CI2 - A Logic for Plural Representation

Felix Yen
Ph.D Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-90-37
December 1990

CI2 – A Logic for Plural Representation

Felix Yen

Department of Computer Science
Brown University Box 1910
Providence, RI 02912

ABSTRACT

The problem we address is the representation of plurals. The solution we present is CI2, a first-order logic named after the *calculus of individuals*. (If one thinks of individuals as being sets, then CI2 is just the calculus of individuals with numbers added.) In CI2, plurals are represented with terms denoting sets. (Singulars are represented with terms denoting unit sets.)

We are primarily concerned with the problem of representing ambiguous sentences and making inferences from them. Our solution involves using a ''new'' but simple ontology as well as special classes of functions which we describe. The result is a theory that enables us to disambiguate sentences incrementally. We also describe a CI2-specific theorem prover (fully implemented) that uses several techniques that may be of use to those interested in developing systems that can reason about sets.

We demonstrate the viability of our theory by presenting representations for twelve ambiguous, plural sentences. We show how each sentence can be disambiguated and how the desired inferences were computed by our theorem prover.

Our dissertation concludes by addressing the problem of modifying existing programs so that they can manipulate plural entities. This is not a trivial modification as it entails solving the closure problem, i.e. making closure assumptions. We present a promising approach in which closure assumptions are made only when needed.

Table of Contents

Chapter 1: Preliminaries

1.1. Introduction

The problem we address is the representation of plurals. The solution we present is CI2, a first-order logic named after the *calculus of individuals* ([Leonard40]). In CI2, plurals are represented with terms denoting sets. (Singulars are represented with terms denoting unit sets.)

We begin by describing the plural representation problem, presenting our goals, and defining the terms used throughout this dissertation. Our work extends results from the Linguistics and Philosophy literature. We review this work in chapter 2, along with related work in Knowledge Representation, Natural Language, Automated Reasoning, and Logic Programming. We describe our logic in chapter 3 and a prototype CI2 theorem prover in chapter 4. In chapter 5, we demonstrate the viability of our theory by presenting representations for twelve ambiguous, plural sentences. We show how each of these sentences can be incrementally disambiguated and how the desired inferences were computed by our theorem prover. In chapter 6, we address the problem of adding plurals to existing systems. We summarize our work and propose topics for future research in chapter 7.

1.2. Representing Plurals

The plural representation problem should be viewed as being a proper superset of its singular analogue. We characterize the singular representation problem as follows: we wish to mechanically convert a stream of singular sentences into a stream of representations and we want to be able to write programs that efficiently make inferences from these representations. Our characterization of the plural representation problem is derived by allowing the input stream to contain plural sentences.[1]

Our primary goal, efficient inference, is unchanged, but the inclusion of plurals introduces a new kind of ambiguity. A sentence is *ambiguous* if it has more than one *reading* or interpretation. A singular sentence is ambiguous only if at least one of its words can be interpreted in more than one way, e.g. "John saw her duck," or if it contains an adverb or prepositional phrase that can modify a noun phrase or a verb, e.g. "John saw the woman with the telescope,"[2] but a plural sentence may be ambiguous in other ways. Unfortunately, the linguistic literature does not yield a consensus as to what constitutes a reading. We will follow Scha's lead and use a liberal definition of reading. [Scha81] describes a system in which readings are generated using purely syntactic rules. Since these rules are syntactic, it is not unusual for one of Scha's readings to be

---

[1] A plural sentence is a sentence that refers to a plural entity. This reference need not be explicit. For example, "Every man loves some woman" is plural because it refers to all men.

[2] We will ignore these types of ambiguity from now on.

semantically anomalous, nor is it unusual for two of his readings to be semantically indistinguishable from each other. Unwanted readings are discarded by appealing to our data base. In other words, syntactic analysis yields an ambiguous sentence's readings and the sentence's intended reading is determined by semantic and pragmatic analyses.

Suppose we have an ambiguous sentence S with a single plural noun phrase α and verb (predicate) P. According to [Scha81], most of S's readings are the result of interpreting α either *distributively* or *collectively*. The distributive interpretation says that P is true of each element in α. There are two collective interpretations. The *collective-A* interpretation, says that P is true of α. And for each $a \in \alpha$, the *collective-B* interpretation says that P is either true of a or it is true of some a-containing subset of α. Let us illustrate with an example:

[0] 600 Dutch firms bought 5000 American computers.[3]

(We will refer to this example repeatedly.) Our analysis yields nine collective-distributive readings:

[R1] firms: distributive, computers: distributive.
There are 600 firms and as many 600 sets of computers, each of cardinality 5000.† Each firm bought each of the computers in one of these sets (one at a time).

[R2] firms: distributive, computers: collective-A.
There are 600 firms and as many as 600 sets of computers, each of cardinality 5000.† Each firm bought one of these sets of computers (all at once).

[R3] firms: distributive, computers: collective-B.
There are 600 firms and as many as 600 sets of computers, each of cardinality 5000.† Each firm bought groups of computers (one group at a time), the union of these groups being one of these sets of computers.

[3] This is almost identical to sentence [7] which was taken from [Scha81].

† The sets may intersect.

[R4] firms: collective-A, computers: distributive.
There is a set of 600 firms that collectively bought each of a set of 5000 computers (one at a time).

[R5] firms: collective-A, computers: collective-A.
There is a set of 600 firms that collectively bought a set of 5000 computers (all at once).

[R6] firms: collective-A, computers: collective-B.
There is a set of 600 firms that collectively bought groups of computers (one group at a time). The union of these groups has cardinality 5000.

[R7] firms: collective-B, computers: distributive.
There are *n* groups of firms. Their union has cardinality 600. There are as many as *n* sets of computers, each having cardinality 5000.† Each group of firms collectively bought each of the computers in one of these sets (one at a time).

[R8] firms: collective-B, computers: collective-A.
There are *n* groups of firms. Their union has cardinality 600. There are as many as *n* sets of computers, each having cardinality 5000.† Each group of firms collectively bought one of these sets of computers (all at once).

[R9] firms: collective-B, computers: collective-B.
There are *n* groups of firms. Their union has cardinality 600. There are as many as *n* sets of computers, each having cardinality 5000.† Each group of firms collectively bought groups of computers (one group at a time), the union of these groups being one of these sets of computers.

Readings may ''overlap'' and even subsume each other. In our example, [R2] is equivalent to the reading of [R3] in which there is only one group of computers and [R1] is equivalent to the [R3] reading in which there are 5000 degenerate groups, i.e. unit sets, of computers. In other words, [R3] subsumes [R1] and [R2]. [R6] and [R9] subsume [R4], [R5], [R7], and [R8] for the same reason. Continuing in this vein, we note that [R6] is equivalent to the [R9] reading in which there is only one group of firms and [R3] is equivalent to the [R9] reading in which there are 600 degenerate groups of firms. In other words, [R9] subsumes the other eight collective-distributive readings.

Scha would attribute [0]'s tenth reading, which we will call its *sum-of-plurals*[4] reading, to cumulative quantification. This reading has two characteristics. First, the plurals are independent, i.e. this reading entails there being exactly 600 firms and exactly

† The sets may intersect.

[4] Our use of this term is due to [Schein86]. This reading subsumes Webber's conjunctive reading.

5000 computers. Second, no mapping is imposed between the firms and the computers. We say that the reading is *underspecified*. In other words, we have:

[R10] sum-of-plurals.
There are *n* groups of firms. The union of these *n* groups has cardinality 600. There are *m* groups of computers. The union of these *m* groups has cardinality 5000. Each group of firms has at least one group of computers and each group of computers is had (owned) by at least one group of firms.

[R9] and [R10] can overlap, but neither subsumes the other. They are independent because [R10] specifies the total number of computers whereas [R9] specifies the number of computers associated with each group of firms. In fact, this is the *only* difference between [R9] and [R10]. In general, we find that readings always agree on the types of entities and relationships involved and only disagree on how cardinality information is used.

1.3. Inference and Disambiguation

Our primary goal is facilitating inference.[5] We cannot depend on always being able to disambiguate a sentence before we need to use it to make an inference. In fact, we cannot depend on always being able to disambiguate. So facilitating inference entails being able to make inferences before disambiguation has taken place. In other words, being able to represent a sentence's readings is not enough. We can represent a sentence with a disjunction of its readings, but this representation will be practicable only if the number of disjuncts is small.

We can represent sentence [0] with a disjunction of our representations of [R9] and [R10] because these two readings subsume all of [0]'s readings. In general, if we have a language that allows us to conveniently represent underspecified readings, we will be able to take advantage of the subsumption of readings and represent ambiguous sentences with compact disjunctions. We will also be able to incrementally constrain these representations by asserting statements contradicting the unwanted readings.

Returning to our example, suppose we happen to know that companies do not buy computers collectively and we add a representation of this fact to our data base. Adding this assertion effectively disambiguates a representation of [R9] so that it becomes a representation of [R3].

If we begin with a data base of statements about our domain, we will find that many unwanted readings will be ruled out in this manner.

---

[5] Disambiguation can be viewed as a form of inference.

1.4. Caveats

Since the ontological status of intensions is rather controversial, we will restrict ourselves to plurals (and singulars) in extensional contexts. In other words, we will not attempt to represent beliefs, desires, or the difference between the Morning Star and Evening Star. We will also ignore the problem of representing generics, e.g. the generic reading of ''Donkeys are stubborn.'' Nor will we address the problems of representing negation and time. We feel safe making these restrictions because we believe that the representational issues that owe their existence to plurals do not depend on these other problems.

We have been interpreting (as did Scha) '600 Dutch firms' as meaning ''precisely 600 Dutch firms.'' We acknowledge that '600 Dutch firms' might be interpreted as meaning ''about 600 Dutch firms'' but we will ignore these ''fuzzy'' readings because we feel that their representation is not a plural representation problem, but an uncertainty representation problem.

Our next caveat concerns our solution's being in the form of a first-order logic. We do not claim that logic is a viable representation language. We do feel that there is a compelling methodological reason for using logic. Its expressivity and its clear semantics have made it the representational *lingua franca* of researchers in linguistics and philosophy for decades. It is easier to take advantage of their work if we continue this tradition.

Let us end this chapter on a more positive note by pointing out that much of the complexity here can be traced to the plural representation problem's natural language roots. If this language-based ambiguity is not present in the domain we are representing, we see no reason why we cannot ignore these complications and use a subset of CI2 to represent sets in this domain.

Chapter 2: Related Work

2.1. Introduction

In this chapter, we review related work from a variety of sources. We begin by surveying the artificial intelligence literature (sections 2.2–2.4) discussing work from the areas of knowledge representation (*KR*), natural language, and automated reasoning.

The plural representation problem can be thought of as being a KR problem. Unfortunately, the KR literature concentrates almost exclusively on representing singulars. Our problem might also be thought of as a natural language problem, but once again we find researchers concentrating on other issues. We also might expect some progress to have been made in the area of automated reasoning, but as we shall see the work done there does not address the problem of plural representation at all.

Recently, there has been some interest in extending Prolog in order to allow one to prove theorems about sets. Unfortunately, no one in this area has applied their results to plural representation and in section 2.5, we shall see why.

The best work on plurals has been done by philosophers and linguists. Our approach borrows heavily from the work described in sections 2.6 and 2.7.

2.2. In Knowledge Representation

KL-ONE is one KR system that does address the plural representation problem. [Brachman79] describes a *DSET* Concept which allows one "to capture sets implicit in noun phrases and clauses." There is no formal difference between this Concept and KL-ONE's singular Concepts.

Unfortunately, the KL-ONE literature does not address the problem of representing ambiguity. It is not obvious how one would represent sentence [0]'s distributive readings in KL-ONE without creating six hundred Dutch-firm Concepts.

2.3. In Natural Language

2.3.1. Webber

Attempts to represent facets of natural language have resulted in several encounters with the plural representation problem. [Webber79] concentrates on representing sentences containing anaphora. Webber uses traditional logical syntax and allows plural and singular terms to be used interchangeably, but she neglects to provide a semantic account of this interchangeability. She also neglects to address the problem of representing ambiguity.

2.3.2. Hobbs

[Hobbs83] introduces the term ''ontological promiscuity'' to describe an approach in which ''one abandons all ontological scruples.'' In particular, Hobbs uses Davidsonian events ([Davidson67]) to represent all predications. (Davidson only used events to represent instances of actions, e.g. ''movements, births, landslides, and explosions'' ([Davidson70]).) We will also use this idea and, like Hobbs, will make no attempt to justify our theory's ontology.

[Hobbs83] concerns itself with the problem of representing quantified English sentences. Hobbs gives an example in which a plural and a singular, both represented by first-order terms, are used interchangeably and he fails to justify this interchangeability. Though his theory does enable one to represent ambiguous sentences, it does not enable one to make any inferences from these representations. (The sentences must be disambiguated first.)

2.4. In Automated Reasoning

AURA and ONTIC ([McAllester87, Wos84]) can be used to ''discover'' theorems of mathematics so it would seem that they must be capable of representing plurals. The fact that they can is somewhat misleading however. In the next chapter, it will become evident that *any* system based on a first-order theory is capable of representing sets as long as we interpret its domain constants appropriately. AURA and ONTIC do nothing to facilitate plural representation. For example, they do not address the problem of representing ambiguity and they do not provide axioms for making inferences about the subsets and members of a plural noun phrase.

