed in K, any fact f allowed in K, and any $p \subset C \times R$:

1′. $(\forall \theta \in \Theta : \theta \in sub(query_n(\alpha)(K, p)) : \mathcal{PC}(K) \models \mathcal{PC}(\alpha\theta))$

2′. $\mathcal{PC}(K) \models \mathcal{PC}(kb(query_n(\alpha)(K, p)))$

3′. $(\mathcal{PC}(K) \wedge \mathcal{PC}(f)) \models \mathcal{PC}(kb(assert_n(f)(K, p)))$

Part 3 of the theorem follows from 3′ by lemma 37 (and the definition of $\models$). Parts 1 and 2 of the theorem follow by induction on the length of α (the only trick is that, in the induction step, one instantiates the induction hypothesis with $rest(\alpha)\eta$ for each $\eta \in sub(query(head(\alpha))(K))$).

To show equations 1′ - 3′. we induct on n.

Base Case: $n = 0$. 3′ follows easily from the definitions and lemma 40. To show 1′ and 2′ we induct on the length of α. If α primitive then 2′ is immediate from case 1 of the definition of $\mathcal{O}_n$. 1′ follows from case 1 of the definition of $\mathcal{O}_n$ and lemma 39. If α is not primitive then we induction on the length of α as in the proof of parts 1 and 2 of the main theorem.

Induction Step: We outline the argument for $1'$ and $2'$ (the argument for $3'$ is similar to that for $2'$). Again we induct on the length of α. First assume α primitive. Note that (from the definition of PC:

$$\rho \in R \Longrightarrow (\mathcal{PC}(K) \models \mathcal{PC}(\rho)).$$

Thus (by induction and the definitions):

$$\rho \in R \wedge \theta \in sub(query_{n-1}(Ant(\rho))(K)) \Longrightarrow (\mathcal{PC}(K) \models \mathcal{PC}(Conseq(\rho)\theta)).$$

Thus (by induction and with some symbol pushing using lemmas 37 and 40):

$$\mathcal{PC}(K) \models \mathcal{PC}(K')$$

From which $2'$ follows immediately, and $1'$ follows by lemma 39.

If α is not primitive then we again induction on the length of α as in the proof of parts 1 and 2 of the main theorem.

∎

B.3 Proofs of Lemmas for Completeness Results

In this section we prove the following lemmas used in the proofs of the completeness results in sections 5.5.3 and 5.5.4.

Lemma 11 For any knowledge-base K there exists a series of variable free paths Γ (allowed in K) such that $T_{\mathcal{LP}(K)}(K) \subset kb(query(\Gamma)(K))$.

Lemma 13 For any knowledge-base K, any primitive path q allowed in K, any $p \subset C \times R$:

$$ground(q) \cap T_{\mathcal{LP}(K\backslash_p)} \uparrow n \subset kb(query_n(q)(K,p)).$$

B.3.1 Preliminary Results

In the lemmas and proofs which follows we use the following notation: If $\alpha = q_1, \ldots, q_n$ is a path, θ a substitution, and $1 \leq i \leq n$ we use the shorthand:

$$\theta\backslash_{\alpha<i} = \theta\backslash_{vars(q_1,\ldots,q_{i-1})}.$$

That is, θ with its domain restricted to the variables appearing in the first $i-1$ propositions of α.

We first show a preliminary lemma which says (roughly) that if a query of every proposition in a path succeeds then a query of the path succeeds.

It turns out to be non-trivial to formalize the idea of a substitution making 'every proposition in a path true'. Consider a (non-empty) path α. It is not necessarily true that $\theta \in$

$sub(query(\alpha)(K)) \Rightarrow \{\{\}\} \in sub(query(\alpha\theta)(K))$[1]; The query of α constructs the substitution θ, but each proposition in α is queried only under the substitutions generated by the propositions appearing before it in α (see equation 5.23). This complication is reflected in the lemma by the restriction of θ to $\theta\backslash_{\alpha<i}$.

Lemma 41 *If $\alpha = q_1, \ldots, q_n$ is a (non-empty) path allowed in a knowledge-base K, $p \subset C \times R$, $n \geq 0$ and*

$$(\exists\theta \in \Theta :: (\forall i : 1 \leq i \leq n : q_i\theta \in kb(query_n(q_i\theta\backslash_{\alpha<i})(K,p))))$$

then there is some substitution $\eta \in sub(query_n(\alpha)(K,p))$ such that η is more general than θ.

Proof: By induction on the length of α. If $\alpha = q_1$ then the result follows from the definitions of *lookup* and *mgru*. If α is of length greater than one then let $\alpha' = q_2, \ldots, q_n$. Further, let $\theta' = \theta\backslash_{vars(\alpha')}$. By induction, there is some substitution $\eta' \in sub(query_n(\alpha'\theta\backslash_{vars(q_1)})(K,p))$ such that η' is more general than θ'. Similarly, there is some substitution $\eta_1 \in sub(query_n(q_1)(K,p))$ such that η_1 is more general than θ. Further, by lemma 8 part 1, η_1 is a ground substitution binding all and only the variables in q_1, hence $\eta_1 = \theta\backslash_{vars(q_1)}$. From here the result follows by the definition of $\mathcal{O}_n$, and the observation that if $\eta_1 = \theta\backslash_{vars(q_1)}$ and η_2 is more general than $\theta\backslash_{vars(\alpha')}$ then $\eta_1 \circ \eta_2$ is more general than θ. ▮

Finally, we have a lemma relating ALL to logic programming. Intuitively, it says that the if-needed rules which 'should' fire actually do fire. It is fairly technical (as is its proof).

Intuitively, the substitution η below is the most general unifier of the *key* of the rule applied, with the proposition queried. Further, ζ is a substitution which makes the antecedent of the rule 'true' in the knowledge-base (see discussion preceding lemma 41).

Recall that (as defined in section 5.3.2):

$$ground(q) = \{f \mid f \text{ is a fact} \wedge (\exists\theta : \theta \text{ is a ground substitution: } f = q\theta)\}.$$

Lemma 42 *Assume K is a knowledge-base, $a \leftarrow \alpha \in Nr$, $\alpha = b_1, \ldots, b_m$, $p \subset C \times R$, q is a primitive proposition allowed in K such that $q \in p$, $n > 0$, $f \in ground(q)$, and κ is a renaming of variables in Nr to variables not in q or K:*

$$(\exists\eta \in \Theta, \zeta \in \Theta : \eta = mgru(a\kappa, q) \wedge f = q\eta\zeta \tag{B.1}$$
$$: (\forall i : 1 \leq i \leq m : b_i\kappa\eta\zeta \in kb(query_{n-1}(b_i\kappa\eta\zeta\backslash_{\alpha<i}))(K,p))))$$
$$\Rightarrow$$
$$f \in kb(query_n(q)(K,p)). \tag{B.2}$$

Proof: Proving this is a matter of working through case 2 of the definition of $\mathcal{O}_n$.

Consider the set R in the calculation of $query_n(q)(K,p)$ (equation 5.15). The preconditions of the lemma imply $(a \leftarrow \alpha)\kappa\eta \in R$. By definition of if-needed rules $\alpha\kappa\eta$ is a (non-empty) path.

[1] Counter-examples can be constructed but tend to be fairly complex. The basic idea is that a query of a proposition with variables in it may cause rules to apply which might not match an instantiation of the proposition.

So by lemma 41 there is some $\theta \in sub(query_{n-1}(\alpha\kappa\eta)(K,p))$ such that θ is more general than ζ. Intuitively, this means that the antecedent of $(a \leftarrow \alpha)\kappa\eta$ 'succeeds'. A few steps of symbol pushing shows that, in fact, $f = a\kappa\eta\theta$.

Intuitively we are now done since f is the consequent of a rule whose antecedent succeeds. Formally the result is shown from the definition of $\mathcal{O}_n$ (using lemma 10 and lemma 5 part 3). $\blacksquare$

Finally, a preliminary lemma for the Partitional Completeness lemma. A primitive path which is not in $p \subset C \times R$ cannot be unified with the rules in $S\backslash_p$:

Lemma 43 *For any knowledge-base K, any $p \subset C \times R$, any set S of rules allowed in K, any primitive path $q \notin p$, and any rule $\rho \in S\backslash_p$:*

$$(\forall \theta \in \Theta :: q\theta \neq Key(\rho)\theta).$$

Proof: Assume $Key(\rho) = r_1(t_1, \ldots, t_n)$ and $q = r_2(s_1, \ldots, s_n)$. $\langle t_1, r_1\rangle \in p$ (by definition of $S\backslash_p$) and $\langle s_1, r_2\rangle \notin p$ (since $q \notin p$). Thus $t_1 \neq s_1$ or $r_1 \neq r_2$. In either case $(\forall \theta \in \Theta :: q\theta \neq Key(\rho)\theta)$. $\blacksquare$

B.3.2 Proofs of Lemmas

> **Lemma 11** For any knowledge-base K there exists a series of variable free paths Γ, (allowed in K) such that $T_{\mathcal{LP}(K)}(K) \subset kb(query(\Gamma)(K))$.

