Research Report AI–1996–01

Natural Language Plurals in Logic Programming Queries

Michael A. Covington

Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602–7415 U.S.A

Natural Language Plurals in Logic Programming Queries

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, GA 30602 U.S.A.
mcovingt@ai.uga.edu

February 1996

Abstract

This paper presents a representation for natural language plurals in knowledge base queries, implementing collective, distributive, cumulative, and multiply distributive senses of the plural by means of higher predicates.

The underlying semantics is based on Franconi's theory of collections. The COLLECTIVE reading of a plural applies the predicate to the whole collection; the DISTRIBUTIVE reading applies the predicate to all of the elements; and the CUMULATIVE reading says that each element either satisfies the predicate, or belongs to some collection that does so. Implicit in the cumulative reading is the notion of ELEMENT–PROPERTY, the property of belonging to some collection that satisfies a given predicate.

Another reading of the plural, here termed the MULTIPLY DISTRIBUTIVE, requires a straightforward extension of the system to allow simultaneous distribution over more than one variable at once, with none of the distributions having scope over any other. Simultaneous distribution is implemented as a metalogical predicate that transforms queries before executing them.

1 Introduction

Notoriously, natural language plurals have multiple readings, or at least multiple sets of truth-conditions. To take a familiar example, *Three men lifted the piano* normally refers to one collective act, while *Three men played the piano* refers to three individual acts. We distinguish these by using real-world knowledge; as far as the language is concerned, either sentence could be interpreted either way.

The translation of sentences containing plurals (such as *Did the men play the piano?*) into database queries is therefore non-trivial. In this paper I present computationally tractable formal representations for natural language plurals in logic programming queries, together with Prolog code for implementing them. Along with the widely recognized collective, distributive, and 'mixed' or cumulative readings of the plural, I handle an additional, previously unrecognized, sense of the plural which I term MULTIPLY DISTRIBUTIVE.[1]

I assume that the database itself is free of ambiguity and contains only facts about individuals and/or collections. The problem that I am addressing is therefore quite distinct from that of how to encode plural facts in a knowledge base, addressed, for example, by Mellish [9], Sowa [11], and Link and Schütze [8]. When encoding knowledge from a natural language source, one has to decide whether each plural is to be taken as cumulative, distributive, or something else. When answering a query, the task is much simpler: find out whether the query is true on any of its possible readings.[2]

2 Plurals as collections

Franconi [3] argues convincingly that plurals in natural language refer, not to sets, but to COLLECTIONS, which are like sets except that two collections can have exactly the same elements and still be distinct. For example, *the Beatles* and *the owners of Apple Records* refer to two distinct collections that at one time consisted of the same four people but had different collective properties

[1]I am grateful to Donald Nute and three anonymous referees for helpful comments and suggestions. All errors here are of course my own.

[2]This, in turn, is why I did not use a mereological or lattice-based account of the plural (as [8], [12]): databases are not mereological.

(in fact, one of them could owe money to the other). Formally, the premises

$beatles \subseteq appleowners$
$appleowners \subseteq beatles$

do not imply

$appleowners = beatles$.

Natural language plurals have at least three readings:

- The COLLECTIVE reading, which attributes a property to the collection as a whole: *John is the leader of the (group called the) Beatles.*
- The DISTRIBUTIVE reading, which attributes a property to each of the elements of the collection: *The Beatles were (each) born in Liverpool* (but their births were four separate events).
- The CUMULATIVE or 'mixed' reading, which is satisfied when each member of the collection either satisfies the predicate itself, or belongs to some collection (not necessarily the named one) that satisfies the predicate. For example, if Paul and George sang 'Yesterday' in the village choir, and Ringo and John separately sang soloes of it, then *The Beatles have sung 'Yesterday'* is true in the cumulative sense.[3]

Both the collective and the distributive readings are special cases of the cumulative. Thus, plurals need not be considered ambiguous; instead; one can say that the plural simply has a variety of truth-conditions ([5], [6]; see [7] for extensive review). Nonetheless, I shall continue to treat the collective and distributive as special cases because they are especially easy to compute.

In the collective reading, the predicate applies to the collection itself:

John is the leader of the Beatles.
$leader\text{-}of(beatles, john)$

[3]The term 'cumulative' raises two minor problems. First, hardly anyone besides Franconi recognizes that a cumulative can be satisfied through a collection larger than the one referred to (e.g., the village choir in my Beatles example); cf. [4]. Second, many scholars reserve the term CUMULATIVE for situations in which the predicate describes some kind of sum of the properties of the individuals (e.g., *The boys earned $6,000*) [10] [12]. Franconi's more general cumulative is then called the WEAK COLLECTIVE. In this paper I will stick with Franconi's nomenclature.