2.5. In Logic Programming

LPS ([Kuper87]) is similar to Prolog in that it is based on a first-order logic and has its own proof theory. In order to represent sets, it uses a two-sorted domain of interpretation. The first sort contains *atoms*, e.g. people and hamburgers, whereas the second sort contains finite sets of atoms. There is a membership predicate relating the elements of these two sorts. This system has its limitations however:

- Function values must be atomic. The only exceptions are the special set constructor function symbols '$\{_n$'. '$\{_n(a_1, \ldots, a_n)$' denotes $\{a_1, \ldots, a_n\}$.
- Function arguments must be atomic.
- Axioms for making inferences about a set's members and subsets are not provided.
- The proof theory does not make use of most general unifiers so an LPS unifier must generate all unifying substitutions.

The restrictions on functions hamper LPS's ability to create plural representations. For example, one would not be able to have a function mapping residences to residents unless everyone lived alone (because function values must be atomic). Nor could one have a function mapping residence owners to residences unless no residences were jointly owned (because function arguments must be atomic). Finally, Kuper's logic does not allow one to interchange atomic terms with set terms. Such a substitution leads to an illegal expression. This property might seem to be a desirable one at first glance, but it has horrible consequences. What was once a single n-place predicate P must now be represented by a collection of predicates for each permissible combination of sets and individuals. We would need as many as 2^n predicates! We would also need as many as $\binom{2^n}{2}$ axioms relating these predicates to each other so that we can make inferences about P-predications.

To put this more concretely, suppose P is the dyadic predicate 'own' and that both of its arguments may be sets or individuals. Then, we would need four predicates, e.g. $Own_{i,i}$, $Own_{i,s}$, $Own_{s,i}$, and $Own_{s,s}$, to represent P. Suppose we know that Chris owns a dog and a cat and we represent this fact with "$Own_{i,s}$(Chris, PetsOf(Chris))" where 'PetsOf(Chris)' denotes Chris' dog and Chris' cat. We would like to be able to infer "$Own_{i,i}$(Chris, DogOf(Chris))" from this. For inferences such as this, we need as many as six axioms relating our four predicates in pairwise fashion, e.g. "$Own_{i,s}(o,p) \wedge e \in p \rightarrow Own_{i,i}(o,e)$". Inference will clearly suffer from this proliferation of predicates and axioms. We conclude that formally distinguishing sets from individuals is a bad idea and that we should instead allow sets and individuals to be used interchangeably.

2.6. In Philosophy

CI2 takes its name from the Calculus of Individuals (*CI*), a first-order theory originally formulated by Leśniewski ([Leśniewski31]) and then rediscovered by Leonard and Goodman ([Leonard40]).[1] More recently, others have adopted it for their own uses, e.g. to explain the semantics of mass terms and group quantification ([Burge72, Massey76, Parsons70]). We present here the formulation of CI found in [Massey76].

What differentiates CI from other first-order theories is its ontology. We begin with D, a domain of atoms. A traditional first-order theory would have D as its domain of interpretation. Instead, we introduce *individuals*, which are either atoms (*atomic individuals*) or *sums* of atoms (*sum individuals*). A sum individual is not a set. Leonard and Goodman explain the difference as follows:[2]

---

[1] Leśniewski first published his theory in an obscure journal in 1916 ([Luschei62], p. 20, 28). The name "calculus of individuals" and its popularity in English-speaking countries are due to Leonard and Goodman. ([Leśniewski31] used a nonstandard notation and was written in Polish.)

[2] [Leonard40], p. 45.

> The concept of an individual and that of a class may be regarded as different devices for distinguishing one segment of the total universe from all that remains. In both cases, the differentiated segment is potentially divisible, and may even be physically discontinuous. The difference in the concepts lies in this: that to conceive a segment as a whole or individual offers no suggestion as to what these subdivisions, if any, must be, whereas to conceive a segment as a class imposes a definite scheme of subdivision – into subclasses and members.

But sum individuals do behave a lot like sets. To illustrate the similarity, let *A(i)* be the set of atoms contained in the 'segment' i. If a and b are individuals, then the sum of a and b is c if and only if $A(a) \cup A(b) = A(c)$.

CI's domain of interpretation is the set of individuals induced by D, i.e. the union of D and the sums of its elements. With this ontology, we can use plural (sum individual) terms and singular (atomic individual) terms interchangeably. Unfortunately, neither Leśniewski nor Leonard and Goodman address the problems of ambiguity and inference.

2.7. In Linguistics

2.7.1. Kempson & Cormack

[Kempson81] directly addresses the problem of representing ambiguity.[3] Kempson and Cormack advocate representing an ambiguous sentence with a single, compact representation. We differ on what this representation should be however. Sets are not domain elements in [Kempson81] (though they are not explicitly excluded) so Kempson and Cormack cannot predicate over sets. For example, their representations of [0] would be inconsistent with a reading in which two firms jointly owned a particularly expensive computer.

2.7.2. Scha

[Scha81] and CI2 are similar in that their domains of interpretation do not contain individuals. In both theories, terms denote sets. Scha's *meaning postulates* are statements that "define the application of a predicate to a collection in terms of the application of that same predicate to the smallest parts of that collection." These enable one to make inferences about a set's members (actually, unit subsets).

---

[3] Actually, Kempson and Cormack suggest that sentences like [0] are not ambiguous, but have a single semantic representation from which their particular readings are derived. As computer scientists, we are not interested so much in linguistic definitions as we are in the representational problems arising from these sentences. So, for the purposes of this discussion, we will treat [Kempson81] as though it maintained the more traditional definition of ambiguity.

Scha's use of two predicates for each verb is a minor problem. For example, the dyadic predicate *Own* also has a monadic analogue whose argument is an ordered pair. But the key difference between [Scha81] and CI2 is that Scha does not address the problem of disambiguation.

[Scha88] presents a more complicated theory in which the domain of interpretation is extended to contain sets of sets in order to represent sentences like:

[4] The juries and the committees gather.

In chapter 5, we argue that this extension is unwarranted.

2.7.3. Schein

[Schein86] gives a comprehensive survey of plural representation in the Linguistics literature. Schein argues for a theory whose domain of interpretation contains Davidsonian events ([Davidson67]). He allows events to "include states or situations" ([Schein86], p. 66) as Hobbs did in [Hobbs83].[4] But like the other work we have reviewed, Schein does not address the problems of ambiguity and inference.

[4] Davidson only used events to represent instances of actions.

Chapter 3: CI2

3.1. Syntax & Semantics

CI2 is syntactically identical to the first-order predicate calculus and its semantics are Tarskian. In the next section, we show how its ontology facilitates plural representation.

3.2. Ontology

A plural representation theory needs to have plural-denoting terms in order to represent collective readings so we represent plurals with terms denoting non-unit sets and we represent singulars with terms denoting unit sets.[1] The addition of set-denoting terms obviates the need for atom-denoting terms. Singulars can be viewed as being degenerate plurals, i.e. unit sets. Once we adopt this view, the singular representation problem disappears.

Let us harp on this point a bit. Traditionally, we had a predicate for relating classes, e.g. *is-a*, and a predicate that allowed one to make statements about class membership, e.g. *inst*. People often say *is-a* when they mean *inst* and this confusion is understandable. The statement "(*is-a* horse mammal)" is often viewed as being an abbreviation for "horses inherit the properties of mammals," while "(*inst* Secretariat horse)" abbreviates "Secretariat inherits the properties of horses."[2] A program that can navigate is-a hierarchies would see little difference between *inst* and *is-a*. We argue that *inst* can be omitted without causing any problems.

If A is a set, then the predication $P(A)$ says nothing about the subsets and members of A thus leaving open the possibility of collective and distributive readings. In other words, the underspecified nature of plural readings is neatly captured by the semantics of predication.

In order to insure that all well-formed expressions denote some element in the domain, we include the empty set in our domain of interpretation. In other words, our domain of interpretation D is $2^A \cup N$ where A is some set of atoms and N is some set of numbers containing the natural numbers, e.g. the integers or the reals.

The set domain elements, i.e. the elements of 2^A, can be organized as a lattice.[3] The lattice's maximal element is A and its minimal element is $\emptyset$. We can associate any number of constant terms, i.e. domain constants or constant functional expressions, with these lattice elements.

---

[1] This approach is suggested in [Webber79], chapter 4. It was also used by Scha in [Scha81].

[2] We are ignoring the issue of defeasibility.

[3] Formal definitions of lattices can be found in lattice theory textbooks, e.g. [Grätzer78]. For our purposes, a lattice can be thought of as being a kind of directed graph. Our lattice has domain elements as its vertices. If a and b are vertices, then there is an edge from a to b if and only if $a \subseteq b \wedge \#a + 1 = \#b$.

In [Frisch82, Hobbs83] events, e.g. eatings, sellings, and walkings, were considered to be domain elements following [Davidson67]. Hobbs and Schein both extend the notion of events to include states or situations. We embrace this extension and include sets of events in our domain of interpretation.[4]

Associated with an event are some number of *constituents*, e.g. its agent and patient. The union of an event set's i-th constituents is its i-th *constituent set*. Constituents must be nonempty sets.[5] It follows that only empty event sets have empty constituent sets. And by definition, all of the empty event set's constituents are empty.

In chapter 1, we saw how competing readings differ only in the way that they make use of cardinality information. So, in order to represent this information, we include numeric terms in our domain of interpretation.

3.3. Special Functions

We augment the functions traditionally used in first-order representational theories, e.g. Skolem functions, with some special functions. We introduce these functions here.

If 'S' denotes a set, then '$\#S$' denotes its cardinality.

The i-th constituent set of an event set may be accessed using the *constituent function* const_i. For example, if 'E' denotes an event set, then '$\mathsf{const}_1(E)$' denotes its first constituent set.

Event functions map constituent sets onto event sets. For example, suppose Eat is an event set with first constituent set Eater and second constituent set Eaten. The event function eat-of, maps subsets of Eater to the corresponding subsets of Eat. Thus, if Chris $\subseteq$ Eater, then 'eat-of (Chris)' denotes the subset of Eat associated with {Chris}. In general, if f is an event function mapping onto E its i-th constituent set, then $f(a)$ is the union of the events in E that have a subset of a as their i-th constituent.[6]

The composition of a constituent function with an event function is especially useful as it allows one to map constituent sets onto each other. In our example, 'const_2(eat-of (Chris))' denotes the set of things eaten by {Chris}. We can easily coin representations for the entities referenced in a sentence, even if they are referenced implicitly, by composing expressions from our constants and functions.

---

[4] Schein views events as being "set-like" so he does not need sets of events. We are not sure of Davidson's position on this matter.

[5] Intuitively, an event having a null constituent is a nonevent. For example, if nothing was eaten, then there was no eating event. If I hit Eugene on the head with nothing, then there was no hitting event.

[6] One might also define $f(a)$ as being the union of the events that have a as their i-th constituent, but the definition given is more useful in practice.

3.4. Predicates

Now that we are representing states and relations with events instead of predications, which predicates do we still need? For set terms, all we need are the equality and subset predicates. Monadic (type) predications, e.g. "$P(a)$", are replaced by the corresponding subset predications, e.g. "$a \subseteq P$". Predicates of more than one argument are replaced with events whose constituents are specified using the equality or subset predicate. Those who wish to represent states or relations without reifying them as events should consult appendix A where we show how relations can be added to CI2. The predicate '>' is useful for making statements about numeric terms. Other numeric predicates might also be useful.

3.5. Proper Axioms

Now that our semantic account is complete, we can focus our attention on the proper axioms that syntactically differentiate CI2 from other first-order theories. In addition to the axioms that define the equality, subset, and greater-than predicates, we also need ones for our event and constituent functions. These axioms describe the behavior of our functions and predicates from a proof-theoretic standpoint.

If f is an event function mapping onto E its i-th constituent set, we have:

A1. $f(x) = \varnothing \leftrightarrow x \cap \mathsf{const}_i(E) = \varnothing$

A2. $f(\mathsf{const}_i(E)) = E$

A3. $x \subseteq y \rightarrow f(x) \subseteq f(y)$

Axiom A1 states that f maps its argument to the empty set when it does not intersect $\mathsf{const}_i(E)$. A2 stipulates that E's i-th constituent set maps to E. And A3 states that f is monotonic. The importance of this property is discussed in chapter 5.

The constituent functions axioms are:

B1. $\mathsf{const}_i(\varnothing) = \varnothing$

B2. $E \subseteq F \rightarrow \mathsf{const}_i(E) \subseteq \mathsf{const}_i(F)$

Axiom schema B1 defines the value of const_i for the empty event set while B2 says that constituent functions are monotonic.

Chapter 4: A Prototype CI2 Theorem Prover

4.1. Overview

We describe here a working inference component for CI2. This implementation (approximately 3500 lines of Common Lisp) is not intended to be a model for practical applications. It exists merely to demonstrate the viability of CI2. Many of the techniques used by this program are fairly well-known. In this section, we will discuss the more interesting ones. A more detailed description of our theorem prover is given in the next section.

This program is essentially a traditional first-order theorem prover. In other words, it equates inference with deduction. Queries are answered by returning a list of unifying substitutions. A query can fail in one of two ways. The program may simply return an empty substitution list. Or the returned substitutions may each bind a query variable to the empty set. Note that these empty set bindings may be disguised as bindings to functional expressions denoting the empty set.

The program is guaranteed to halt and we guarantee the correctness of any substitution that binds all of its query variables to nonempty sets. We pay for these guarantees with the occasional "false negative." In other words, sometimes a query fails not because unifying substitutions do not exist, but because the program is incapable of finding them.