Proof: note that the statement of this lemma uses the identification of knowledge-bases with Herbrand models discussed in section 5.5.1.

Intuitively, this lemma says that there is a series of paths Γ such that querying Γ has at least as much effect on the knowledge-base as the immediate consequence operator. That is, querying Γ should cause all if-needed rules in the knowledge-base which can fire, to fire. Thus, let Γ be the list of all ground instantiations of the keys of all if-needed rules in K (trivially this is a series of paths allowed in K). It remains only to show that querying Γ really does cause all rules to fire. Take an arbitrary fact $f \in T_{\mathcal{LP}(K)}(K)$. By definition of $T_{\mathcal{LP}(K)}$ there exists a rule $a \leftarrow b_1, \ldots, b_n \in \mathcal{LP}(K)$ and a substitution $\theta \in \Theta$ such that $f = a\theta$ and $K \models (b_1 \wedge \ldots \wedge b_n)\theta$. Recall from section 5.5.1 that for a fact f, $K \models f \iff f \in K$. Thus for all i $(1 \leq i \leq n)$, $b_i\theta \in K$. By definition of the most general unifier there exists substitutions η and ζ such that $\eta = mgru(f, a)$ and $\theta = \eta \circ \zeta$. Combining all this together, for an arbitrary fact $f \in T_{\mathcal{LP}(K)}(K)$:

$$(\exists \eta \in \Theta, \zeta \in \Theta, a \leftarrow b_1, \ldots, b_n \in \mathcal{LP}(K) :: \eta = mgru(f, a) \wedge (\forall i : 1 \leq i \leq n : b_i(\eta \circ \zeta) \in K)). \quad \text{(B.3)}$$

To prove the lemma above it now suffices to show that B.3 implies $f \in kb(query(\Gamma)(K))$. We prove this in three cases:

Case 1: $a \leftarrow b_1, \ldots, b_n$ is $\mathcal{LP}$ of some fact in K.

Case 2: $a \leftarrow b_1, \ldots, b_n$ is $\mathcal{LP}$ of some if-needed rule in K.

Case 3: $a \leftarrow b_1, \ldots, b_n$ is $\mathcal{LP}$ of some if-added rule in K.

In case 1, f is a fact in K and the result is immediate since ALL operations cannot cause knowledge-bases to shrink (by lemma 9 part 2).

For case 2, observe first that since Γ is a list of all ground instantiations of the keys of rules in K, B.3 implies f is a member of the list Γ. Thus (since, by lemma 9 parts 2 and 3, ALL operations cannot cause knowledge-bases to shrink and are monotonic) it suffices to show that B.3 implies $f \in kb(query(f)(K))$. But to prove this it suffices to show that B.3 implies $f \in kb(query_1(f)(K, par_K(f)))$ (since $\mathcal{O}$ is the union of $\mathcal{O}_n$ for all $n > 0$). This follows from lemma 42.

For case 3 note that by lemma 7, $kb(query(\Gamma)(K))$ is closed. By lemma 4, a closed knowledge-base contains all consequents of its if-added rules, thus $f \in kb(query(\Gamma)(K))$. ∎

Lemma 13 For any knowledge-base K, any primitive path q allowed in K, any $p \subset C \times R$:

$$ground(q) \cap T_{\mathcal{LP}(K\backslash_p)} \uparrow n \subset kb(query_n(q)(K, p)).$$

Proof Consider any $f \in ground(q) \cap T_{\mathcal{LP}(K\backslash_p)} \uparrow n$. We prove the lemma in two cases.

Case 1: $f \notin p$. Since $f \in T_{\mathcal{LP}(K\backslash_p)} \uparrow n$ there must be some $\rho \in \mathcal{LP}(K\backslash_p)$ and some θ such that $Key(\rho)\theta = f\theta$. If ρ were $\mathcal{LP}$ of some if-needed rule in $K\backslash_p$ then by lemma 43 there could be no such θ. Hence, ρ is $\mathcal{LP}$ of some fact in $K\backslash_p$. $query_n(q)$ cannot cause the knowledge-base to shrink (by lemma 8 part 2), thus $f \in kb(query_n(q)(K, p))$.

Case 2: $f \in p$. In this case we induct on n. The base case is trivial since $T_{\mathcal{LP}(K\backslash_p)} \uparrow 0 = \emptyset$. For the inductive case take any $f \in (ground(q) \cap T_{\mathcal{LP}(K\backslash_p)} \uparrow n)$. By definition of $T_P \uparrow n$:

$$(\exists \theta \in \Theta, a \leftarrow b_1, \ldots, b_m \in \mathcal{LP}(K\backslash_p) : f = a\theta \\ : T_{\mathcal{LP}(K\backslash_p)} \uparrow (n-1) \models (b_1 \wedge \ldots \wedge b_m)\theta). \tag{B.4}$$

If $a \leftarrow b_1, \ldots, b_m$ is $\mathcal{LP}$ of a fact in $K\backslash_p$ then $f \in kb(query_n(q)(K, p))$ by lemma 8 part 2. Thus assume $a \leftarrow b_1, \ldots, b_m$ is $\mathcal{LP}$ of an if-needed rule in $K\backslash_p$. From here we would like to apply lemma 42. A fair amount of technical manipulation of substitutions is required but is left to the reader. Essentially, we refine B.4 by characterizing θ as the composition of a renaming κ (from variables in Nr to variables not in K or q), $\eta = mgru(a\kappa, q)$, and a ground substitution ζ. This gives us:

$$(\exists \eta \in \Theta, \zeta \in \Theta, a \leftarrow b_1, \ldots, b_m \in Nr : \eta = mgru(a\kappa, q) \wedge f = q\eta\zeta \\ : T_{\mathcal{LP}(K\backslash_p)} \uparrow (n-1) \models (b_1 \wedge \ldots \wedge b_m)\kappa\eta\zeta). \tag{B.5}$$

From which the result follows by induction and lemma 42.

∎

B.4 Proofs of Lemmas for Complexity Result

In this section we prove the two lemmas used in section 7:

Lemma 33 Computation Lemma Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $n > 0$:

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O})))$$
$$\Longrightarrow$$
$$\mathcal{O}(K) = \mathcal{O}_n(K, par_K(\mathcal{O})).$$

Lemma 34 $closure_{f\&r}$ **lemma** Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $\mathcal{O}' \in ops(\mathcal{O})$:

1. For any rule ρ, if $\rho \in \mathcal{O}'(K)$, but $\rho \notin K$, then $\rho \in closure_{f\&r}(\mathcal{O})$.
2. For any fact f, if $f \in \mathcal{O}'(K)$, but $f \notin K$, then $f \in closure_{f\&r}(\mathcal{O})$.

B.4.1 Basic Definitions and Lemmas

We now develop the full definition of *ops* and prove a series of lemmas. First we have some utility functions. For these definitions, assume that all t_i are terms. Recall from the main text, that for any knowledge-base K and any $p \subset C \times RK$:

$$rules_K(p) = Ar \cup Nr\backslash_p$$

(see section 5.5.4 for the definition of a set of rules restricted to p). Intuitively, $rules_K(p)$ returns all the rules which can ever apply in the calculation of an operation in p. Next we have the idea of *the set of all instantiations of a path* (or rule). If α is a path (or rule) and fr a set of frames then let:

$$inst(\alpha, fr) = \{\alpha\theta \mid \theta\, ground \wedge range(\theta) \subset fr\}.$$

If $\mathcal{O} = query(\alpha)$ then (for any set of frames fr):

$$inst(\mathcal{O}, fr) = \{query(\alpha') \mid \alpha' \in inst(\alpha, fr)\}.$$

Similarly, if $\mathcal{O} = assert(\alpha)$ (where α of length 1) then:

$$inst(\mathcal{O}, fr) = \{assert(f) \mid f \in inst(\alpha, fr) \wedge f\, ground\}.$$

note that *inst* maps paths to sets of paths and operations to sets of operations.[2] We can extend it to also map sets to sets. If S is a set of operations then:

$$inst(S, fr) = (\bigcup \mathcal{O} : \mathcal{O} \in S : inst(\mathcal{O}, fr).$$

[2]Technical note: If α is a primitive path then $assert(\alpha)$ is not necessarily an operation (since only facts may be asserted), but all elements of $inst(assert(\alpha), fr)$ are operations (since all such elements are necessarily ground).