If a domain element is named by several constant terms, one is chosen to be the class' name. We keep track of equivalence classes of constant terms denoting the same domain element. Class names and the names of their elements are stored in hash tables and all predications are expressed in terms of these class names.

The axioms introduced in chapter 3 are procedurally encoded. Axioms defining event functions are instantiated when these functions are declared. For example, the event function eat-of from eaters to eating events would be declared as follows:

ev-fn (*eat-of*, (*eat*, 1))

In other words, eat-of is the event function for eat's first constituent.[1]

At this point, let us discuss the importance of monotonicity. The axioms for our special functions merely specify their boundary values and state that they are monotonic. From a proof-theoretic standpoint, this is all we know about them. A theorem prover can frequently take advantage of the monotonicity of these functions. For example, suppose we seek a set S which stands in a specified relationship with the set A. Suppose that R has this relationship with B and that $B \subseteq A$. Knowing this, we can restrict our search for S to supersets of R. If $A \subseteq B$, we would restrict our search to subsets of R. In other words, monotonicity enables us to bound our search.

For this reason, we associate with each variable its upper and lower bounds. This allows us to prune unwanted search paths. In chapter 3, we organized our set domain elements in a lattice whose maximal element is A, the set of all domain atoms, and whose

[1] These function declarations are extra-logical so we need not worry about their being second-order.

minimal element is ∅. Query variables that will be bound to sets (as opposed to numbers) initially have A and ∅ as their upper and lower bounds. A variable's upper bound is the constant term associated with the smallest known superset of the variable's ultimate value. Its lower bound is the constant term associated with its largest known subset. As retrievals progress, we update these bounds and any path requiring an out-of-bounds binding is discarded.[2] Updating lower (upper) bounds entails computing set unions (intersections). If we do not know of a constant term for the desired intersection (union), instead of creating one, we approximate the intersection (union) with the constant term denoting the largest (smallest) known named set smaller (larger) than it. The results of these computations are stored in hash tables for future reference.

Computing intersections and unions and determining subset relationships is facilitated by a hash table in which the smallest named supersets of each constant set term is stored. The results of previous subset queries are also stored in a hash table.

All variables are universally quantified. The need for existentially quantified variables is mitigated by our use of cardinality variable expressions which we will describe shortly. Other existential variables may be replaced by defining the appropriate Skolem functions. Variables are displayed as strings of alphanumeric characters prefixed with a question mark.

We introduce one new technique as well. To facilitate inference, implications of the form "If A is a subset of S of size i, then . . ." are transformed into *cardinality variable expressions* (or *CV expressions*). distributive readings. (See chapter 5, sentence [9].) If S is a set term and $E(x)$ is a well-formed formula with free variable x, then the cardinality variable expression $\mathsf{cv}_i(S)$ is defined by the following equivalence:

$$E(\mathsf{cv}_i(S)) \quad =_{def} \quad (?s \subseteq S \wedge \#?s = i) \rightarrow E(?s)$$

We will omit the subscript when $i = 1$.

Our theorem prover successfully made seventeen inferences corresponding to ten of the examples presented in the following chapter.[3] It took a Symbolics 3645 less than two minutes to make these inferences. Five of the inferences took less than one second.

4.2. In-depth View

4.2.1. Introduction

In this section, we give a detailed description of our theorem prover. As we noted in the previous section, we are equating inference with deduction here. Our program is essentially a traditional first-order theorem prover, but axioms must either be

---

[2] This technique is similar to the one described in [Frisch87].

[3] One example did not enable one to make any interesting deductions and the inferences associated with another example were too difficult.

predications or implications. Conjunctive axioms are translated into sets of axioms, one axiom per conjunct, but disjunctive axioms are not allowed.[4] In section 4.2.2, we describe how axioms are asserted.

Queries must be either predications, implications, or conjunctions. The process of answering a query is known as *retrieval*. If the retrieval is successful, the program returns a nonempty list of unifying substitutions. Applying any of these substitutions to the query results in a theorem. If the retrieval fails, the program returns the empty list. In section 4.2.3, we describe how queries are answered.

4.2.2. Making Assertions

Axiom assertion is conceptually straightforward. Suppose we wish to assert the statement A. We will assume that A is an implication or predication. (Conjunctions can be asserted one conjunct at a time.) First, we create equivalence classes (if necessary) for the constant expressions in A. Let these class names be a_n and b_n respectively. Each class is named after one of its members. The program tries to assign each class the shortest name it can.

Information about equivalence classes is stored in three hash tables. One table maps constant expressions to class names while a second one maps class names to class members. If A is an equality predication, the program merges a_n and b_n. The name of the equivalence class resulting from this merge will either be a_n or b_n. Suppose the program prefers b_n to a_n. We now replace all occurrences of a_n in our axiom set with b_n using a third hash table to find class names like 'f (a_n)' so that class name changes are propagated when classes are merged. In other words, the class named 'f (a_n)' would be renamed 'f (b_n)' and the appropriate substitutions would be made in the system's axiom set.

If A is a subset predication, the program makes sure that a_n lies below b_n in its representation of the domain lattice. This representation is kept in a set of three hash tables. One table maps a set to the named sets immediately above it in the lattice. A second table maps a set to the named sets immediately below it. And since subset queries are frequently posed, a third table maps ordered pairs of sets to Boolean values. For example, it would map (a_n, b_n) to t and (b_n, a_n) to *nil*.

If a or b contains or is a variable, or if A is an implication or a predication with a user-defined predicate, then we simply add it to the system's axiom set. (CV expressions are considered to be variable-containing.)

---

[4] Note that CI2 does not restrict one from using disjunctive axioms and we make use of several in the following chapter. If we allowed one to assert disjunctive axioms, our theorem prover would have to do some form of truth maintenance. Wherever we use a disjunction to represent a sentence in chapter 5, we assert its disjuncts one at a time. If we can make an inference regardless of the disjunct that was asserted, then one can safely conclude that this inference could be made by a more sophisticated theorem prover that has not committed itself to one or more of the disjuncts. So, by asserting disjuncts one at a time, we can demonstrate the viability of CI2 without implementing a truth-maintenance system.

There are two types of implications: forward-chaining rules and backward-chaining rules. Backward-chaining rules are used during retrievals whereas forward-chaining rules are used when assertions are being made. So, now that we have updated our hash tables and/or the system's axiom set, the program uses the algorithm described in the following section to retrieve applicable forward-chaining rules. If "$p \rightarrow q$" is a forward-chaining rule and θ is the substitution unifying A with p, then the program (recursively) asserts $q\theta$, i.e. the result of applying θ to q.

4.2.3. Query Retrieval

4.2.3.1. Retrieval as Tree Generation

Suppose we ask our theorem prover to retrieve Q. We will assume that Q is a conjunction of n conjuncts. (n may be one.) Our theorem prover attempts to find every possible proof for Q. It essentially does this by finding the proofs of the n-th conjunct that are compatible with the proofs it recursively found for the first n-1 conjuncts.

There are two ways of proving a conjunct. The simplest way is to "unify"[5] it with a system axiom. The resulting substitution when applied to the conjunct yields a theorem in the system. The second way is to unify it with the consequent of a backward-chaining rule and then prove the result of applying the unifying substitution to the rule's antecedent.

The retrieval of Q can be viewed as a tree generation problem. We begin with a tree rooted at a node representing the ordered pair (Q, *nil*). The tree grows as follows. First, we choose the "simplest" conjunct in Q. Suppose this conjunct is C. Roughly speaking, C is the conjunct that is easiest to *disprove*. We process conjuncts that have no variables first, especially equality predications because these can be proved simply by performing equivalence-class hash table lookups. If C is Q's n-th conjunct and it is a false constant equality predication, we could waste a lot of time generating proofs for the first n-1 conjuncts; our simplest-first strategy attempts to minimize the amount of time spent on proofs that are doomed to fail.

Another simple conjunct is a subset predication having at least one variable argument. This conjunct is simple not because it is easy to disprove, but because it can be proved simply by computing unions or intersections as described in chapter 4. In order to make these computations as fast as possible, we store each result in a hash table.

An example of a "hard" conjunct is an equality predication having one variable argument. This would be innocuous if it weren't for the fact that the variable argument enables many backward-chaining rules to be "fired". As we shall see, this conjunct would probably cause the tree to grow very bushy.

Processing C may cause one or more subtrees to be created. For example, if C unifies with the axiom A, we create a descendant node representing (Q', $\theta_{A,C}$) where

[5] We will explain these quotation marks in the next section.

$\theta_{A,C}$ is the substitution unifying A and C, and Q' is the result of applying $\theta_{A,C}$ to the conjunction of Q's remaining conjuncts. If a descendant node represents the empty conjunct list, it is marked "success".

If C unifies with the consequent of the backward-chaining rule R, we create a descendant node representing $(\overline{Q}, \theta_{R,C})$ where $\theta_{R,C}$ is the substitution unifying R's consequent and C, and $\overline{Q}$ is the result of applying $\theta_{R,C}$ to the conjunction of Q's remaining conjuncts and the conjuncts one gets by applying $\theta_{R,C}$ to R's antecedent.

Each rule antecedent conjunct is tagged with the rules that were used to generate it. In other words, the R-antecedent conjuncts get C's rule tags (if any) along with a tag for R. Our program will not use R to prove an R-tagged conjunct. This admittedly crude strategy prevents rules from being fired ad infinitum and is also the reason why our program sometimes fails to deduce theorems. In other words, we pay a price for decidability, namely completeness.

If we are unable to unify C with any of our axioms or rule consequents, the node is marked "failure".

We repeat this process for each leaf node that is neither a success node nor a failure node until each leaf node is either a failure node or a success node. At this point, each success node corresponds to a proof of Q. If S is a success node, then merging the substitutions associated with the nodes on the path from S to the root yields θ_S, the substitution for the proof associated with S. (Our implementation actually merges the substitutions as it creates the descendant nodes. This technique saves us memory because we no longer need to store the tree's interior nodes.) $Q\,\theta_S$, i.e. the result of applying the substitution associated with S to the query, is a theorem of the system.

4.2.3.2. "Unification"

If it were not for the complications described in this section, all proofs would be based solely on unification and *modus ponens*. Until now, all we have described is an extension of a conventional first-order theorem prover. We have added hash tables representing our domain lattice and these tables clearly facilitate the retrieval of subset predication conjuncts. This section describes what is not so obvious, i.e. how these tables complicate retrieval.

Let f be an event function mapping b onto e. So the equivalence class e contains $f(b)$ as one of its members. Suppose that we have also asserted "$a \subseteq b$", that we are attempting to retrieve "$(?x \subseteq e) \wedge \cdots$", and that the substitution we seek is $\{?x \leftarrow f(a)\}$.

In order to deduce "$f(a) \subseteq e$", we need to make use of procedurally encoded axioms about the equality and subset relationships and about event functions. We make this deduction as follows. When we process the conjunct "$(?x \subseteq e)$", we *conceptually* unify this predication with our axiom set (among other things). However, information about subset relationships is stored in the domain lattice hash tables so we simulate this unification as follows.

First, we get e's class members from an equivalence class hash table. Then, we create subtrees for each of the class members. Of particular interest here is the subtree corresponding to $f(b)$. This subtree corresponds to the substitution $\{?x \leftarrow f(?y)\}$ where $?y$'s upper bound is b. This binding of $?x$ enables our program to discover the desired substitution. If we had also asserted "$g(h(c)) = e$", we would also have two subtrees corresponding to $g(h(c))$. One subtree would correspond to the substitution $\{?x \leftarrow g(?u)\}$ where $?u$'s upper bound would be $h(c)$. The other subtree would correspond to $\{?x \leftarrow g(h(?v))\}$ where $?v$'s upper bound would be c.

These *weakened* bindings are generated whenever we conceptually unify expressions that are in a *subset context*. In this example, $?x$ is in a subset context because it is the subset argument of a subset predication. The superset argument of a subset predication is also in a subset context. For example, if we were to unify "$f(a) \subseteq ?z$" with "$f(a) \subseteq e$", we would generate subtrees corresponding to the substitutions $\{?z \leftarrow f(?r)\}$, $\{?z \leftarrow g(?s)\}$, and $\{?z \leftarrow g(h(?t))\}$ (among others). $?r$'s lower bound would be b, $?s$'s lower bound would be $h(c)$, and $?t$'s lower bound would be c.

Note the importance of event function monotonicity here. $f(a)$ is a subset of e only because the event function f is monotonic. Also note that event function monotonicity guarantees that weakened bindings will not cause our program to discover invalid proofs.

Chapter 5: Examples

5.1. Introduction

In this chapter, we demonstrate CI2's viability as a representation language. Twelve examples are presented in increasing order of complexity. We give a representation for each example sentence, and show how important inferences can be made. All but one of the inferences in this chapter were made by the CI2-specific theorem prover described in the preceding chapter using the domain-independent rules listed in Appendix B. We also demonstrate the viability of our disambiguation strategy by presenting statements that can be asserted in order to constrain the initial representation once the intended reading has been determined.

5.2. One Plural Noun Phrase

When a sentence only has one plural noun phrase, its sum-of-plurals reading degenerates to a collective reading. We still have the potential for ambiguity, i.e. the sentence still has collective and distributive readings. Consider for example:

[1] John dunked Mary's braids in the inkwell. ([Webber79])

John did the dunking and Mary's braids were dunked, but *how* did he dunk them? Perhaps he did so one at a time, perhaps all at once. The following representation captures this ambiguity:

; dunk = all dunking events.
; $const_1$(dunk) = the ones doing the dunking.
; $const_2$(dunk) = the things that were dunked.

dunk-1 $\subseteq$ dunk	; [1]'s dunking events.
$const_1$(dunk-1) = john-1	; john-1 = John
john-1 $\subseteq$ man	; man = all men
#john-1 = 1	
$const_2$(dunk-1) $\subseteq$ braid	; braid = all braids
#$const_2$(dunk-1) > 1	
$const_3$(dunk-1) $\subseteq$ inkwell	; inkwell = all inkwells
#$const_3$(dunk-1) = 1	

Though our intuition fails to disambiguate this sentence, we are still able to make the important inferences, i.e. we can still infer that Mary's braids were dunked and that John did the dunking. For example, the query "did John dunk anything in an inkwell?" would be represented as:

```
?dunk ⊆ dunk ∧                  ; Is there a set of dunkings
 const1(?dunk) = john-1 ∧       ;   in which John dunked something
 const3(?dunk) ⊆ inkwell        ;   in an inkwell?
 #const3(?dunk) = 1
```

Our theorem prover responds by returning the unifying substitution: {?dunk ← dunk-1}.

If we somehow learn that Mary's braids were all dunked at once, we can constrain our representation by asserting "#dunk-1 = 1". The case in which the braids were dunked one at a time can be represented by asserting "$\#const_2(cv(dunk\text{-}1)) = 1$". This statement says that the dunked constituent of each event in dunk-1 is a unit set.

We can also represent more complicated cases. For example, suppose John dunked two of Mary's braids at one point and her third and remaining one at some later point. We can represent this information as follows:

```
ev-fn(dunk-of, dunk, 2)                          ; dunk-of maps dunked things
                                                 ;   to their dunking events.

#dunk-of(braid-2) = 1                            ; First he dunked two,
braid-2 ⊆ const2(dunk-1) ∧ #braid-2 = 2
#dunk-of(braid-3) = 1                            ;   then he dunked the last one.
braid-3 ⊆ const2(dunk-1) ∧ #braid-3 = 1

#braid-1 = 3                                     ; braid-1 is the disjoint union
cv(braid-1) ⊆ braid-2 ∨ cv(braid-1) ⊆ braid-3    ;   of braid-2 and braid-3.[1]
```

The declaration of dunk-of should be viewed as being a part of the definition of dunk and not part of our representation. In fact, event function declarations are only one class of statements that tend to be shared by sentence representations. The representation of any noun phrase denoting the same set in two or more sentence readings would also be shared. A metric of representational size should take this sharing into account.

Now, let us turn to an example where we do have disambiguating information.

[2] Beat egg yolks until thick and lemon-colored.
(*New York Times*)

[1] This disjunction is needed to insure that braid-2 and braid-3 represent disjoint sets of braids.

Our representation:

; beat = all cooking-related beating events.
; $const_1$(beat) = the food being beaten.

beat-2 $\subseteq$ beat	; [2]'s beating events
$const_1$(beat-2) $\subseteq$ yolk	; yolk = all egg yolks
#$const_1$(beat-2) > 1	

Shall we beat them all at once, or one at a time? Any cook knows that only one beating event is called for. In order to represent this knowledge, we need a representation of cooking events, e.g:

; cook = all cooking events.
; $const_1$(cook) = raw food constituent.
; $const_2$(cook) = cooking task constituent.
; $const_3$(cook) = cooked food constituent.

; cook-ev maps cooking tasks to their cooking events.

ev-fn(cook-ev, cook, 2)

Now we can express our disambiguating axiom as follows:

?e $\subseteq$ beat $\wedge$ $const_1$(?e) $\subseteq$ yolk $\wedge$	; If ?e is a set of yolk-beating events
#cook-ev(?e) = 1 $\rightarrow$	; associated with one cooking event,
#?e = 1	; then it is a unit set.

This axiom allows allowing us to deduce #beat-2 = 1. It also prevents one from making any inferences that depend on beat-2 being a non-unit set.

Sentences with one plural noun phrase can be more complicated. For example:

[3] At the Paris zoo, Bruce saw a lion, a tiger, a giraffe, a hippopotamus and an elephant. ([Webber79])

; see = all seeing events.
; $const_1$(see) = those doing the seeing.
; $const_2$(see) = what was seen.
; $const_3$(see) = where the seeing was done.

see-3 $\subseteq$ see	; [3]'s seeing events
$const_1$(see-3) = bruce-3	; bruce-3 = Bruce
bruce-3 $\subseteq$ man	; man = all men
#bruce-3 = 1	
$const_2$(see-3) $\subseteq$ animal	; animal = all animals
lion-3 $\subseteq$ lion $\wedge$ lion-3 $\subseteq$ $const_2$(see-3)	; lion = all lions

#lion-3 = 1	; lion-3 = a lion
tiger-3 ⊆ tiger ∧ tiger-3 ⊆ $const_2$(see-3)	; tiger = all tigers
#tiger-3 = 1	; tiger-3 = a tiger
giraffe-3 ⊆ giraffe ∧ giraffe-3 ⊆ $const_2$(see-3)	; giraffe = all giraffes
#giraffe-3 = 1	; giraffe-3 = a giraffe
hippo-3 ⊆ hippo ∧ hippo-3 ⊆ $const_2$(see-3)	; hippo = all hippopotami
#hippo-3 = 1	; hippo-3 = a hippopotamus
elephant-3 ⊆ elephant ∧ elephant-3 ⊆ $const_2$(see-3)	; elephant = all elephants
#elephant-3 = 1	; elephant-3 = an elephant
$const_3$(see-3) = paris-zoo	; paris-zoo = the Paris zoo
paris-zoo ⊆ zoo	zoo = all zoos
#paris-zoo = 1	

This sentence does not specify how many seeing events there are. We know that animals are segregated in zoos so that there must have been one seeing event per animal. We would like to compute the number of seeing events by counting the number of animal types that were seen. We cannot do this in CI2 though we can simulate it by painfully adding axioms to this effect.

Let us look at this simulation to see just how painful it is. First, let lion-of(A) be the set of lions in A. Formally, lion-of would be defined by the following axiom:

$$?x \subseteq \text{lion} \wedge ?x \subseteq ?s \leftrightarrow ?x \subseteq \text{lion-of}(?s)$$

We would define functions for other animal types in the same manner. Now we can use these functions to create "counting axioms", e.g.:

$$?s \subseteq \text{see} \wedge const_3(?s) \subseteq \text{zoo} \wedge$$
$$\#\text{lion-of}(?s) > 0 \wedge \#\text{hippo-of}(?s) > 0 \wedge$$
$$\#\text{tiger-of}(?s) = 0 \wedge \#\text{giraffe-of}(?s) = 0 \wedge$$
$$\#\text{elephant-of}(?s) = 0 \wedge \cdots \rightarrow \#?s > 1$$

In other words, if ?s is a set of zoo-seeing events in which only lions and hippopotami were seen, then ?s contains at least two events.

Our simulation reveals two shortcomings:

- One cannot count efficiently in CI2 (or in any logic for that matter).
- We would not have to define lion-of *et al.* if set intersection was a primitive in our language.

Of course, one would never implement this simulation in practice. Yet, even without these "counting axioms", we can still make the important inferences, i.e. we can still infer that Bruce saw animals at the Paris zoo and that these animals included a lion, a tiger, a giraffe, a hippopotamus, and an elephant.

The sentence that Scha uses to argue for his multi-level plural representations is:

[4] The juries and the committees gather.

[Scha88] claims that this sentence only has three readings:

> One in which each of the juries gathers alone and each of the committees gathers alone as well (distribution over two levels), another in which all persons who are committee members gather in one place and all persons who are jurors gather in another place (distribution over one level), and finally a third in which all jurors and committee members unite in one large convention (completely collective). It seems inescapable, therefore, that the internal multi-level structure of NPs has to be preserved.

We are unhappy with this analysis for several reasons. First, although we do prefer readings in which no committee or jury is split up among two or more gatherings, we see no reason to limit ourselves to the Scha's three readings. We have no objection to a reading in which there were two jury gatherings and two committee gatherings. Nor do we think it advisable to encode mere preferences in our logic. If we do, we will be unable to represent extenuating circumstances, e.g. a case in which the women members gathered in one place leaving the men to gather haphazardly. Preferences will have to be represented outside of our logic, e.g. by embedding our logic in a probabilistic system.

[Scha88] seems to equate the extension of 'the committees' with the extension of 'the members of the committees.' We note that a committee has properties that one would not attribute to its members, e.g. date of formation.[2] So we will differentiate committees (and juries) from their members and represent [4] as follows:

; gmember = all group membership relations.
; $const_1$(gmember) = the groups.
; $const_2$(gmember) = the members.

ev-fn(member-of, gmember, 1) ; maps groups to their members

gmember-4 ⊆ gmember ; for [4]'s committees
$const_1$(gmember-4) ⊆ committee ; committee = all committees
#$const_1$(gmember-4) > 1

gmember-5 ⊆ gmember ; for [4]'s juries
$const_1$(gmember-5) ⊆ jury ; jury = all juries
#$const_1$(gmember-5) > 1

gather-4 ⊆ gather ; gather = all gatherings
$const_2$(gmember-4) ⊆ $const_1$(gather-4)
$const_2$(gmember-5) ⊆ $const_1$(gather-4)

[2] This difference might be attributed to the committee's being an intensional entity.

Our distinction between committees and their members comes into play when we express our preference for readings that leave the gathered committees and juries intact. Let PCI2 be a probabilistic system for representing this sort of preference. We might express our preference in PCI2 as follows:

If ?g ⊆ gather ∧ #?g = 1 ∧

(?s ⊆ committee ∨ ?s ⊆ jury) ∧ #?s = 1 ∧

?t ⊆ $const_1$(?g) ∧ ?t ⊆ $const_2$(gmember-of(?s)) ∧ #?t > 0

Prefer $const_2$(gmember-of(?s) ⊆ $const_1$(?g)

In other words, if ?g is a gathering event, and ?s is a committee or jury, and ?g's gatherers intersect ?s's members, then we prefer readings in which ?g's gatherers include all of ?s's members.

We cannot make any interesting inferences here unless we know something about the gatherings. We can only infer that if M is a set of committee or jury members, then there is a set of gathering events whose gatherers are a superset of M. In other words:

?m ⊆ $const_1$(gather-4) →

(E ?g) (?g ⊆ gather ∧ ?m ⊆ $const_1$(?g))

This is not very interesting since the conditional will always be true if we bind ?g to gather-4.

5.3. 1½ Plural Noun Phrases

Sometimes a sentence refers to one or two plural groups depending on how it is interpreted. Such a sentence always contains an unambiguously plural noun phrase and an existentially quantified singular one. The sentence's ambiguity is usually attributed to competing quantifier scopings. The sentence has one plural noun phrase if the existentially quantified noun phrase has wide scope. It has two plurals if the unambiguous plural has wide scope. The classic example:

[5] Every man loves some woman.

Sentence [5]'s readings are traditionally represented as follows:

[5a] $(\mathrm{E}\, w)(m)\ (w \in \text{women} \wedge (m \in \text{men} \rightarrow \text{Love}(m, w)))$
There is a unit set of women. Every man loves the woman in this set.

[5b] $(m)(\mathrm{E}\, w)\ (m \in \text{men} \rightarrow (w \in \text{women} \wedge \text{Love}(m, w)))$
There is a set of women. Every man loves a woman in this set.

These two readings are extensionally equivalent if the set of women in the second reading is a unit set. In other words, [5b] subsumes [5a] and we can represent [5] without disjunctions as follows:

; love = all lovings.
; $const_1$(love) = the ones doing the loving.
; $const_2$(love) = the ones that are loved.

ev-fn(love-of, love, 1) ; maps lovers to love events.

love-5 $\subseteq$ love ; [5]'s lovings
$const_1$(love-5) = man
$const_2$(love-5) $\subseteq$ woman ; woman = all women
#$const_2$(love-of(cv(man))) = 1

Instead of having to cope with the competing quantifier scopings of the traditional representation, CI2 makes use of the subsumption of one reading by another to enable us to tersely represent the ambiguity. ''#$const_2$(love-of(cv(man))) = 1'' specifies that each man in love-1 loves exactly one woman (because 'every man' and 'some woman' are both singular). We have not excluded the possibility of two men loving the same woman. Nor have we excluded the possibility of there being other loving events.

How the intended reading is determined is a problem which we will not address. We might have reason to believe that woman-5 is a unit set of women and conclude that [5a] was intended. In this case, we would amend our representation by asserting ''#$const_2$(love-5) = 1''. On the other hand, we might decide that $const_2$(love-5) is a non-unit set of women. In this case, we can amend our representation by asserting something like ''#$const_2$(love-5) > 1''.

Even before disambiguation, if we have:

john-5 $\subseteq$ man $\wedge$ #john-5 = 1 ; john-5 is a man.

we will be able to infer that john-5 loves some woman. We would submit the following query:

?love $\subseteq$ love $\wedge$; Is there a set of lovings
$const_1$(?love) = john-5 ; in which John loved someone?

Our theorem prover binds ?love to $const_2$(love-of(john-5)). The rule we used to make this deduction insures that this does not denote the empty set. (The nonemptiness of $const_2$(love-of(john-5)) can also be verified by submitting the query ''#$const_2$(love-of(john-5)) > 0''.

In sentence [5], the unambiguously plural noun phrase is universally quantified. Now let us look at an example in which the unambiguous plural is numerically quantified.