For a primitive path $q = r(c, t_1, \ldots, t_n)$ let $fs(q)$ (the 'frame-slot' or q) be $\langle c, r\rangle$. For a set S of primitive paths, let:

$$fs(S) = \{fs(q) \mid q \in S\}.$$

Finally, for any knowledge-base K and $p \subset C \times R$, $frames\_in_K(p)$ returns the set of all frames which occur in frame-slots in p:

$$frames\_in_K(p) = \{c \mid (\exists \langle c_0, r_0\rangle \in p :: r_0(c_0, t_1, \ldots, t_m, c, t_{m+1}, \ldots, t_n) \in F)\}.$$

We now formalize the idea of "*the portion of the knowledge-base potentially accessible during an operation*". We do this with the mutually recursive functions $ireach_{K,n}$ and $ibranches_{K,n}$, and the function $frames_{K,n}$. These functions will be used to prove the basic lemmas and to define the functions $reach$ and $change$ discussed in the text.

Consider any primitive operation $\mathcal{O} = query(q)$ or $\mathcal{O} = assert(q)$ (for some primitive path q). Now consider any primitive proposition q' which is queried or asserted in the process of computing $\mathcal{O}$. A query (or assertion) of q' can chain to queries of other primitive paths through the access-paths in rules. The union of all the primitive paths potentially accessible to rules in $rules_K(p)$ (unified with q' and with backchaining to depth n) forms $ireach_{K,n}(q', q, p)$. Now consider a single access path α which is the antecedent of some rule for q'. $ibranches_{K,n}(\alpha, q, p)$ computes the set of all primitive paths potentially queried while following the path α (with backchaining allowed to depth n). Now consider a single variable x, in $head(\alpha)$. It turns out that (in the worst case) x may end up bound to any frame in the frame-slots of the primitive paths in $ireach_{K,n}(q, q, p)$, to any frame in q, or to any frame appearing explicitly in a rule in $rules_K(p)$. We define this set of frames to be $frames_{K,n}(q, p)$.

In following definitions assume that $q' = r(c, t_1, \ldots, t_n)$ and q are primitive propositions allowed in a knowledge-base K, α is a (non-empty) path allowed in K, and $p \subset C \times R$. Consider any rule $\rho \in rules_K(p)$. By definition of $rules_K$, there exists some ρ' such that either $\rho' \in Ar$ and $\rho = \rho'$ or $\rho' \in Nr$ and ρ is an instantiation of ρ'. In either case, let $base\_rule(\rho) = \rho'$. Let κ be a renaming from variables in the rules in Nr to variables not in K or q' (see footnote in definition of $\mathcal{O}_n$ case 2).

We formally define $ireach_{K,n}(q', q, p)$ to include the primitive paths accessible while querying the antecedents and asserting the consequents of the rules for q'. Since we do not know in advance what the antecedents of the rules will return, we include the reach of the assertions of the consequents under any possible substitution mapping to frames in $frames_{K,n-1}(q, p)$. The rules for q' are the if-added rules which unify with q' and the rules in $Nr\backslash_{p\cap\{c,r\}}$ whose base rules unify with q' (note that we carefully apply κ to the if-needed rules so that the rules in $ireach$ will have the same variable names as those in the definition of $\mathcal{O}$ — this simplifies the proofs below):

$$ireach_{K,0}(q', q, p) = \{q'\} \tag{B.6}$$

$$\begin{aligned} ireach_{K,n}(q', q, p) = {} & \{q'\} \cup \\ & (\textstyle\bigcup \rho \in \{\sigma\theta \mid ((\sigma \in Ar \wedge \theta = mgru(key(\sigma), q')) \vee \\ & \qquad (\sigma \in Nr\backslash_{p\cap\{c,r\}} \wedge \theta = \kappa \circ mgru(key(base\_rule(\sigma))\kappa, q')))\} \\ & \quad :: ibranches_{K,n-1}(Ant(\rho), q, p) \cup \end{aligned} \tag{B.7}$$

$$(\bigcup q_c \in inst(Conseq(\rho), frames_{K,n-1}(q,p)) \\ :: ireach_{K,n-1}(q_c, q, p)))$$

$$\begin{aligned} ibranches_{K,n}(nil, q, p) &= \emptyset && \text{(B.8)} \\ ibranches_{K,n}(\alpha, q, p) &= ireach_{K,n}(head(\alpha), q, p) \cup && \text{(B.9)} \\ & \quad (\bigcup \theta \in \Theta : (\theta : vars(head(\alpha)) \longrightarrow frames_{K,n}(q,p)) \\ & \qquad : ibranches_{K,n}(rest(\alpha)\theta, q, p)) \end{aligned}$$

$$\begin{aligned} frames_{K,n}(q,p) &= frames\_in_K(reach_{K,n}(q,q,p)) \cup constants(q) && \text{(B.10)} \\ & \quad \cup \{c \mid (\exists \rho \in rules_K(p) :: c \in constants(base\_rule(\rho)))\}. \end{aligned}$$

We abbreviate $ireach_{K,n}(q,q,p)$ and $ibranches_{K,n}(q,q,p)$ to $ireach_{K,n}(q,p)$ and $ibranches_{K,n}(q,p)$, respectively.

We can also compute the set of frame-slots which an operation may change. Essentially, this is a matter of binding the variables in all rules in $rules_K(p)$ to all the frames in $frames_{K,n}(q,p)$, and collecting the frame-slots referenced by the consequent of the rules. If q' and q are primitive paths allowed in knowledge-base K, and $p \in C \times R$, then:

$$\begin{aligned} ichange_{K,n}(q', q, p) &= \{Conseq(\rho)\theta \mid \rho \in rules_K(p \cap reach_{K,n}(q', q, p)) \wedge \\ & \qquad (\theta : vars(\rho) \longrightarrow frames_{K,n}(q,p))\} \\ & \quad \cup \{q\} \; if \; q \; ground. && \text{(B.11)} \end{aligned}$$

Again, we abbreviate $ichange_{K,n}(q,q,p)$ to $ichange_{K,n}(q,p)$.

Consider any operation $\mathcal{O}$ such that $\mathcal{O} = query(q)$, or q a fact and $\mathcal{O} = assert(q)$. The operations which $\mathcal{O}$ depends on are the following:

$$\begin{aligned} iqueries_{K,n}(\mathcal{O}) &= \{query(q') \mid q' \in ireach_{K,n}(q, par_K(\mathcal{O})\} && \text{(B.12)} \\ iassertions_{K,n}(\mathcal{O}) &= \{assert(f) \mid f \in ichange_{K,n}(q, par_K(\mathcal{O}) \cap && \text{(B.13)} \\ & \qquad ireach_{K,n}(q, par_K(\mathcal{O})\} \\ iops_{K,n}(\mathcal{O}) &= iqueries_{K,n}(\mathcal{O}) \cup iassertions_{K,n}(\mathcal{O}). && \text{(B.14)} \end{aligned}$$

In all cases which follow we will be working with a single knowledge-base and will simply write: $rules(\ldots)$, $ireach_n(\ldots)$, $ibranches_n(\ldots)$, $frames_n(\ldots)$, $ichange_n(\ldots)$, $iqueries_n(\ldots)$, $iassertions_n(\ldots)$, and $iops_n(\ldots)$. Further, as defined in the text for *reach*, *change*, and *ops*:

$$\begin{aligned} ireach(q,p) &= (\bigcup n : n > 0 : ireach_n(q,p)) && \text{(B.15)} \\ ichange(q,p) &= (\bigcup n : n > 0 : ichange_n(q,p)) && \text{(B.16)} \\ iops(\mathcal{O}) &= (\bigcup n : n > 0 : iops_n(\mathcal{O})). && \text{(B.17)} \end{aligned}$$

Unfortunately, *ireach* (and thus *iops*) is not very useful in establishing time bounds because it is often infinite ! Fortunately, *ireach* can be infinite only because it can contain infinite collections of primitive paths which differ only in variable names — *i.e.*, infinite collections of primitive paths which are variants of each other (these infinite collections arise because of the variable renaming step in the application of if-needed rules). Further, the results of queries of paths which differ only in variable names are essentially identical. The precise relationship is given in the lemma below. Recall that for V a set of variables and θ a substitution, $\theta\backslash_V$ denotes the substitution θ with its domain restricted to only the variables in V. For a set of substitutions Λ, let $\Lambda\backslash_V = \{\theta\backslash_V \mid \theta \in \Lambda\}$.

Lemma 44 (Rename Lemma) *Assume α_1 and α_2 are paths allowed in a knowledge-base K. If α_1 and α_2 are variants such that, for some renaming κ, $\alpha_1 = \alpha_2\kappa$ then:*

1. $kb(query(\alpha_1)(K)) = kb(query(\alpha_2)(K))$
2. $sub(query(\alpha_2)(K)) = (\kappa \circ sub(query(\alpha_1)(K)))\backslash_{vars(\alpha_2)}$

Proof: While the result is intuitive, a careful proof requires a significant amount of formal manipulation of substitutions. Details are left to the reader. ∎

We can also bound the frames which can occur in $ireach_n$:

Lemma 45 *Consider any operation $\mathcal{O}$, allowed in a knowledge-base K, such that $\mathcal{O} = query(q)$ or q a fact and $\mathcal{O} = assert(q)$. If (for some $n \geq 0$) $q' \in ireach_n(q, par_K(\mathcal{O}))$ then:*

$$(\exists l : l \geq 0 : constants(q') \subset frames_l(q, par_K(\mathcal{O}))).$$

Proof: By induction on n. The trick is to generalize the hypothesis to state that for any $\mathcal{O}$, q, and n as in the lemma, any $m \geq 0$, and any q' such that $constants(q') \subset frames_m(q, par_K(\mathcal{O}))$:

$$(\exists l : l \geq 0 : (\forall q'' \in ireach_n(q', q, par_K(\mathcal{O})) :: constants(q'') \subset frames_l(q, par_K(\mathcal{O}))))$$

∎

As discussed in the text, we would like to characterize the complexity of operations in terms of the frame-slots that can be accessed and/or changed by an operation. To this end let:

$$reach_n(q, p) = fs(ireach_n(q, p)) \tag{B.18}$$
$$change_n(q, p) = = fs(ichange_n(q, p)) \tag{B.19}$$

We can now define the set of all queries of frame-slots in *reach* and assertions into frame-slots in *change*. Assuming that $x_1, \ldots, x_m$ are free variables, and that $\mathcal{O} = query(q)$ or q a fact and

$\mathcal{O} = assert(q)$, let:

$$
\begin{aligned}
queries_n(\mathcal{O}) &= inst(\{query(r(c, x_1, \ldots, x_m)) \mid \langle c, r\rangle \in reach_n(q, par_K(\mathcal{O}))\}, \\
&\qquad frames_n(q, par_K(\mathcal{O}))) && \text{(B.20)} \\
assertions_n(\mathcal{O}) &= inst(\{assert(r(c, x_1, \ldots, x_m)) \mid \langle c, r\rangle \in change_n(q, par_K(\mathcal{O})) \cap \\
&\qquad\qquad reach_n(q, par_K(\mathcal{O}))\}, \\
&\qquad frames_{K,n}(q, par_K(\mathcal{O}))) && \text{(B.21)}
\end{aligned}
$$

From lemma 45 it follows that (up to variable names):

$$
\begin{aligned}
iqueries_n(\mathcal{O}) &\subset queries_n(\mathcal{O}) && \text{(B.22)} \\
iassertions_n(\mathcal{O}) &\subset assertions_n(\mathcal{O}) && \text{(B.23)}
\end{aligned}
$$

However, the converse is not true since some instantiations in $queries_n$ (or $assertions_n$) may not occur in $iquery_n$ (or $iassertions_n$). To force the converse to be true, we define ops_n to be the union of $queries_n$ and $assertions_n$ restricted to operations which are variants of operations in $iops_n$:

$$ops_n(\mathcal{O}) = \{\mathcal{O}' \mid \mathcal{O}' \in queries_n(\mathcal{O}) \cup assertions_n(\mathcal{O}) \wedge (\exists \mathcal{O}'' \in iops_n(\mathcal{O}) :: \mathcal{O}' \textit{ a variant of } \mathcal{O}'')\}$$

In most of the lemmas below we work with $ireach_n$ and $iops_n$, and we then 'convert' up to $reach_n$ and ops_n to prove the main lemmas from the text (this two stepped approach is necessary since, as will be seen, some low level results hold for *ireach* but not for *reach*).

Figure B.1 shows what *reach*, *change*, and *ops* work out to in a slightly more complex case than the cases shown in the text (one can also verify that in this case *ireach* is infinite).

B.4.2 Preliminary Lemmas

A key lemma for *ireach* is that, for a query of q, the set of if-needed rules in the definition of *ireach* includes the rules in R in the definition of *query* (see equation 5.15):

Lemma 46 *Assume K is a knowledge-base, and $p \subset C \times R$. For any operation $query(q)$ such that $q = r(c, t_1, \ldots, t_m)$, $q \in p$, and q allowed in K, if a rule, ρ, is in the set R in the calculation of $query_n(q)(K, p)$ (see equation 5.15) and κ is a renaming from variables in Nr to variables not in K or q, then:*

$$
\begin{aligned}
(\exists \rho' \in Nr\backslash_{\{\langle c,r\rangle\}}, \theta \in \Theta &: \theta = mgru(key(base\_rule(\rho'))\kappa, q) \\
&: \rho = \rho'\kappa\theta) && \text{(B.24)}
\end{aligned}
$$

Proof: Consider an arbitrary if-needed rule $\rho \in R$. By definition of R, there is some if-needed rule $\sigma \in Nr$ such that for $\theta = mgru(key(\sigma)\kappa, q)$:

$$\rho = \sigma\kappa\theta.$$

Further, we can choose ρ' such that $\sigma = base\_rule(\rho')$ and, for some ground substitution ζ, $\rho' = \sigma\zeta$.

Let K be as follows:

$$\begin{aligned}
C &= \{c_1, c_2, c_3, c_4\} \\
R &= \{r\} \\
Nr &= \{r(x,z) \leftarrow r(x,y), r(y,z)\} \\
Ar &= \{\} \\
F &= \{r(c_1,c_2), r(c_2,c_3), r(c_3,c_4)\} \\
P &= \{\{\langle c_1, r\rangle, \langle c_2, r\rangle, \langle c_3, r\rangle, \langle c_4, r\rangle\}\}
\end{aligned}$$

Consider $\mathcal{O} = query(r(c_1, x))$ (where x is a variable). Let $q = r(c_1, x)$. We can compute as follows:

$$\begin{aligned}
reach_0(q,p) &= \{\langle c_1, r\rangle\} \\
reach_1(q,p) &= \{\langle c_1, r\rangle, \langle c_2, r\rangle\} \\
reach_2(q,p) &= \{\langle c_1, r\rangle, \langle c_2, r\rangle, \langle c_3, r\rangle, \langle c_4, r\rangle\} \\
reach_n(q,p) &= \{\langle c_1, r\rangle, \langle c_2, r\rangle, \langle c_3, r\rangle, \langle c_4, r\rangle\} \\
reach(q,p) &= \{\langle c_1, r\rangle, \langle c_2, r\rangle, \langle c_3, r\rangle, \langle c_4, r\rangle\}
\end{aligned}$$

Thus:

$$\begin{aligned}
change(q,p) &= \{\langle c_1, r\rangle, \langle c_2, r\rangle, \langle c_3, r\rangle, \langle c_4, r\rangle\} \\
ops(\mathcal{O}) &= \{query(r(c_1,x)), query(r(c_2,x)), query(r(c_3,x)), query(r(c_4,x)), \\
&\quad query(r(c_1,c_1)), query(r(c_2,c_1)), query(r(c_3,c_1)), query(r(c_4,c_1)), \\
&\quad query(r(c_1,c_2)), query(r(c_2,c_2)), query(r(c_3,c_2)), query(r(c_4,c_2)), \\
&\quad query(r(c_1,c_3)), query(r(c_2,c_3)), query(r(c_3,c_3)), query(r(c_4,c_3)), \\
&\quad query(r(c_1,c_4)), query(r(c_2,c_4)), query(r(c_3,c_4)), query(r(c_4,c_4)), \\
&\quad assert(r(c_1,c_1)), assert(r(c_2,c_1)), assert(r(c_3,c_1)), assert(r(c_4,c_1)), \\
&\quad assert(r(c_1,c_2)), assert(r(c_2,c_2)), assert(r(c_3,c_2)), assert(r(c_4,c_2)), \\
&\quad assert(r(c_1,c_3)), assert(r(c_2,c_3)), assert(r(c_3,c_3)), assert(r(c_4,c_3)), \\
&\quad assert(r(c_1,c_4)), assert(r(c_2,c_4)), assert(r(c_3,c_4)), assert(r(c_4,c_4))\}
\end{aligned}$$

Figure B.1: The accessible frame-slots and dependent operations for a more complex query.