[6] Four men lifted a piano. ([Roberts87])

Strictly speaking, [6] has four readings:

[6a] distributive
There are four men. There are as many as four pianos. Each man lifted one of these pianos.

[6b] collective A
There are four men. There is one piano. The four men lifted this piano together.

[6c] collective B
There are *n* groups of men. Their union has cardinality 4. There are as many as *n* pianos. Each group of men lifted one of these pianos.

[6d] sum-of-plurals
There are *n* groups of men. Their union has cardinality 4. There is one piano. Each group of men lifted this piano.

As was the case with sentence [0] (analyzed in section 1.2), the collective-B reading subsumes the distributive and collective-A readings. In this case, because 'a piano' is singular, it subsumes the sum-of-plurals reading as well. So we can represent [6] without disjunctions as follows:

; lift = all lifting events.
; $const_1$(lift) = the lifters.
; $const_2$(lift) = the lifted.

ev-fn(lift-of, lift, 1)	; lifters to lift events.
lift-6 $\subseteq$ lift	; [6]'s lifting events
$const_1$(lift-6) $\subseteq$ man	; man = all men
#$const_1$(lift-6) = 4	
$const_2$(lift-6) $\subseteq$ piano	; piano = all pianos
#$const_2$(cv(lift-6)) = 1	; one piano at a time

If we learn that the four men have superhuman strength, we might decide that they each lifted the piano to test their strength. In other words, if we learn that the distributive reading was intended, we can modify our representation by asserting ''#$const_1$(cv(lift-6)) = 1''. This statement says that the lifter constituent of each event in lift-6 is a unit set.

If we determine that nothing abnormal transpired, i.e. that the four men cooperated to lift the piano, we can indicate that the first collective reading was intended by asserting ''#lift-6 = 1''. If we discover that some subset of the four men lifted the piano, we might prefer the second collective reading and assert ''#lift-6 > 1''. And if we decide that only one piano was involved, we can indicate our preference for the sum-of-plurals reading by asserting ''#$const_2$(lift-6) = 1''.

Suppose 'man-6' denotes a proper subset of $const_1$(lift-6). Then 'lift-of(man-6)' denotes the lifting events in which man-6 or one of its subsets was the lifter. Once again, we have no guarantee that this denotes a nonempty set. This is where disambiguating assertions come into play. For example, we can deduce lift-of(man-6) = ∅ from #lift-6 = 1.

5.4. Two Plural Noun Phrases

Our next example is almost identical to sentence [0] which was discussed at length in chapter 1.

[7] 600 Dutch firms have 5000 American computers. ([Scha81])

; own = all owning events.
; $const_1$(own) = the owners.
; $const_2$(own) = the property.

ev-fn(own-of-1, own, 1)	; maps owners to own events
ev-fn(own-of-2, own, 2)	; maps property to own events
own-7 ⊆ own	; [7]'s owning events
$const_1$(own-7) ⊆ df #$const_1$(own-7) = 600	; df = all Dutch firms
$const_2$(own-7) ⊆ ac	; ac = all American computers
#$const_2$(own-7) = 5000 ∨	; sum-of-plurals[3]
#$const_2$(own-of–1($const_1$(cv(own-7)))) = 5000	; collective-distributive[4]

In chapter 1, we showed how eight of sentence [0]'s collective-distributive readings were subsumed by a single reading, [R9]. This enabled us to compactly represent [0] with the disjunction of [R9] and its sum-of-plurals reading, [R10]. We have taken the same approach here.

If we adopt the convention that entities are always owned singly and enforce this convention with the axiom "#$const_2$(cv(own)) = 1", we will no longer be able to retrieve any reading which interprets the computers collectively. If we then determine that American computers are never owned by more than company and represent this fact with "#$const_1$(own-of-2(cv(ac))) = 1", we will only be able to retrieve the following readings:[5]

---

[3] Sentence [7]'s sum-of-plurals reading refers to a total of 5000 computers.

[4] Sentence [7]'s collective-distributive readings associate 5000 computers with each owner constituent.

[5] [A] corresponds to [R1] and [B] corresponds to [R10].

[A] firms: distributive, computers: distributive.
There are 600 firms. There are as many as 600 sets of computers, each having cardinality 5000.† Each of the firms has each of the computers in one of these sets.

[B] sum-of-plurals.
There are n groups of firms. The cardinality of the union of these n groups is 600. There are m groups of computers. The cardinality of the union of these m groups is 5000. Each group of firms collectively has (owns) at least one group of computers and each group of computers is had (owned) by at least one group of firms.

Our disambiguating assertions constrain these readings as well. Since each firm in $const_1$(own-7) is now constrained to be a constituent of at least one of own-7's elements, then each firm in $const_1$(own-7) must own at least one computer. In other words, we can deduce #$const_2$(own-of-1(cv($const_1$(own-7)))) > 0 without committing ourselves to [A] or [B]. Also, now that we know that computers are each owned by one company, we know that [A] entails #$const_2$(own-7) = 3,000,000 and we know that m = 5000 and n = 600 in [B]. Presumably, we could now appeal to company records and deduce the impossibility of [A].

Here is an example with definite plurals:

[8] The squares contain the circles. ([Scha81])

We can represent [8] without disjunctions because there is so little cardinality information. Our representation:

; contain = all containments.
; $const_1$(contain) = the containers.
; $const_2$(contain) = the contained things.

ev-fn(contain-of, contain, 2) ; contained things to contain events

contain-8 ⊆ contain ; [8]'s containments

$const_1$(contain-8) ⊆ square ; square = all squares
#$const_1$(contain-8) > 1

$const_2$(contain-8) ⊆ circle ; circle = all circles
#$const_2$(contain-8) > 1

† The sets may intersect.

There is only one inference here: if circle-9 is one of the circles, i.e. $\text{circle-9} \subseteq \text{const}_2(\text{contain-8}) \wedge \#\text{circle-9} = 1$, then ?s is a square containing circle-9. Our query is:

$?c \subseteq \text{contain} \wedge$; Is there a set of containment events
$\text{const}_2(?c) = \text{circle-9}$; in which circle-9 is contained?

Our theorem prover responds by binding ?c to contain-of(circle-9). If we adopt the convention that things are always contained singly, i.e. $\#\text{const}_2(\text{cv}(\text{contain})) = 1$, we can then deduce that contain-of(circle-9) is a nonempty containment event set and that $\text{const}_1(\text{contain-of(circle-9)})$ must be a nonempty (possibly non-unit) set of squares.

Here is an example with an enumerated plural:

[9] Chris, Leslie, and Robin ate four hamburgers.

; eat = all eating events.
; $\text{const}_1(\text{eat})$ = the eaters.
; $\text{const}_2(\text{eat})$ = the edibles.

$\text{disjoint}(\text{eat}, 2)$	; Food is eaten once.[6]
$\text{eat-9} \subseteq \text{eat}$	; [9]'s eating events
$\text{const}_1(\text{eat-9}) = \text{clr}$	; Chris, Leslie, and Robin
$\text{const}_2(\text{eat-9}) \subseteq \text{hamburger}$	; hamburger = all hamburgers
$\text{ev-fn}(\text{eat9-of}, \text{eat}, 1)$	; eaters to eat-9 events
$\text{chris} \subseteq \text{clr} \wedge \#\text{chris} = 1$	; clr is the
$\text{leslie} \subseteq \text{clr} \wedge \#\text{leslie} = 1$	; disjoint union
$\text{robin} \subseteq \text{clr} \wedge \#\text{robin} = 1$	; of chris, leslie,
$\#\text{clr} = 3$	; and robin.[7]
$\text{cv}(\text{clr}) \subseteq \text{chris} \vee \text{cv}(\text{clr}) \subseteq \text{leslie} \vee \text{cv}(\text{clr}) \subseteq \text{robin}$	
$\#\text{const}_2(\text{eat-9}) = 4 \vee$	; sum-of-plurals
$\#\text{const}_2(\text{eat9-of}(\text{const}_1(\text{cv}(\text{eat-9})))) = 4$	; collective-distributive[8]

---

[6] $\text{disjoint}(e, i)$ is syntactic sugar for:

$?s \subseteq e \wedge \#?s = 1 \wedge$
$?t \subseteq e \wedge \#?t = 1 \wedge$
$?s \neq ?t \wedge$
$?r \subseteq \text{const}_i(?s) \wedge$
$?r \subseteq \text{const}_i(?t) \rightarrow ?r = \varnothing$

In other words, each of e's i-th constituents belongs to exactly one of e's i-th constituents.

[7] This disjunction is needed to insure that chris, leslie, and robin represent different people.

[8] Sentence [9]'s collective-distributive readings associate four hamburgers with each eater constituent.

As was the case with [7], sentence [9]'s ambiguity is reflected in a disjunctive assertion. Suppose we somehow learn that the sum-of-plurals reading was intended. We can add this information to our data base by asserting "#const$_2$(eat-9) = 4". Now, if we ask if Chris ate a hamburger, i.e. if #const$_2$(eater1-ev(chris)) > 0, we will get a "negative" response, i.e. our theorem prover will fail to prove this inequality because #const$_2$(eater1-ev(chris)) = 0 is consistent with what we know. If Chris, Leslie, and Robin each ate one-third of each hamburger, then we would have #const$_2$(eater1-ev(leslie)) = 0 and #const$_2$(eater1-ev(robin)) = 0 as well.

In this example, the sum of plurals reading does not offer us many interesting inferences unless we have additional information constraining the mapping between clr and const$_2$(eat-9). For example, we might learn that Chris ate two hamburgers and infer that Leslie and Robin ate the other two. We would need a more sophisticated theorem prover to be able to efficiently take advantage of such information too. (We might be able to make do by modifying our axioms, but a better approach would be to procedurally encode some simple axioms about natural numbers.)

Suppose we somehow learn that no hamburgers were shared and that a collective-distributive reading was intended. If we add this information by asserting "#const$_2$(eat9-of(cv(clr))) = 4", i.e. each member of clr ate four hamburgers, we would then be able to deduce that Chris ate exactly four hamburgers and that a total of twelve hamburgers were eaten.

So far, we have used CV expressions on a number of occasions, but their value has not been obvious. At this point, we can demonstrate how valuable they can be. Suppose we submit the query "who ate four hamburgers?":

?e $\subseteq$ eat $\wedge$	; Is there a set of eating events
const$_1$(?e) = ?p $\wedge$	; in which someone ate
const$_2$(?e) $\subseteq$ hamburger $\wedge$ #const$_2$(?e) = 4	; four hamburgers?

We get the answer {?e $\leftarrow$ eat9-of(cv(clr)), ?p $\leftarrow$ cv(clr)}, i.e. each unit subset of clr ate four hamburgers. The alternative, namely returning chris, leslie, and robin instead, would not be unreasonable in this case, but consider the case where #clr = 30. Suppose we only have names for 10 of these people. If we did not have CV expressions at our disposal, we would have to generate a constant for each unnamed unit person set and then return these 20 constants along with the 10 we already have. We would also want to return a statement saying that these constants denote *disjoint* unit sets. Returning a cardinality variable expression is clearly the superior alternative.

5.5. More Complicated Examples

In this section we present examples that are complicated by a secondary relationship. Our first sentence is from [Hobbs83]:

[10] Every representative of a company arrived.

The primary relationship in [10] is between representatives and arrivals. The secondary relationship is between the representatives and one or more companies. The traditional representations for [10]'s two readings are:[9]

[10a] $(\mathrm{E}\, c)\,(r)\,(\mathrm{Company}\,(c) \wedge \mathrm{Represents}\,(r, c) \rightarrow \mathrm{Arrive}\,(r))$
There is a unit set of companies. Every representative of the company in this set arrived.

[10b] $(c)(r)\,(\mathrm{Company}\,(c) \wedge \mathrm{Represents}\,(r, c) \rightarrow \mathrm{Arrive}\,(r))$
Every representative of every company arrived.

We represent the sentence as follows:

; arrive = all arrivals.
; const_1(arrive) = the arrivers.

; employ = all employings.
; const_1(employ) = the employers.
; const_2(employ) = the employees.

ev-fn(employ-of, employ, 1)	; employer to employing event
employ-10 $\subseteq$ employ	; [10]'s employings
const_1(employ-10) $\subseteq$ company	; company = all companies
const_2(employ-10) = rep-10	; the representatives
arrive-10 $\subseteq$ arrive	; [10]'s arrivals
const_1(arrive-10) = rep-10	; the arrivers = the representatives
#const_1(employ-10) = 1 $\vee$	; [10a]
const_1(employ-10) = company	; [10b]

Suppose we know that Chris is a representative of company-11 and we query "Chris $\subseteq$ rep-10", i.e. we ask if Chris arrived. We can only deduce Chris' arrival if we know that company-11 $\subseteq$ company-10, i.e. if the second reading was intended or if we happen to know that company-10 = company-11.

---

[9] These are very similar to the traditional representations for sentence [5]'s two readings.