So need only show $\rho = \rho'\kappa\theta$, which follows from $\sigma\kappa\theta = \sigma\zeta\kappa\theta$. Intuitively, this equation says that ζ, which maps from a rule in Nr to a rule in $Nr\backslash_{\{\langle c,r\rangle\}}$, does not 'do anything' to σ that θ does not also do (in unifying $key(\sigma)\kappa$ with q). Formally, it can be shown with some symbol pushing. ∎

One basic fact about *ireach*, *ibranches* and *frames* is that they are monotonic in their subscripts:

Lemma 47 *For any primitive paths q and q' allowed in a knowledge-base K, any path α allowed in K, any $p \subset C \times R$, any $n \geq m \geq 0$:*

1. $ireach_n(q', q, p) \supset ireach_m(q', q, p)$
2. $ibranches_n(\alpha, q, p) \supset ireach_m(\alpha, q, p)$
3. $frames_n(q, p) \supset frames_m(q, p)$

Proof: By induction on n. ∎

A key property of *ireach* is a kind of idempotence (this lemma does not seem to hold for *reach*, and is one of the main reasons for working with *ireach* instead):

Lemma 48 (Reach Lemma) *For any primitive path q allowed in a knowledge-base K, any $p \subset C \times R$, any $n \geq 0$, any $m \geq 0$, and any $q' \in ireach_n(q, p)$:*

$$ireach_m(q', q, p) \subset ireach_{n+m}(q, p).$$

Proof: The hard part of this proof is that additional notation is needed to state the induction hypothesis. Consider the set of frame-slots accessed in the evaluation of the antecedent of a rule for a query. This set includes the frame-slots accessed directly by the antecedent of the rule, and the frame-slots accessed while backchaining on the predicates in the antecedent of the rule. We define sr_n ("sub-reach") to include just the frame-slots access directly by the antecedent of the rule. In a sense this is just the "top level" of the tree of accessible frame-slots (though it is unequal to $ireach_1$ since the path branches on all values that can be found by backchaining to depth n). We then define $(sr_n)^m$ to return the mth level of this tree.

Formally, for any knowledge-base K, any primitive operations q and $q' = r(c, t_1, \ldots, t_m)$ allowed in K, any path α allowed in K, any $p \in C \times R$, any $m > 0$, and any $n > 0$:

$$\begin{aligned}
sr_0(q', q, p) &= \{q'\} \\
sr_n(q', q, p) &= \{q'\} \cup (\bigcup \rho \in \{\sigma\theta \mid ((\sigma \in Ar \wedge \theta = mgru(key(\sigma), q')) \vee \\
&\qquad (\sigma \in Nr\backslash_{p\cap\{c,r\}} \wedge \theta = \kappa \circ mgru(key(base_rule(\sigma))\kappa, q')))\} \\
&\qquad :: srb_{n-1}(Ant(\rho), q, p) \cup
\end{aligned}$$

$$inst(Conseq(\rho), frames_{n-1}(q,p)))$$

$$\begin{aligned}
srb_n(nil,q,p) &= \emptyset \\
srb_n(\alpha,q,p) &= \{head(\alpha)\} \cup \\
&\quad (\bigcup \theta \in \Theta : (\theta : vars(head(\alpha)) \longrightarrow frames_n(q,p)) \\
&\qquad : srb_n(rest(\alpha)\theta, q, p))
\end{aligned}$$

$$\begin{aligned}
(sr_n)^0(q',q,p) &= q' \\
(sr_n)^m(q',q,p) &= (\bigcup q'' \in sr_n(q',q,p) :: (sr_{n-1})^{m-1}(q'',q,p))
\end{aligned}$$

We then have two claims. The first claim says, intuitively, that the tree of frame-slots in $ireach_n$ can be broken into the top m levels (the first component of the right-hand-side), and the rest of the tree (the second component):

Claim 1: For any knowledge-base K, any primitive paths q and $q' = r(c, t_1, \ldots, t_k)$ allowed in K, any $p \subset C \times R$, any $n > 0$, and any $m \leq n$:

$$\begin{aligned}
ireach_n(q',q,p) &= (\bigcup l : 0 \leq l < m : (sr_n)^l(q',q,p) \\
&\quad \cup (\bigcup q'' \in (sr_n)^m(q',q,p) :: ireach_{n-m}(q'',q,p))
\end{aligned}$$

This claim is proven by induction on n and then on m.

The second claim says that increasing the depth of backchaining can only increase the size of the levels of the tree:

Claim 2: For any knowledge-base K, any primitive paths q and q' allowed in K, any $p \subset C \times R$, any $m \geq 0$, $n_2 > 0$, and any $n_1 \geq n_2$:

$$(sr_{n_1})^m(q',q,p) \supset (sr_{n_2})^m(q',q,p).$$

Which follows from the definitions and lemma 47.

The result follows from these claims. From claim 1 one can also show that $q' \in ireach_n(q,p)$ implies "anything in the tree must be in some level of the tree":

$$(\exists l : 0 \leq l \leq n : q' \in (sr_n)^l(q,q,p)).$$

Using both claims one can then show "the ireach starting from any node in the tree is in the tree":

$$ireach_{n+m}(q,p) \supset (\bigcup q'' \in (sr_n)^l(q,q,p) :: ireach_m(q'',q,p)).$$

Which implies $ireach_m(q',q,p) \subset ireach_{n+m}(q,p)$. ∎

Corollary 2 (Change Lemma) *For any primitive path q allowed in a knowledge-base K, any $p \subset C \times R$, any $n \geq 0$, any $m \geq 0$, and any $q' \in ireach_n(q,p)$:*

$$ichange_m(q',q,p) \subset ichange_{n+m}(q,p).$$

Proof: The result follows from lemma 48 and the definition of *ichange*. ▮

Then a technical lemma letting us rewrite the definitions of *ibranches* and *query*:

Lemma 49 *For any path $\alpha = append(\alpha_1, \alpha_2)$ allowed in a knowledge-base K, any primitive path q allowed in K, any $n > 0$, and any $p \subset C \times R$:*

$$sub(query_n(\alpha)(K,p)) = (\bigcup \theta \in sub(query_n(\alpha_1)(K,p)) :: \theta \circ sub(query_n(\alpha_2\theta)(K,p))) \quad (B.25)$$

$$ibranches_n(\alpha,q,p) = ibranches_n(\alpha_1,q,p) \cup (\bigcup \theta \in \Theta : (\theta : vars(\alpha_1) \longrightarrow frames_n(q,p)) : ibranches_n(\alpha_2\theta,q,p)) \quad (B.26)$$

Proof: By induction on the length of α_1. ▮

Then we need to bound the set of facts and rules which can be added by the *closure* operation. We call the bound on the set of facts $closure\_f_n$. It will simply be the intersection of $ichange_n$ and $ireach_n$. The rules are somewhat more complex. Recall that *closure* was defined iteratively — *closure* is the union over all n of $closure_n$, and $closure_n$ is the result of closing the knowledge-base produced by $closure_{n-1}$ over all the (forward chaining) rules in that knowledge-base. In a similar way we define the set $closure\_r_n$ which bounds the set of if-added rules that can be added. For these definitions, assume $\mathcal{O}$ is a primitive operation allowed in K, and that $\mathcal{O} = query(q)$, or q ground and $\mathcal{O} = assert(q)$:

$$closure\_f_n(\mathcal{O}) = ireach_n(q, par_K(\mathcal{O})) \cap ichange_n(q, par_K(\mathcal{O})) \quad (B.27)$$

$$closure_f(\mathcal{O}) = (\bigcup n : n \geq 0 : closure\_f_n(\mathcal{O})) \quad (B.28)$$

$$closure\_r_0(\mathcal{O}) = \{rest(\alpha)\theta \rightarrow a\theta \mid \alpha \rightarrow a \in Ar \wedge (\exists f \in closure\_f(\mathcal{O}) :: \theta = mgru(key(\alpha \rightarrow a), f))\} \quad (B.29)$$

$$closure\_r_n(\mathcal{O}) = \{rest(\alpha)\theta \rightarrow a\theta \mid \alpha \rightarrow a \in closure\_r_{n-1}(\mathcal{O}) \wedge (\exists f \in K \cup closure\_f(\mathcal{O}) :: \theta = mgru(key(\alpha \rightarrow a), f))\} \quad (B.30)$$

$$closure_r(\mathcal{O}) = (\bigcup n : n \geq 0 : closure\_r_n(\mathcal{O})) \quad (B.31)$$

Finally,

$$closure_{f\&r}(\mathcal{O}) = closure_r(\mathcal{O}) \cup closure_f(\mathcal{O}).$$

The key lemma for $closure_{f\&r}$, of course, is that it really bounds the facts and rules which can be added by *closure*:

Lemma 50 *For any primitive operation $\mathcal{O}$ allowed in a closed knowledge-base K, and any $S \subset closure_{f\&r}(\mathcal{O})$:*

$$closure(K+S) \subset K+S+closure_{f\&r}(\mathcal{O}).$$

Proof: It suffices to show that, for any rule $\alpha \to a \in K+S$:

$$closure(K+S, \alpha \to a) \subset K+S+closure_{f\&r}(\mathcal{O}).$$

If the length of $\alpha > 1$ then it is fairly easy to show that the rules added are in $closure_r$. If the length of $\alpha = 1$ then $closure$ can add facts, and we must show that these facts are in $ireach_n(q, par_K(\mathcal{O})) \cap ichange_n(q, par_K(\mathcal{O}))$. If $\alpha \to a \in K$ then this is straightforward. If $\alpha \to a \in S$ then a bit more work is required. One must characterize the rules in $closure_r(\mathcal{O})$ in terms of $ireach$ and $frames$. Intuitively, these rules are the result of starting with a rule in Ar, closing it with respect to a fact in S, and then repeatedly closing it with respect to facts in $K+S$. The key is to show that all of the facts we close with respect to contain only constants in $frames(q,p)$. The details are fairly tedious and are left to the reader. ▮

Finally, the key lemma of this subsection. note that by unfolding the definition of ALL operations (applying case 2 of the definition and then case 3), any primitive operation, $\mathcal{O}_n$, can be written as a function of primitive operations (subscripted $n-1$). For any $p \subset C \times R$, let $unfold(\mathcal{O}_n, p)$ be the set of all operations appearing in such an unfolding.