Usually the secondary relationship complicates matters a great deal. For example:

[11] The girls' fathers bought them cars. ([Roberts87])

Let us represent the father relation as follows:

ev-fn(father-of, father, 2)	; child to father relation
$const_1$(father) ⊆ man	; fathers are men
$const_2$(father) = person	; everybody has a father

Now we can represent [11] as follows:

; buy = all buying events.
; $const_1$(buy) = the buyers.
; $const_2$(buy) = the purchases.
; $const_3$(buy) = the recipients.

buy-11 ⊆ buy	; [11]'s buying events
$const_1$(buy-11) = $const_1$(father-of($const_3$(buy-11)))	; the buyers = the fathers
$const_2$(buy-11) ⊆ car	; car = all cars
#$const_2$(buy-11) > 1	; 'cars' is plural
$const_3$(buy-11) ⊆ girl	; girl = all girls
#$const_3$(buy-11) > 1	; 'girls' is plural

We might infer from [11] that each of the fathers bought one or more cars for (some of) his daughters though there is nothing in [11] that precludes one of the fathers from buying a car for someone else's daughter. Perhaps this inference is partly due to a "rule" that says that car recipients are members of the buyer's family. The simplest ways of encoding this information either entail using rules that contain existential variables or allowing families to be domain elements. We will take the second approach and assume that the fmember relationship relates families with their members.[10] We will need the following statements:

ev-fn(fmember-of, fmember, 2)	; family members to family-member relationships
$const_1$(fmember) = family	; family = all families
$const_2$(fmember) = person	; person = all people
#$const_2$(cv(fmember)) = 1	; relate families with individual members

This last statement enforces a convention about fmember relationships. This convention insures that 'fmember-of(P)' does not denote the empty set if 'P' denotes a nonempty subset of person. We can now assert our "rule" as follows:

[10] This is reminiscent of our treatment of committees in example [4].

$const_2(cv(buy)) \subseteq car \rightarrow$

$const_3(cv(buy)) \subseteq const_2(fmember\text{-}of(const_1(cv(buy)))) \wedge$

$\#const_1(fmember\text{-}of(const_1(cv(buy)))) = 1$

In order to infer that each father bought at least one car, we also tacitly assumed that each family has only one father. (If a family has two fathers, they could jointly buy their daughters' car(s) in which case neither father would have bought a car.) We can encode this assumption as follows:

$const_1(fmember\text{-}of(?p)) = const_1(fmember\text{-}of(?q)) \wedge$	; If ?p's family = ?q's family, and
$const_1(father\text{-}of(?p)) \subseteq const_2(fmember\text{-}of(?p)) \wedge$	; ?p's father is in ?p's family,
$const_1(father\text{-}of(?q)) \subseteq const_2(fmember\text{-}of(?q)) \wedge$	; and ?q's father is in ?q's
$\#const_1(fmember\text{-}of(?p)) = 1 \rightarrow$	; family, then
$const_1(father\text{-}of(?p)) = const_1(father\text{-}of(?q))$	; ?p and ?q have the same father.

Now we can infer that each father bought at least one car.‡

A sentence containing two relationships can be complicated by the presence of additional cardinality information. For example:

[12] Five insurance associates gave a $25 donation to several charities. ([Roberts87])

We will simplify matters by letting 'S' denote 'several'. Our representation:

; give = all giving events.
; $const_1(give)$ = the givers.
; $const_2(give)$ = the gifts.
; $const_3(give)$ = the recipients.

ev-fn(give-of-1, give, 1)	; givers to giving events
ev-fn(give-of-2, give, 2)	; gifts to giving events
ev-fn(give-of-3, give, 3)	; recipients to giving events
disjoint(give, 2)	; Gifts are given once.[11]
no-duplicates(give, 1, 3)	; [12]
give-12 $\subseteq$ give	; [12]'s giving events
$const_1(give\text{-}12) \subseteq person$	; person = all people
$\#const_1(give\text{-}12) = 5$	

‡ Unfortunately, our prototype theorem prover is not powerful enough to make this deduction.

[11] See sentence [9], footnote 6.

$const_2(give\text{-}12) \subseteq money$	; money = all money
$\#const_2(cv(give\text{-}12)) = 25$	; $25 per donation
$const_3(give\text{-}12) \subseteq charity$	; charity = all charities
$\#const_3(give\text{-}12) = S \vee$	; several charities in all or
$\#const_3(give\text{-}of\text{-}1(cv(const_1(give\text{-}12)))) = S$	; several per associate
$\#give\text{-}12 = 1 \vee$	; one donation in all or
$\#give\text{-}of\text{-}1(cv(const_1(give\text{-}12))) = 1 \vee$	; one per associate or
$\#give\text{-}of\text{-}3(cv(const_3(give\text{-}12))) = 1$	; one per charity

We begin by tackling the problem of determining the possible ways of assigning the cardinality information in [12]. Clearly, we have five associates and $25 per donation. What is not so clear is the number of charities and the number of donations. There are two ways of interpreting 'several' here. There might be several charities in all, or several per associate. There are three ways of interpreting the indefinite 'a' here. There might be one donation in all, one per associate, or one per charity.

Our tax codes give us reason to believe that each donation is being given by a single associate, i.e. "$\#const_1(cv(give)) = 1$". This axiom rules out any reading in which the number of donations is less than five. We will also assume that all donations have a single recipient, i.e. "$\#const_3(cv(give)) = 1$". This axiom rules out any reading in which the number of donations is greater than the number of charities. Taken together, these two axioms limit us to the following cardinality assignments:

[12a] one donation per associate, several (at most five) charities in all.

[12b] one donation per charity, several (at least five) charities in all.

[12c] one donation per charity, several charities per associate.

There also appears to be a reading which ignores the cardinality information supplied by the indefinite 'a':

[12d] several charities per associate.

---

[12] no-duplicates(e, i, j) is syntactic sugar for: $?s \subseteq e \wedge \#const_i(?s) = 1 \wedge \#const_j(?s) = 1 \rightarrow \#?s = 1$

In other words, the number of events in e associated with a unit subset of $const_i(e)$ and a unit subset of $const_i(e)$ is at most one.

This construct suggests that we might profit by generalizing the event function so that it takes any number of arguments. In this example, if we had a two-place event function give-of-1-3 mapping $const_1(give) \times const_3(give)$ onto give, we could replace our no-duplicate assertion with "$\#give\text{-}of\text{-}1\text{-}3(cv(const_1(give)), cv(const_3(give))) = 1$".

This last reading allows charities to be ''shared'' by associates. If there are no shared charities, [12d] degenerates to [12c].

If we commit to [12a] by asserting that there is one donation per associate, i.e. ''#give-of-1(cv(const$_1$(give-12))) = 1'', we can then deduce that a total of \$125 was donated. If we commit to [12b] by asserting that there is one donation per charity and several charities in all, i.e. ''#give-of-3(cv(const$_3$(give-12))) = 1'' and ''#const$_3$(give-12) = S'', we can then deduce that a total of S× \$25 was donated. And if we commit to [12d] by asserting that there are several charities per associate, i.e. ''#const$_3$(give-of-1(cv(const$_1$(give-12)))) = S'', we can then deduce that a total of S× \$125 was donated.[13]

5.6. Redundant Examples

The following sentences were considered uninteresting because they do not give rise to issues that are not covered in examples [1]–[12].

[13] I saw the guys from "Earth Wind & Fire" on TV today. ([Webber79])

[14] The declarations should be studied with some care. (*The C Programming Language*)

[15] The authors maintain a fine balance of building interesting programs and teaching rock-solid engineering principles. (*CACM*)

[16] Several linguists attended the masquerade. They dressed up as cyclic transformations. ([Webber79])

[17] The bacteria was found in abundance in refrigerators, ants, and even in the bodies of employees at the Jalisco plant, he said. (*New York Times*)

[18] The sides of rectangle 1 cross the sides of rectangle 2. ([Scha81])

[19] Twenty-two fires burned out of control in nine Western states yesterday. (*New York Times*)

[20] Citing the growing need for legal services for the poor, a New York panel has urged that the state's 88,000 practicing lawyers be required to devote at least 20 hours a year to public service. (*New York Times*)

---

[13] We can commit to [12c] by adding the assertion that there is one donation per charity, i.e. ''#give-of-3(cv(const$_3$(give-12))) = 1'', but this does not change the total amount donated.

[21] I spoke yesterday with several students who have been flagrantly violating the dormitory regulations. ([Roberts87])

[22] It is undeniable, of course, that women, not men, take pregnancy leaves. (*New York Times*)

Chapter 6: Adding Plurals to an Existing System

6.1. Introduction

At this point, one might think that it would be easy to enable a system to manipulate plurals (sets) simply by replacing its representational theory with CI2. If the old representational theory was a logical one, the conversion might seem to be a trivial matter since CI2 is syntactically identical to the first-order predicate calculus and its semantics are Tarskian. As we are about to see, the conversion is anything but trivial.

Consider the following story: "Chris went to the supermarket. She found the milk and left in a hurry." This story might be represented in a traditional system, e.g. FRAIL ([Charniak88]), as follows:

```
inst(shop-6, shopping-event)      ; shop-6 is a shopping event.
inst(milk-6, milk)                ; milk-6 is milk.
inst(Chris, person)               ; Chris is a person.

agent(shop-6) = Chris             ; Chris is the shopper in shop-6.
purchase-of(shop-6) = milk-6      ; milk-6 is the item purchased in shop-6.
```

Now consider the following two queries:

[*i*] Did Chris buy eggs?

[*ii*] Did Chris pay for her milk?

Suppose this is the only shopping event representation in our system. Our system's inference component responds negatively to query [*i*] because it fails to find a representation for a shopping event in which Chris bought eggs. This failure is partly due to the system's singular representational theory which effectively prohibits people from buying more than one item per shopping event. The following axiom allows us to correctly answer query [*ii*]:

```
inst(?shop, shopping-event) →                          ; The shopping event's
   inst(pay–of(?shop), pay-event)                      ; shopper pays for
   agent(pay-of(?shop)) = agent(?shop) ∧               ; his purchase.
   purchase-of(pay-of(?shop)) = purchase-of(?shop)
```

Now let us see what happens when we translate these traditional representations into CI2. The axiom we need becomes:

```
?shop ⊆ shopping-event →                               ; The shopping event's
   pay-of(?shop) ⊆ pay-event ∧                         ; payers are a subset
   const1(pay-of(?shop)) ⊆ const1(?shop) ∧             ; of its shoppers.
   const2(pay-of(?shop)) = const2(?shop)
```

Our story becomes:

shop-6 ⊆ shopping-event	; shop-6 is a shopping event.
milk-6 ⊆ milk	; milk-6 is milk.
Chris ⊆ person	; Chris is a person.
Chris ⊆ $const_1$(shop-6)	; Chris is a shopper in shop-6.
milk-6 ⊆ $const_2$(shop-6)	; milk-6 is purchased in shop-6.

Allowing for the possibility of plurals has resulted in a weaker set of assertions. Since the first sentence of our story (''Chris went to the supermarket'') could be followed by ''She picked up her brother on the way,'' we do not know that Chris is the only shopper. So, instead of asserting ''Chris = $const_1$(shop-6)'', we assert the weaker ''Chris ⊆ $const_1$(shop-6)''. A similar line of reasoning applies to milk-6. Other items could be purchased along with the milk, so it cannot be equated with $const_2$(shop-6).

At this point, suppose we submit query [*i*]:

?s ⊆ shop ∧	; Is there a shopping event set
$const_1$(?s) = chris-6 ∧	; in which Chris was the shopper
?e ⊆ egg ∧	; and eggs were bought?
?e ⊆ $const_2$(?s) ∧	
#?e > 0	

Our system would do one of two things. It would either respond equivocally because it does not know that Chris did not buy eggs in shop-6, or it would respond negatively because it failed to find a shopping event set in which Chris unequivocally bought eggs. The latter approach (negation-as-failure) is used by the theorem prover described in chapter 4 and is clearly preferable for this query, but it leads to a negative response to query [*ii*] which we represent as follows:

?p ⊆ pay ∧	; Is there a paying event set
$const_1$(?s) = chris-6 ∧	; in which Chris paid for
milk-6 ⊆ $const_2$(?p)	; the milk?

Since we have introduced the possibility of there being additional shoppers, we can no longer deduce Chris' paying for her milk.

6.2. The Closure Problem

Our system's failure to deduce Chris' paying for her milk is due to the *closure problem* (our terminology). In the general case, solving the closure problem entails identifying an upper bound for some of the sets in the representation. In our shopping example, if we continue to use the negation-as-failure approach, we cannot deduce Chris' paying for her milk. We can only presume that she did, e.g. by assuming ''Chris = $const_1$(shop-6)'' or by making the more direct assumption ''Chris = $const_1$(pay-of(shop-6))''. We call these assumptions *closure assumptions*. The first one places an upper bound on shop-6 whereas the second one bounds pay-of(shop-6).

The newness of the closure problem appears to be an artifact of the traditional representational theory's bias against plurals. When one's representational theory can only represent singulars, one tends to assume that all entities are singular, i.e. assume that all sets are of unit cardinality. Of course once one makes this assumption, all nontrivial lower bounds become upper bounds as well. In other words, by ignoring plurals, we conveniently overlooked a very general and serious problem. Whenever we learn that a set A is associated with a second set B, we might have to allow for the possibility that the relevant event or predication actually relates a proper superset of A with (a proper superset of) B.

Other stories that exhibit the closure problem include:

[*a*] Chris wore blue jeans to work.

[*b*] Robin loosened the bolt with an Allen wrench. Leslie held the nut with a crescent wrench to keep it from turning.

We certainly do not want to assume that blue jeans were the only thing Chris wore to work in story [*a*]. And after the first sentence in story [*b*], all we know is that Robin is a member of the bolt-loosening event's agent set. If we assume that this is a unit set, we will have to retract our assumption when we process the second sentence. After story [*b*]'s first sentence we also know that the Allen wrench is a subset of the event's instrument set. Again, if we assume that there is only one instrument, we will have to retract this assumption when we process the second sentence.