Lemma 51 *Consider any primitive operation $\mathcal{O}$ allowed in a closed knowledge-base K and any $n > 0$. Assume $\mathcal{O} = query(q)$ or q ground and $\mathcal{O} = assert(q)$. Further, let $p = par_K(\mathcal{O})$. For any $\mathcal{O}' \in iops(\mathcal{O})$, there exists an l such that:*

1. *$unfold(\mathcal{O}'_n, p) \subset iops_l(\mathcal{O})$.*
2. *For any fact or rule x, if $x \in \mathcal{O}'_n(K,p)$ and $x \notin K$ then $x \in closure_{f\&r}(\mathcal{O})$.*
3. *For any $\theta \in sub(\mathcal{O}'_n(K,p))$:*
$$range(\theta) \subset frames_l(q,p).$$

Proof: By induction on n. If $n = 0$ then $unfold(\mathcal{O}'_n, p) = \emptyset$, so part 1 is trivial. Part 2 follows by lemma 50. Part 3 follows from the definitions.

Consider $n > 0$. For part 1 we have to show that the operations in equations 5.18 and 5.20 are in $iops_l(\mathcal{O})$. Consider any $\rho \in R$, and let $\alpha = Ant(\rho)$.

Claim: For any α_1, α_2 such that $\alpha = append(\alpha_1, \alpha_2)$:

$$unfold(query_{n-1}(\alpha_1)(K,p)) \subset \{query_{n-1}(q'') \mid q'' \in ibranches_l(\alpha_1, q, p)\}$$

$$(\forall \theta \in sub(query_{n-1}(\alpha_1)(K,p)) :: range(\theta) \subset frames_l(q,p))$$

This claim is proven by induction on the length of α_1 (the trick in the induction step is to use lemma 49). From the first part of the claim (and lemma 46) one can show that the queries in 5.18 and 5.20 are in $iops_n(\mathcal{O})$. The second part of the claim (and lemma 46) imply that the assertions in 5.18 and 5.20 are in $iops_n(\mathcal{O})$.

For part 2, consider any such f. If f was added by *closure* then the result follow by lemma 50. If $f = q$ then the result follows from the definitions. Else, by part 1, there must be some $\mathcal{O}'' \in iops_l(\mathcal{O})$ such that $f \in \mathcal{O}''_{n-1}(K, p)$. Thus, by induction, $f \in closure_{f\&r}(\mathcal{O})$.

For part 3, note that θ is the result of a *lookup* in the frame-slot of $\mathcal{O}'$. Thus, (by definition of $frames_n$) part 3 could only be false if some frame not in $frames_l(q, p)$ were added to the frame-slot of $\mathcal{O}'$ in the calculation of $\mathcal{O}'_n$. But this is impossible by part 2 (and lemma 45). ∎

B.4.3 Proof of Computation Lemma

First we have one last preliminary lemma:

Lemma 52 *Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $n > 0$:*

$$(\forall \mathcal{O}' \in iops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O}))) \tag{B.32}$$

$$\Longrightarrow$$

$$\mathcal{O}_{n+1}(K, par_K(\mathcal{O})) = \mathcal{O}_n(K, par_K(\mathcal{O})). \tag{B.33}$$

Proof: From lemma 51 part 2, $\mathcal{O}_{n+1}(K, par_K(\mathcal{O}))$, and $\mathcal{O}_n(K, par_K(\mathcal{O}))$ both unfold to operations in $iops(\mathcal{O})$. Further, these unfoldings differ only in subscripts, so B.32 implies B.33. ∎

Finally, we put it all together to prove the computation lemma:

Lemma 33 Computation Lemma Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $n > 0$:

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O})))$$

$$\Longrightarrow$$

$$\mathcal{O}(K) = \mathcal{O}_n(K, par_K(\mathcal{O})).$$

Proof: From left-hand side above:

$$(\forall \mathcal{O}' \in ops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O})))$$

By lemma 44 and the definition of *ops* this implies:

$$(\forall \mathcal{O}' \in iops(\mathcal{O}) :: \mathcal{O}'_n(K, par_K(\mathcal{O})) = \mathcal{O}'_{n-1}(K, par_K(\mathcal{O})))$$

But this implies, using lemma 51:

$$(\forall \mathcal{O}' \in iops(\mathcal{O}) :: \mathcal{O}'_{n+1}(K, par_K(\mathcal{O})) = \mathcal{O}'_n(K, par_K(\mathcal{O})))$$

And these together imply, by lemma 52:

$$\mathcal{O}_{n+2}(K, par_K(\mathcal{O})) = \mathcal{O}_{n+1}(K, par_K(\mathcal{O})) = \mathcal{O}_n(K, par_K(\mathcal{O}))$$

And thus (by induction on m):

$$\mathcal{O}_n(K, par_K(\mathcal{O})) = \mathcal{O}_{n+m}(K, par_K(\mathcal{O})).$$

From which the result follows by the monotonicity of $\mathcal{O}_n$ (lemma 8 part 4). ∎

And the two results on the complexity of *closure*:

Lemma 34 Let $\mathcal{O}$ be a primitive operation allowed in a closed knowledge-base K. For any $\mathcal{O}' \in ops(\mathcal{O})$:

1. For any rule ρ, if $\rho \in \mathcal{O}'(K)$, but $\rho \notin K$, then $\rho \in closure_{f\&r}(\mathcal{O})$.
2. For any fact f, if $f \in \mathcal{O}'(K)$, but $f \notin K$, then $f \in closure_{f\&r}(\mathcal{O})$.

Proof: Consider any $\mathcal{O}' \in ops(\mathcal{O})$. Lemma 44 implies, there exists some $\mathcal{O}'' \in iops(\mathcal{O})$ such that $kb(\mathcal{O}'(K)) = kb(\mathcal{O}''(K))$. The result then follows by lemma 51. ∎

Bibliography

[Allen, 83] Allen, J. (1983). Maintaining knowledge about temporal intervals. *CACM* **26**(11):832-43.

[Allen et. al, 84] Allen, J. F., Giuliano, M., and Frisch A. M. (1984). *The HORNE Reasoning System*, TR 126, Computer Science Department, University of Rochester, Rochester NY.

[Apt, 88] Apt, Krzysztof, R. (1988). Introduction to logic programming (revised and extended version), Technical Report TR-87-35, Department of Computer Science, University of Texas at Austin, Austin Texas. To appear in *Handbook of Theoretical Computer Science*, (J. van Leeuwen, Managing Editor), North-Holland.

[Bobrow,1985] D. G. Bobrow, editor. (1985). *Qualitative Reasoning about Physical Systems.* Bradford Books/MIT Press, Cambridge, MA.

[Bobrow & Winograd, 77] Bobrow, Daniel G., and Winograd, Terry (1977). An overview of KRL, a knowledge representation language. *Cognitive Science* **1**(1), 3-46. (Reprinted in [Brachman & Levesque, 85], pp. 263-285.)

[Boolos & Jeffrey, 80] Boolos, George S., and Jeffrey, Richard C. (1980). *Computability and Logic*, Cambridge University Press, New York.

[Brachman, 77] Brachman, R. J. (1977). What's in a concept: structural foundations for semantic networks. *Int. J. Man-Machine Studies 9*: 127-152.

[Brachman, 79] Brachman, R. J. (1979). On the epistemological status of semantic networks. In [Findler, 79]. (Reprinted in [Brachman & Levesque, 85], pp. 191-215.)

[Brachman, 90] Brachman, R. J. (1990). The Future of Knowledge Representation. In *AAAI-90*, pp. 1082-1092.

[Brachman et al., 83] Brachman, R.J., Fikes,R. E., and Levesque H. J. (1983). Krypton: A functional approach to knowledge representation. FLAIR Technical Report No. 16, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, CA, May, 1983 (revised version in IEEE Computer **16**(10), 1983, 67-73). (Reprinted in [Brachman & Levesque, 85], pp. 412-429.)

[Brachman & Levesque, 84] Brachman, R.J. and Levesque, H.J. (1984). The tractability of subsumption in frame-based description languages. In *Proc. AAAI-84*, pages 34-37, Austin, TX.

[Brachman & Levesque, 85] Brachman, R.J. and Levesque, H.J. (1985). *Readings in Knowledge Representation*, Morgan Kaufmann, Los Altos, Cal.

[Bundy et. al, 1985] Bundy, A., Byrd, L., Mellish, C. (1985). Special-purpose, but domain-independent, inference mechanisms. In *Progress in Artificial Intelligence*, ed. L. Steels, J. Campbell, pp. 93-111. London: Ellis Horwood.

[Cadoli & Lenzerini, 90] Cadoli, M., Lenzerini, M. (1990). The complexity of closed world reasoning and circumscription. In *AAAI-90*.

[Crawford] Crawford, J. M. Algernon User's Guide. Forthcoming Technical Report, Artificial Intelligence Laboratory, The University of Texas at Austin.

[Crawford, Farquhar, & Kuipers, 90] Crawford, J., Farquhar, A., and Kuipers, B. (1990). QPC: A Compiler from Physical Models into Qualitative Differential Equations. *AAAI-90*.

[Crawford & Kuipers, 89] Crawford, J. M., and Kuipers, B. (1989). Towards a theory of access-limited logic for knowledge representation. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, Los Altos, California.

[Crawford & Kuipers, 89] Crawford, J. M., and Kuipers, B. (1990). ALL: Formalizing Access Limited Reasoning. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, Morgan Kaufmann, Los Altos, California.

[Crawford & Kuipers, 90] Crawford, J. M., and Kuipers, B. (1990). Towards a Formalization of Access-Limited Logic. Technical Report number AI90-133, Artificial Intelligence Laboratory, The University of Texas at Austin.

[deHaan & Schubert, 86] deHaan, J., and Schubert, L. K. (1986). Inference in a topically organized semantic net. *AAAI-86*, pp. 334-338.

[de Kleer and Brown, 1984] J. de Kleer and J. S. Brown. (1984). A qualitative physics based on confluences. *Artificial Intelligence*, **24**:7–83.

[de Kleer, 1986] J. de Kleer. (1986). An assumption-based truth maintenance system, Extending the ATMS, Problem solving with the ATMS. *Artificial Intelligence*, **28**(2):127–224.

[Dvorak and Kuipers, 1989] D. Dvorak and B. J. Kuipers. (1989). Model-based monitoring of dynamic systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI–89)*, pages 1238–1243, Los Altos, CA. Morgan Kaufman.

[Etherington, 88] Etherington, D.W. (1988). *Reasoning with Incomplete Information.* Pitman, London.

[Etherington et al., 89] Etherington, David W., Borgida, Alex, Brachman, Ronald J., and Kautz, Henry (1989). Vivid knowledge and tractable reasoning: preliminary report. *IJCAI-89* pp. 1146-1152.

[Fagin & Halpern, 88] Ronald Fagin and Joseph Y. Halpern. (1988). Belief, awareness, and limited reasoning. *Artificial Intelligence*: **34**: 39-76.

[Falkenhainer and Forbus, 1988] B. Falkenhainer and K. D. Forbus. (1988). Setting up large-scale qualitative models. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, Los Altos, CA. Morgan Kaufman.

[Findler, 79] Findler, N.V. (1979). *Associative Networks: Representation and Use of Knowledge by Computer*, Academic Press, New York.

[Fine, 85] Fine, Kit (1985). *Reasoning With Arbitrary Objects*, Aristotelian Society Series, Volume 3, Basil Blackwell, Oxford.

[Forbus, 84] K. D. Forbus. (1984). Qualitative process theory. *Artificial Intelligence*, **24**:85–168.

[Forbus, 1989] K. D. Forbus. (1989). Pushing the edge of the (QP) envelope. 3rd Qualitative Physics Workshop, Stanford, CA.

[Forbus, 1990] K. D. Forbus. (1990). The qualitative process engine. In *Readings in Qualitative Reasoning about Physical Systems*, pages 220–235. Morgan Kaufman.

[Franke and Dvorak, 1989] D. Franke and D. Dvorak. (1989). Component connection models. Model-Based Reasoning Workshop, Eleventh International Joint Conference on Artificial Intelligence (IJCAI–89), Detroit, Michigan.

[Ginsberg, 87] Ginsberg, M.L., ed. (1987). *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, San Mateo, CA.

[Graubard, 88] Stephen R. Graubard. (1988). *The Artificial Intelligence Debate.* MIT Press, Cambridge, Mass.

[Hanks & McDermott, 86] Hanks, Steve, and McDermott, Drew (1986). Default reasoning, nonmonotonic logics, and the frame problem. *AAAI-86*, pp. 328-333.

[Hayes, 74] Hayes, Patrick J. (1974). Some problems and non-problems in representation theory. In *Proc. AISB Summer Conference*, University of Sussex. (Reprinted in [Brachman & Levesque, 85], pp. 3-22.)

[Hayes, 79] Hayes, Patrick J. (1979). The logic of frames. In *Frame Conceptions and Text Understanding*, ed. D. Metzing, Walter de Gruyter and Co., Berlin, pp. 46-61. (Reprinted in [Brachman & Levesque, 85], pp. 288-295.)

[Hayes, 85] Hayes, Patrick J. (1985). The second naive physics manifesto. In *Formal Theories of the Commonsense World*, ed. Hobbs, Jerry R. and Moore, Robert C., Ablex Publishing Co., New Jersey, pp. 18-30.

[Hayes-Roth, 85] Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, **26**:251-321.

[Hintikka, 62] Hintikka, J. (1962). *Knowledge and Belief.* Cornell University Press, Ithaca, NY.

[Hunter, 71] Hunter, G. (1971). *Metalogic: An Introduction to the Metatheory of Standard First Order Logic.* University of California Press, Berkeley, CA.

[Iwasaki, 90] Iwasaki, Y. (1990). Reasoning with multiple abstraction models. 4th Qualitative Physics Workshop, Lugano, Switzerland.

[Kay, 73] Kay, M. (1973). The MIND system. In *Natural Language Processing*, ed. R. Rustin. Algorithmics Press, New York.

[Komaroff et al., 77] Komaroff, A.L., McCue, K., and Pass, T.M. (1977). Headache. In *Common Acute Illness: A Problem-Oriented Textbook with Protocols*, eds. Komaroff, A.L., and Winickoff, R.N. Boston, Mass., Little, Brown and Company, 1977.

[Kuipers, 78] Kuipers, B. J. (1978). Modeling spatial knowledge. *Cognitive Science* **2**: 129-153.

[Kuipers, 79] Kuipers, B. J. (1979). On representing commonsense knowledge. In [Findler, 79].

[Kuipers, 1984] B. J. Kuipers. (1984). Commonsense reasoning about causality: deriving behavior from structure. *Artificial Intelligence*, **24**:169–204.

[Kuipers, 1985] B. J. Kuipers. (1985). The limits of qualitative simulation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Altos, CA. William Kaufman.

[Kuipers, 1986] B. J. Kuipers. (1986). Qualitative simulation. *Artificial Intelligence*, **29**:289–338.

[Kuipers, 1989] B. J. Kuipers. (1989). Qualitative reasoning: modeling and simulation with incomplete knowledge. *Automatica*, 25:571–585.

[Kuipers and Berleant, 1988] B. J. Kuipers and D. Berleant. (1988). Using incomplete quantitative knowledge in qualitative reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 324–329, Los Altos, CA. Morgan Kaufman.

[Kuipers and Byun, 1988] B. J. Kuipers and Yung-Tai Byun. (1988). A Robust, Qualitative Method for Robot Spatial Learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 774-779, Los Altos, CA. Morgan Kaufman.

[Kuipers and Chiu, 1987] B. J. Kuipers and C. Chiu. (1987). Taming intractable branching in qualitative simulation. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87).*

[Lenat et al., 86] Doug Lenat, Mayank Prakash, and Mary Shepard. (1986). CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Magazine* **VI**(4).

[Lenat and Guha, 1990] D. B. Lenat and R. V. Guha. (1990). *Building Large Knowledge-Based Systems.* Addison-Wesley, Reading, MA.

[Levesque, 84] Levesque, H.J. (1984). A logic of implicit and explicit belief, *AAAI-84*, pp. 198-202.

[Levesque, 86] Levesque, H. J. (1986). Knowledge representation and reasoning. In *Ann. Rev. Comput. Sci.* 1:255-87. Annual Reviews Inc, Palo Alto, California.

[Levesque & Brachman, 85] Levesque, H.J., Brachman R.J. (1985). A fundamental tradeoff in knowledge representation and reasoning (revised version). In [Brachman & Levesque, 85], pp. 41-70.

[Lloyd, 84] Lloyd, J.W. (1984). *Foundations of Logic Programming*, Springer-Verlag, New York.

[McAllester et al., 89] McAllester, D., Givan, R., and Fatima, T. (1989). Taxonomic syntax for first order inference. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, Los Altos, California, pp. 289-300.

[McAllester, 90] McAllester, D. (1990). Truth Maintenance. In *AAAI-90*, pp. 1109-1116.