Intuitively, we want to do something reminiscent of circumscription, i.e. assume that all sets are as small as possible. Since we have no reason to believe that there are other shoppers in our first example, we would like to assume that there are no other shoppers. Our other two stories demonstrate that we should be careful about making closure assumptions prematurely. Perhaps they are best made only if needed.

The obvious solution to the closure problem is to use some form of non-monotonic or probabilistic reasoning. Non-monotonic reasoning would enable us to retract closure assumptions when we discover conflicting information whereas probabilistic reasoning would enable us to adjust our probabilities accordingly. Both would allow us to safely make the closure assumptions we need to infer things like Chris' paying for her milk and cope with conflicting information that is received later.

If the input is natural language discourse, one might profit from implementing a module that uses Grice's maxims to propose closure assumptions ([Grice75]). But it is not obvious how one would implement such a module and it would probably not be as efficient as the approach we are about to describe.

6.3. A Solution

Stories [*a*] and [*b*] suggest that the best approach is to only make closure assumptions when they are actually needed to make an inference. We have modified the theorem prover of chapter 4 so that it can successfully ''deduce'' Chris' buying her milk by making a closure assumption.

Our program's closure assumption assertion mechanism is triggered by failing equality conjunct retrievals. Suppose our program has failed to retrieve equality conjunct C. Instead of pruning the tree at this point, we put the conjunct back into the conjunct list for later processing. In short, we defer processing C with the hope that retrieving other conjuncts will allow us to bind some or all of the variables in C. Ideally, all of the variables in C will be bound before we process it again, i.e. the closure assumption we want will be fully specified. Deferral also has the advantage of insuring that the assumption we make will actually lead to a successful query retrieval.

This modification did not require any major changes to our program's architecture. The resulting program answers the following three encodings of query [*ii*]:

```
?p ⊆ pay ∧                          ; Is there a paying event in
chris-6 = const1(?p) ∧              ;  which Chris was the payer
milk-6 ⊆ const2(?p)                 ;  and the milk was paid for?

?s ⊆ shop ∧                         ; Is there a shopping event in
chris-6 = const1(pay-of(?s)) ∧      ;  which Chris was the shopper
milk-6 ⊆ const2(?s)                 ;  and the milk was purchased?

?s ⊆ shop ∧                         ; Is there a shopping event in
chris-6 = const1(pay-of(?s)) ∧      ;  which Chris was the shopper
milk-6 ⊆ const2(pay-of(?s))         ;  and the milk was paid for?
```

In each case, our program successfully inferred Chris' paying for milk by asserting the closure assumption "Chris = const$_1$(pay-of(shop-6))".

Our program still answers query [*i*] and the queries in chapter 5 correctly, and it requires roughly the same amount of time. But since our program does not need to make any closure assumptions in order to correctly answer these other queries (and since it only makes a closure assumption if it needs to), we should defer making any hard conclusions about this approach until we have studied more examples.

6.4. Conclusion

As we have seen, adding plurals to an existing (singular) system is not a trivial matter even if the system's representational theory is first-order. Replacing the system's representational theory with CI2 might be trivial, but one will also have to design a module that proposes closure assumptions, and may have to implement non-monotonic or probabilistic reasoning as well.

We have looked at a promising approach in which closure assumptions are only proposed and asserted when needed, but further work needs to be done on the long-overlooked closure problem before our approach can be judged.

Chapter 7: Conclusion

7.1. Looking Forward

With a theory of representation in hand, what comes next? We might want to direct our attention to those problems that we have deliberately ignored, i.e. representing intensions, generics, negation, and time. Until now, we have assumed that the plural representation problem is independent of these. It would be nice if we could verify this assumption by demonstrating how solutions to these other problems are compatible with CI2.

We might try to build a practical inference component. This might entail restricting our representations to some computationally tractable subset of CI2. Perhaps we should translate CI2 representations into a graphical language with a procedural semantics. We might also want to take into account uncertainty. (Information about probabilities can sometimes be translated into statements about set cardinalities.) We will probably want to add primitives, e.g. an intersection function, and give our system the ability to do simple arithmetic.

In the preceding chapter, we proposed an attractive solution to the closure problem in which closure assumptions are proposed and asserted by the inference component when they are needed to make inferences. This approach needs further scrutiny.

Another solution to the closure problem involves implementing a module that uses Grice's maxims to propose closure assumptions when the input is initially processed. We might want to pursue this approach if we tried to build a system that generates our initial, ambiguous representations. Perhaps we could modify the program described in [Harper89] so that it can handle plural sentences.

7.2. Looking Back

We have identified two design goals for a theory of plural representation: facilitating the disambiguation of ambiguous sentences and the making of inferences from them. CI2 is a first-order theory that achieves these goals. It is the first-order predicate calculus embellished with a handful of proper axioms and a "new" but simple ontology. We have identified special classes of functions useful for creating these representations and making inferences from them. We have shown how our theory enables one to incrementally disambiguate sentences and demonstrated its viability with a series of twelve examples using a CI2-specific inference component. And we have identified the closure problem which must be solved if one wishes to add plural representations to an existing system.

Our theorem prover illustrates several techniques that may be of use to those interested in developing systems that can reason about sets. One technique in particular, translating implications into cardinality variable expressions, was shown to be especially useful for representing distributive readings.

It is interesting to speculate why a problem as basic as plural representation received so little attention from the computer science community. Presumably because it appeared to require a higher-order solution. It is somewhat embarrassing that our problem should have not only a simple solution, but one whose roots go back half a century.[1]

[1] If one thinks of individuals as being sets, then CI2 is just CI with numbers added.

References

Brachman, R. et al. (1979) ''An Introduction to KL-ONE'', in *Research in Natural Language Understanding, Annual Report*, Bolt Beranek and Newman Report No. 4274, Cambridge, Mass.

Burge, T. (1972) Truth and Mass Terms, *The Journal of Philosophy*, **69**:263–282.

Charniak E. and Goldman, R. (1988) A Logic for Semantic Interpretation, *Proc. of the 26th Annual Meeting of the ACL*, 87–94.

Davidson, D. (1967) ''The Logical Form of Action Sentences'' in N. Rescher (ed.), *The Logic of Decision and Action*, University of Pittsburgh Press.

Davidson, D. (1970) Events as Particulars, *Noûs*, **4**:25–32.

Frisch, A. (1987) *Knowledge Retrieval as Specialized Inference*, University of Rochester, TR 214.

Frisch, A. and Allen, J. (1982) *Knowledge Representation and Retrieval for Natural Language Processing*, University of Rochester, TR 104.

Grätzer, G. (1978) *General Lattice Theory*, Birkhäuser, Stuttgart.

Grice, P. (1975) ''Logic and Conversation'' in P. Cole and J. Morgan (eds.), *Syntax and Semantics 3: Speech Acts*, Academic Press, New York.

Hobbs, J. (1983) An Improper Treatment of Quantification in Ordinary English, *Proc. of the 21st Annual Meeting of the ACL*, 57–63.

Kempson, R. and Cormack, A. (1981) Ambiguity and Quantification, *Linguistics and Philosophy*, **4**:259–309.

Kuper, G. (1987) Logic Programming With Sets, *Proc. of the 6th ACM Conf. on PODS*, 11–20.

Leonard, H. and Goodman, N. (1940) The Calculus of Individuals and Its Uses, *Journal of Symbolic Logic*, **5**:45–56.

Leśniewski, S. (1927–1931) *O Podstawach Matematyki* (Polish), ***Przeglad Filozoficzny*, 30–34**.

Luschei, E. (1962) *The Logical Systems of Leśniewski*, North-Holland, Amsterdam.

Massey, R. (1976) Tom, Dick, and Harry, and All the King's Men, *American Philosophical Quarterly*, **13**(2):89–107.

Parsons, T. (1970) An Analysis of Mass Terms and Amount Terms, *Foundations of Language*, **6**:362–388.

Roberts, C. (1987) Modal Subordination, Anaphora, and Distributivity, Ph.D. thesis, University of Massachusetts.

Scha, R. (1981) "Distributive, Collective, and Cumulative Quantification", in J. Groenendijk, T. Janssen, and M. Stokhof (eds.), *Formal Methods in the Study of Language*, Mathematical Centre Tracts, nos. 135–136, Amsterdam.

Scha, R. and Stallard, D. (1988) Multi-level Plurals and Distributivity, *Proc. of the 26th Annual Meeting of the ACL*, 17–24.

Schein, B. (1986) Event Logic and the Interpretation of Plurals, Ph.D. thesis, Massachusetts Institute of Technology.

Webber, B. (1979) *A Formal Approach to Discourse Anaphora*, Garland Publishing, New York.

Appendix A: Adding Relations to CI2

Our use of events to represent states and relations is motivated by philosophical and computational concerns. It is not due to restrictions arising from the semantics of CI2. In this Appendix, we will show how to represent states and relations with predications.

If we assert the predication *P(a,b)*, we can then speak of *a* and *b* as being *arguments* of a *P*-relation.

Argument functions map arguments of a relation onto each other in the same way that compositions of event functions and a constituent functions map constituent sets onto each other. In other words, an argument function maps subsets of one argument to the corresponding subsets of another argument. For example, if Love is a relation consisting of tuples of the form (*lover*, *loved-one*), one might want a function mapping its first argument onto its second. If LovedBy is this function and 'Chris' denotes {Chris}, then 'LovedBy(Chris)' denotes the set of people or things loved by {Chris}.

There is a catch. The empty set may not be a predicate argument just as it may not be an event constituent. So, if Chris does not love anyone or anything, we would represent this lack of feeling with $(x)(\neg \text{Love}(\text{Chris}, x))$, not with $\text{Love}(\text{Chris}, \varnothing)$. (Though, in this case 'LovedBy(Chris)' would denote the empty set.)

In general, if g is an argument function mapping R's i-th argument onto its j-th, then $g(a)$ is the union of the j-th entries of those R-tuples having a subset of a as their i-th entry.[1]

So much for semantics. From a proof-theoretic standpoint, if g is an argument function mapping R's first argument onto its second, its behavior would be defined by the following axioms:

C1. $g(y) = \varnothing \leftrightarrow y = \varnothing \vee (z)(x \subseteq y \rightarrow \neg R(x,z))$

C2. $x \subseteq y \wedge R(x,z) \rightarrow z \subseteq g(y)$

C3. $w \subseteq g(y) \wedge \#w = 1 \rightarrow (\text{E}z)(x \subseteq y \wedge R(x,z) \wedge w \subseteq z)$

C4. $x \subseteq y \rightarrow g(x) \subseteq g(y)$

Axiom C1 states that g maps the empty set to itself. g maps a nonempty argument to the empty set when no tuple in R has a subset of it as its first entry. Axiom C2 gives a lower bound for $g(y)$. (If the first entry of a tuple in R is a subset of y, then its second entry is a subset of $g(y)$.) An upper bound is set by axiom C3. (Each element of $g(y)$ is a member of the second entry of some tuple in R whose first entry is a subset of y.) Axiom C4 states that g is monotonic. (See chapter 4 for a discussion of the importance of monotonicity.)

These axioms are quite a bit more complicated than the corresponding event function axioms. This is a direct result of event sets being domain elements. Since predicates are not terms in CI2, we cannot use a function analogous to a constituent function to name a predicate's i-th argument.

---

[1] One might define $g(a)$ to be the union of those R-tuples having a as their i-th entry, but the definition given is more useful in practice.

We assert these axioms via function declarations. For example, the function LovedBy would be declared as follows:

arg-fn(LovedBy, Love, 1, 2)

In other words, LovedBy is an argument function from Love's first argument to its second.[2]

[2] As was the case with event functions, we need not worry about argument function declarations being second-order because they are extra-logical.

Appendix B: Domain-Independent Rules

In this appendix, we list the rules used by our theorem prover to make the inferences in chapter 5.[1] In the preceding pages, the symbol '$\rightarrow$' was used to express logical implications. In this appendix, we will redefine this symbol so we can differentiate between forward-chaining and backward-chaining rules. "$A \leftarrow B$" denotes the backward-chaining rule "to prove A, prove B" whereas "$A \rightarrow B$" denotes the forward-chaining rule "assert B when asserting A." In addition, e_i will denote the i-th constituent of e. We have simplified some of the rules to make them easier to understand; the version of the rule that was actually used differs only in that it is more general. Simplified rules are marked with a dagger (†).

```
#?s > ?n  ←  #?s = ?m ∧ ?m > ?n          ; <rule-1>[2]

#const?i(?e) > 0  ←  #?e > 0             ; <rule-2>
                                         ; ei is nonempty, if e is nonempty.

#?f(?s) > 0  ←                           ; <rule-3>
   ev-fn(?f, ?e, ?i) ∧                   ; f(s) is nonempty, if
   ?s ⊆ const?i(?e) ∧                    ;    f maps ei onto e, and
   #?s > 0                               ;    s is a nonempty subset of ei, and
   #const?i(cv(?e)) = 1                  ;    each unit subset of e is
                                         ;    associated with a unit subset of ei.

#?f(?s) = 0  ←                           ; <rule-4>
   ev-fn(?f, ?e, ?i) ∧                   ; f(s) is empty, if
   #?e = 1 ∧                             ;    f maps ei onto e, and
   ?s ⊆ const?i(?e) ∧                    ;    e is a unit set, and
   #const?i(?e) > #?s                    ;    s is a proper subset of ei.

#const?i(?a) =                           ; <rule-5> †
   #const?j(?a) ×                        ; The size of ei is
   #const?i(?f(cv(const?j(?a)))) ←       ;    the size of ej multiplied by
                                         ;    the number of i-th constituents
      ev-fn(?f, ?e, ?j) ∧                ;    associated with each unit subset of ej, if
      disjoint(?e, ?i) ∧                 ;    the i-th constituents are disjoint, and
      ?a ⊆ ?e ∧                          ;    all j-th constituents are unit sets.
      #const?j(cv(?a)) = 1
```

[1] Procedurally encoded axioms, e.g. the ones introduced in chapter 3, are not listed here.