[McCarthy, 87] McCarthy, J. (1987). Generality in artificial intelligence. *CACM* **30:**1030-1035.

[Minsky, 75] Minsky, Marvin. (1975). A framework for representing knowledge. In *The Psychology of Computer Vision*, ed. P.H. Winston, McGraw-Hill, New York. (A later version reprinted in [Brachman & Levesque, 85], pp. 246-262.)

[Moore, 79] Moore, R. (1979). Reasoning about Knowledge and Action, MIT PhD thesis.

[Murray & Porter 89] Murray, K.S., and Porter, B.W. (1989). Controlling search for the consequences of new information during knowledge integration. In *Proceedings of the Sixth International Workshop on Machine Learning.*

[Murray 90] Murray, K.S. (1990). Improving explanatory competence. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society.*

[Patel-Schneider, 85] Patel-Schneider, P.F. (1985). A decidable first-order logic for knowledge representation. *IJCAI-85.*

[Patel-Schneider et al., 90] Patel-Schneider, P.F., et at. (1990). Term subsumptions languages in knowledge representation. In *AI Magazine* **11**(2):16-22.

[Palamidessi, 89] Palamidessi, Catuscia. (1989). Algebraic Properties of Idempotent Substitutions. University of Pisa, manuscript.

[Pearl, 88] Pearl, Judea. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, Palo Alto, CA.

[Pearl, 86] Pearl, Judea. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* **29**:241-288.

[Porter *et al.*,1988] B. W. Porter, J. Lester, K. Murray, K. Pittman, A. Souther, L. Acker, and T. Jones. (1988). AI research in the context of a multifunctional knowledge base: the botany knowledge base project. Technical Report AI TR-88-88, University of Texas at Austin.

[Powers, 78] Powers, L. H. (1978). Knowledge by deduction. The Philosophical Review, LXXXVII, No. 3, pp. 337-371.

[Quillian, 67] Quillian, M. Ross. (1967). Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science* **12**, 410-430. (Reprinted in [Brachman & Levesque, 85], pp. 98-118.)

[Raphael, 68] Raphael, B. (1968). SIR: semantic information retrieval. In *Semantic Information Processing*, ed. M. Minsky, MIT Press, Cambridge, MA., pp. 33-145.

[Reiter, 80] Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, **13**:81-132.

[Reiter, 87] Reiter, R. (1987). Nonmonotonic reasoning. In *Annual Review of Computer Science, Volume 2*, pp. 147-186. Annual Reviews Inc., Palo Alto, CA.

[Schubert, 79] Schubert, L.K., R.G. Goebel, and N.J. Cercone, (1979). The structure and organization of a semantic net for comprehension and inference. In [Findler, 79], pp. 121-175.

[Schubert et al., 83] Schubert, L.K., Papalaskaris, M.A., and Taugher, J. (1983). Determining type, part, color, and time relationships. *IEEE Computer*, **16**(10), 53-60.

[Schubert & Hwang] Schubert, L.K., and Hwang C.H. (1989). An Episodic Knowledge Representation for Narrative Texts. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, Los Altos, California.

[Schwarcz et al., 70] Schwarcz, R.M., Burger, J.F., and Simmons, R.F. (1970). A deductive question-answerer for natural language inference. *CACM 13*, **3** (March 1970), 167-183.

[Shapiro, 78] Shapiro, S.C. (1978). Path-based and node-based inference in semantic networks. *Tinlap-2: Theoretical Issues in Natural Languages Processing*, ed. D. Waltz, ACM, New York, 219-225.

[Shapiro, 89] Shapiro, S.C. and the SNePS Implementation Group (1989). SNePS-2 User's Manual. Department of Computer Science, SUNY at Buffalo, 31pp.

[Shapiro & Woodmansee, 69] Shapiro, S.C. and Woodmansee, G.H. (1969). A net structure based relational question answerer: description and examples. *IJCAI-69*, The MITRE Corp., Bedford, MA., pp. 325-346.

[Shastri & Ajjanagadde, 90] An Optimally Efficient Limited Inference System. (1990). In *AAAI-90*, pp. 563-570.

[Smith & Genesereth, 85] Smith, David E., and Genesereth, Michael R. (1985). Ordering conjunctive queries. *Artificial Intelligence*, **26:** 171 - 215.

[Stallman & Sussman, 77] Stallman, R.M. and Sussman, G.J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9**:135-196.

[Touretzky, 86] Touretzky, David S. (1986). *The Mathematics of Inheritance Systems*, Morgan Kaufmann, Los Altos, CA.

[Treitel & Genesereth, 87] Treitel, Richard, and Genesereth, Michael R. (1987). Choosing directions for rules. *Journal of Automated Reasoning* **3:** 395-431.

[Vilain, 85] Vilain, M. (1985). The restricted language architecture of a hybrid representation system. *IJCAI-85*, pp. 547-551.

[Vincent,1988] T. C. Vincent. (1988). Model building using qualitative process theory. Master's thesis, UT Austin.

[Warren, 83] D. H. D. Warren (1983). An Abstract Prolog Instruction Set. Technical note number 309, Artificial Intelligence Center, SRI International, October, 1983.

[Weld and de Kleer, 1990] D. S. Weld and J. de Kleer. (1990). *Readings in Qualitative Reasoning About Physical Systems.* Morgan Kaufman, Los Altos, CA.

[Williams, 1988] B. Williams. (1988). Minima: A symbolic approach to qualitative algebraic reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 264–269, Los Altos, CA. Morgan Kaufman.

[Williams, 1989] B. Williams. (1989). Qualitative analysis of MOS circuits. *Artificial Intelligence*, **24**:281–346.

[Woods, 75] Woods, W. (1975). What's in a link: foundations for semantic networks. In *Representation and Understanding: Studies in Cognitive Science*, ed. Bobrow, D. and Collins, A., Academic, New York, pp. 35-82.

[Wylie, 57] Wylie, C.R., Jr. (1957). *101 Puzzles in Thought & Logic*, Dover Publications Inc, Mineola, New York.

Index

a-assert, 36
a-query, 36
access path, 10, 109
access paths, 16
ALL(+neg), 133
all-paths, 27
allowed, 110
alphabet, 108
Ant, 109
apply, 30
arbitrary objects, 66, 166
ask, 20, 28
ask-slots, 28, 94
assertion, 110
assertions, 17
assume, 27
assumption, 137

backward-chaining, 17
bind, 30
bind-to-values, 30
Boil the patient, 9
branch-on-values, 30
built in knowledge-base, 32

chronological backtracking, 138
clear-slot, 28
closed, 116
complexity of closure, 158
complexity theorem, 159
computation lemma, 158
Conseq, 109
consistent, 142
counter, 73
create, 27

decl-slots, 28
default reasoning, 85
del-rule, 26
del-rules, 26
del-srule, 26
del-srules, 26
delete, 28
dependency directed backtracking, 138

ECOSYSTEM, 161

f-inconsistent, 135
fact, 108
finite state machine, 75
forc, 20, 26
formula, 110
forward-chaining, 17
frame accesses, 37
frame insertions, 37
frames, 15
functional shorthand, 21, 24

generic rules, 25
ground, 108
ground substitution, 112

headache protocol, 92
Herbrand Base, 127
Herbrand Model, 127
Herbrand Universe, 127

if-added, 17
if-added closed, 116
if-added incompleteness, 31, 115
if-added rule, 109
if-needed, 17
if-needed rule, 109
if-needed rule completion, 60

immediate consequence operator, 128
in-own-partition, 26
insertions, 37
iterations, 37

kbs, 135
KEE, 163
Key, 109
knowledge theory, 110
Knowledge-Base, 110
knowledge-base structures, 135

lisp, 29
logic program, 127

matches, 37
maximally p-consistent, 152
mgru, 112
mixed quantification, 166
modus tolens, 59
more general substitution, 112
most general unifier, 112

name, 21
names of frames, 15, 24
neq, 30
Nixon diamond, 85
numbers, 25

operation, 110

p-consistent, 152
Partitional Completeness, 11, 126, 165
partitioning, 110
partitions, 11, 31, 110, 111
primitive operation, 110
primitive path, 109
primitive query, 110
Prolog, 162
proposition, 108
provability, 147

queries, 17
query, 110

reasoning by contradiction, 50
reductio ad absurdum, 50
relevant, 112
renaming, 112
representative, 66
retrieve, 27
rule applications, 37
rule completion, 31
rules, 20, 25

show, 28
slot, 20, 28
Socratic Completeness, 11, 126, 165
srules, 20, 25
strings, 25
substitution, 112
symbol grounding, 15

taxonomy, 20, 30
term, 108
the, 20, 27

unifications, 37
unifier, 112
unp, 27

variant, 112
views, 85
vivid knowledge-bases, 161

w-contra-positive, 26
well-formed, 136
wo-contra-positive, 26