[2] <rule-1> "fixes" a bug in the program's equivalence class manager. (The program should not be using "#?s" as a class name if #?s = ?n. This is explained in section 4.2.2.)

```
#const?i(?a) =                                  ; <rule-6> †
   #const?j(?a) ×                               ; The size of ei is
   #const?i(cv(?a))  ←                          ;    the size of ej multiplied by
                                                ;    the number of i-th constituents
      ev-fn(?f, ?e, ?j) ∧                       ;    associated with each unit subset of e, if
      disjoint(?e, ?i) ∧                        ;    the i-th constituents are disjoint, and
      ?a ⊆ ?e ∧                                 ;    there is a one-to-one correspondence
      #const?j(cv(?a)) = 1                      ;    between ej and e.
      #?f(cv(const?j(?a))) = 1

#const?i(?a) =                                  ; <rule-7> †
   #const?i(cv(?a)) ×                           ; The size of ei is
                                                ;    the number of i-th constituents associated
   #const?j(?a) ×                               ;    with each unit subset of e multiplied by
   #const?k(?f(cv(const?j(?a))))  ←             ;    the size of ej multiplied by
                                                ;    the number of k-th constituents
      ev-fn(?f, ?e, ?j) ∧                       ;    associated with each unit subset of ej, if
      disjoint(?e, ?i) ∧                        ;    the i-th constituents are disjoint, and
      no-duplicates(?e, ?j, ?k) ∧               ;    a unit subset of ej and a unit
      ?a ⊆ ?e ∧                                 ;    subset of ek defines a unit
      #const?j(cv(?a)) = 1 ∧                    ;    subset of e.
      #const?k(cv(?a)) = 1

disjoint(?s, ?i)  ←                             ; <rule-8>
   disjoint(?t, ?i) ∧ ?s ⊆ ?t                   ; s's i-th constituents are disjoint, if
                                                ;    t's i-th constituents are disjoint, and
                                                ;    s is a subset of t.

[#?f(?e) = ?n  →  #?e > 0]  ←  ?n > 0           ; <rule-9>
                                                ; e is nonempty if f(e) is nonempty.

[const?i(?e) = ?s  →  ?f(?s) = ?e]  ←           ; <rule-10>
   ev-fn(?f, ?x, ?i) ∧ ?e ⊆ ?x                  ; f maps s onto e, if
                                                ;    s is ei.

ev-fn(?f, ?e, ?i)  →                            ; <fc-rule-1>
   [const?i(?f(?s)) = ?s  ←                     ; If f maps ei onto e, then
      ?s ⊆ ?t ∧                                 ;    the i-th constituent of f(s) is s, if
      ?t ⊆ const?i(?e) ∧                        ;    each element of s is
      #?f(cv(?t)) > 0]                          ;    associated with a
                                                ;    nonempty subset of e.
```

Appendix C: Sample Theorem Prover Trace

In this appendix, we present an annotated trace of our theorem prover making an inference. We will infer that each of the Dutch firms in reading [A] of example 7 owns at least one American computer. (This example is presented on pp. 46-48.) The inference is made by proving the theorem "#const$_2$(own-of-1(cv(const$_1$(own-7)))) > 0". Our Symbolics 3645 needed about nine seconds to make this inference.

The following trace contains statements generated by two of our program's key routines: *foo-retrieve-conj* and *foo-default-ret*.[1] The input to foo-retrieve-conj is a *branch*. A branch consists of a list of conjuncts and a substitution. Its printed representation is:

(branch *conjunct-list substitution*)

The printed representation of a substitution is:

(subst *query-variable-binding-list data-base-variable-binding-list*)

foo-retrieve-conj tries to prove the first conjunct in the conjunct list either by unifying it with an axiom in the data base, or by invoking a backward-chaining rule. It returns a branch for each unifying axiom and for each applicable rule. If the conjunct list in the returned branch is empty, the branch's substitution corresponds to a proof of the theorem.

foo-default-ret is the default conjunct retrieval routine. It is called by foo-retrieve-conj either directly or via the program's specialized retrieval routines. This is the routine where conjuncts are unified with stored assertions and backward-chaining rules are applied.

By tracing foo-retrieve-conj and foo-default-ret, we can see which axioms and rules are used. The following trace was edited somewhat for readability. Our annotations are given on the right.

```
retrieve-conj: input =                                    ; the original query
  (branch (#const2(owner-ev(cv(const1(own-7)))) > 0)
      (subst nil nil))
 default-ret: firing <rule-2>
 default-ret: firing <rule-1>
retrieve-conj: input =                                    ; <rule-1> applied.
  (branch (#const2(owner-ev(cv(const1(own-7)))) = ?m1,
      ?m1 > 0)
      (subst nil nil))
 default-ret: firing <rule-12>
 default-ret: firing <rule-11>
```

[1] Temporary names have a way of becoming permanent.

<rule-12> is the result of applying <fc-rule-1> for the event function ownee-ev, whereas <rule-11> corresponds to owner-ev.

```
default-ret: firing <rule-7>
default-ret: firing <rule-6>
default-ret: firing <rule-5>
```

None of these rules would lead to a proof of the theorem. The program is smart enough to prune the tree, i.e. delete the branches corresponding to these rule firings, before continuing. But the attempt to prove the theorem via <rule-1> continues because ?m1 unifies with #const$_2$(owner-ev(cv(const$_1$(own-7)))). This leads to:

```
retrieve-conj: input =
   (branch (#const2(owner-ev(cv(const1(own-7)))) > 0)
        (subst nil nil))
 default-ret: firing <rule-2>
retrieve-conj: input =                                  ; <rule-1> and <rule-2>
   (branch (#owner-ev(cv(const1(own-7))) > 0)           ; applied.
        (subst nil nil))
 default-ret: firing <rule-3>
retrieve-conj: input =                                  ; <rule-1>, <rule-2>,
   (branch (1 > 0,                                      ; and <rule-3> applied.
         #const?i6(cv(owner-ev(cv(const1(own-7))))) = 1)
        (subst nil nil))
retrieve-conj: input =                                  ; since 1 > 0 . . .
   (branch (#const?i6(cv(owner-ev(cv(const1(own-7))))) = 1)
        (subst nil nil))
 default-ret: matched ''#const1(cv(ownee-ev(ac))) = 1''
```

Here we see a complication introduced by our CV expressions. If $A \subseteq B$ and $E(x)$ is some expression with free variable x, then $E(\text{cv}(A))$ is true whenever $E(\text{cv}(B))$ is true. So, if $E(\text{cv}(B))$ is an axiom and we attempt to ''unify'' it with $E(\text{cv}(A))$, what we want to do is add ''$A \subseteq B$'' to the conjunct list and continue. In this case, the added conjunct is ''owner-ev(cv(const$_1$(own-7))) $\subseteq$ ownee-ev(ac)''.

```
retrieve-conj: input =                                  ; [2]
   (branch (1 = 1,
         owner-ev(cv(const1(own-7))) ⊆ ownee-ev(ac))
        (subst nil nil))
retrieve-conj: input =
   (branch (owner-ev(cv(const1(own-7))) ⊆ ownee-ev(ac))
        (subst nil nil))
```

[2] Recall that cv is shorthand for cv$_1$.

This retrieval succeeds, but only after making seven calls to foo-retrieve-conj:

```
retrieve-conj: input =
   (branch (ev-fn(owner-ev, ?x7))
        (subst nil nil))
  default-ret: matched ''ev-fn(owner-ev, (own, 1))''
retrieve-conj: input =
   (branch ((ev-fn(ownee–ev, ?x8))
        (subst nil nil))
  default-ret: matched ''ev-fn(ownee-ev, (own, 2))''
retrieve-conj: input =
   (branch (own = own-7)
        (subst nil nil))
retrieve-conj: input =
   (branch (const2(own) = const2(own-7))
        (subst nil nil))
retrieve-conj: input =
   (branch (const2(own) = ac)
        (subst nil nil))
retrieve-conj: input =
   (branch (own = ownee-ev(ac))
        (subst nil nil))
retrieve-conj: input =
   (branch (ev-fn(ownee-ev, ?x9))
        (subst nil nil))
  default-ret: matched ''ev-fn(ownee-ev, (own, 2))''
```

Since the desired subset relationship holds, the proof beginning with the application of <rule-1> has succeeded. We record the subset relationship in a hash table and continue searching for a proof beginning with the application of <rule-2>.[3] The preceding proof never really made use of <rule-1> so we expect to find a second proof here.

```
retrieve-conj: input =                                  ; <rule-2> applied.
   (branch (#owner-ev(cv(const1(own-7))) > 0)
        (subst nil nil))
  default-ret: firing <rule-3>
  default-ret: firing <rule-1>
retrieve-conj: input =                                  ; <rule-2> and <rule-1> applied.
   (branch (#owner-ev(cv(const1(own-7))) = ?m3,
        ?m3 > 0)
        (subst nil nil))
```

[3] At this point, we could stop, but it is easier to write a program that simply tries everything.

default-ret: firing <rule-12>
default-ret: firing <rule-11>
default-ret: firing <rule-4>

Once again, the program deletes the branches corresponding to <rule-11> and <rule-12>, unifies ?m3 with #owner-ev(cv(const$_1$ (own-7))), and continues.

retrieve-conj: input =
(branch (#owner-ev(cv(const$_1$ (own-7))) > 0)
(subst nil nil))

default-ret: firing <rule-3>

retrieve-conj: input = ; <rule-2>, <rule-1>,
(branch (1 > 0, ; and <rule-3> applied.
#const$_{?i14}$ (cv(owner-ev(cv(const$_1$ (own-7))))) = 1)
(subst nil nil))

retrieve-conj: input =
(branch #const$_{?i14}$ (cv(owner-ev(cv(const$_1$ (own-7))))) = 1)
(subst nil nil))

default-ret: matched ''#const$_1$ (cv(ownee-ev(ac))) = 1''

retrieve-conj: input =
(branch (1 = 1,
cv(owner-ev(cv(const$_1$ (own-7)))) ⊆ ownee-ev(ac))
(subst nil nil))

retrieve-conj: input =
(branch (cv(owner-ev(cv(const$_1$ (own-7)))) ⊆ ownee-ev(ac))
(subst nil nil))

The previous retrieval of owner-ev(cv(const$_1$ (own-7))) ⊆ ownee-ev(ac) modified the hash table so that we only need one call to foo-retrieve-conj to make this subset retrieval. (We can ignore any CV operators to the left of the subset operator.)

retrieve-conj: input =
(branch (ev-fn(owner-ev, ?x10))
(subst nil nil))

default-ret: matched ''ev-fn(owner-ev, (own, 1))''

We now have a proof of the theorem that begins with the application of <rule-2>. We still have two more paths to check. The first one is due to <rule-4>:

```
retrieve-conj: input =                                  ; <rule-2>, <rule-1>,
  (branch (ev-fn(owner-ev, (?e5, ?i13)),                ; and <rule-4> applied.
       0 > 0,
       cv(const1 (own-7)) ⊆ const?i13 (?e5),
       #?e5 = 1,
       #const?i13 (?e5) > 1)
      (subst nil nil))
 default-ret: matched (fn owner-ev (own 1))
retrieve-conj: input =                                  ; no luck.
  (branch (#own = 1,
       0 > 0,
       cv(const1 (own-7)) ⊆ const1 (own),
       #const1 (own) > 1)
      (subst nil nil))
```

Note that the order of the conjuncts may change as query variables are bound. In this case, the conjunct ''#?e5 = 1'' became ''#own = 1'' as ?e5 became bound. It might appear that we should keep ''0 > 0'' as the conjunct list's first entry, but the program's equivalence class hash tables make ''#own = 1'' about as easy to prove or disprove as ''0 > 0''.

So the <rule-4> path fails to lead to a proof. We have one more path, <rule-2> followed by <rule-3>, to investigate. This path should lead to a proof because it is essentially identical to the previous successful paths.

```
retrieve-conj: input =                                  ; <rule-2> and <rule-3>
  (branch (1 > 0,                                       ; applied.
       #const?i12 (cv(owner-ev(cv(const1 (own-7))))) = 1)
      (subst nil nil))
retrieve-conj: input =
  (branch (#const?i12 (cv(owner-ev(cv(const1 (own-7))))) = 1)
      (subst nil nil))
 default-ret: matched ''#const1 (cv(ownee-ev(ac))) = 1''
retrieve-conj: input =
  (branch (1 = 1,
       cv(owner-ev(cv(const1 (own-7)))) ⊆ ownee-ev(ac))
      (subst nil nil))
retrieve-conj: input =
  (branch (cv(owner-ev(cv(const1 (own-7)))) ⊆ ownee-ev(ac))
      (subst nil nil))
 retrieve-conj: input =
(branch (ev-fn(owner-ev, ?x11))
     (subst nil nil))
  default-ret: matched ''ev-fn(owner-ev, (own 1))''     ; Q.E.D.
```