For the distributive and cumulative readings, Franconi introduces the "plural quantifiers" $\lhd$, $\rhd$, $\trianglelefteq$, and $\trianglerighteq$, defined as follows:

$$\begin{array}{lll} \lhd R(a,b) & \text{iff} & \forall x.\ x \in a \rightarrow R(x,b) \\ \rhd R(a,b) & \text{iff} & \forall x.\ x \in b \rightarrow R(a,x) \\ \trianglelefteq R(a,b) & \text{iff} & \forall x.\ x \in a \rightarrow (R(x,b) \vee \exists s.\ x \in s \wedge R(s,b))) \\ \trianglerighteq R(a,b) & \text{iff} & \forall x.\ x \in b \rightarrow (R(a,x) \vee \exists s.\ x \in s \wedge R(a,s))) \end{array}$$

Thus, the distributive reading of *The Beatles were born in Liverpool* is represented as

$$\lhd \mathit{born\text{-}in}(\mathit{beatles}, \mathit{liverpool})$$

and the cumulative reading of *The Beatles sang 'Yesterday'* (in an unspecified mix of collective and individual actions) is:

$$\trianglelefteq \mathit{sang}(\mathit{beatles}, \mathit{yesterday})$$

3 Not quantifiers but higher predicates

Franconi's plural quantifiers form part of a conceptual-graph-like language in which there are no predicates with more than two arguments and no use of higher-order logic. Prolog subject to neither of these restrictions, and in Prolog, it is more convenient to implement plurality through higher–order predicates, for two reasons.

First, plural quantifiers are not quantifiers in the classical sense: they do not bind variables. Nor are they generalized quantifiers ([2], [13]), since they make no statements about the relative numbers of individuals that satisfy two predicates.

Second, and more importantly, a separate set of plural quantifiers is needed for each argument position. For example, $\lhd$ distributes the first argument and $\rhd$ distributes the second. To distribute a third argument would require a third, quantifier. The same holds for cumulatives ($\trianglelefteq$, $\trianglerighteq$). *Entia non sunt multiplicanda.*

Viewed as second–order predicates, $\lhd$ and $\trianglelefteq$ can be defined thus:

$$\lhd(c,P) \quad \text{iff} \quad \forall x.\ x \in c \rightarrow P(x)$$

$$\trianglelefteq(c,P) \quad \text{iff} \quad \forall x.\ x \in c \rightarrow (P(x) \vee \exists s.\ x \in s \wedge P(s))$$

In English:

$\lhd(c, P)$ means P is true of every element of c;

$\trianglelefteq(c, P)$ means that for every element x of c, P is true of x or of some collection s containing x.

Here P is a one–place predicate constructed by lambda–abstraction from any suitable expression. Some examples:

The Beatles were born in Liverpool.
(distributive reading)
$\lhd(beatles, \lambda x.born\text{-}in(x, liverpool))$

The Beatles sang 'Yesterday.'
(cumulative reading)
$\trianglelefteq(beatles, \lambda x.sang(x, yesterday))$

Regardless of the complexity of the predication, only $\lhd$ and $\trianglelefteq$ are needed.

4 Generalizing the cumulative reading

The new definition of $\trianglelefteq$ expresses two ideas:

- the cumulative reading involves all the elements of the collection;
- individual elements can satisfy the predicate either directly, or by belonging to a collection that satisfies the predicate.

Let us pick these apart by defining another second–order predicate $\mathcal{E}$ ("element–property") that expresses only the second idea:

$\mathcal{E}(x, P) \quad \text{iff} \quad P(x) \vee \exists s.\ x \in s \wedge P(s)$

There are natural–language singulars that need $\mathcal{E}$. For example, suppose John sang 'Yesterday' in the village choir, but not by himself. Then *John sang 'Yesterday'* is true in the sense of

$\mathcal{E}(john, \lambda x.\ sang(x, yesterday))$

even though $sang(john, yesterday)$ is false. That is, John has the property "sang *Yesterday*" as an element of a collection, but not as an individual. This is what Lasersohn calls a PARTICIPATORY DISTRIBUTIVE ([7], p. 107).

Given $\mathcal{E}$, we can make the definitions of $\lhd$ and $\trianglelefteq$ more parallel:

$\lhd(c, P)$ iff $\forall x.\ x \in c \rightarrow P(x)$

$\trianglelefteq(c, P)$ iff $\forall x.\ x \in c \rightarrow \mathcal{E}(x, P)$

That is, the cumulative reading does to $\mathcal{E}$ what the distributive reading does to the simple predicate.

5 Implementation in Prolog

The system thus far described is easily implemented in Prolog. Four meta–predicates (higher–order predicates) are needed:

```
% all(G1,G2)
%   succeeds if every instantiation that satisfies
%   G1 can be further instantiated to satisfy G2.

all(G1,G2) :- \+ (G1, \+ G2).

% dist(C,L)
%   succeeds if L (a lambda-expression) is true of
%   each of the elements of collection C.

dist(C,lambda(X,P)) :- all(element(X,C),P).

% e(E,L)
%   succeeds if L is true of E or of a collection
%   to which E belongs.

e(E,lambda(E,P)) :- P.

e(E,lambda(C,P)) :- element(E,C), P.

% cumu(C,L)
%   succeeds if L is cumulatively true of C
```

```
%   (in Franconi's sense).

cumu(C,lambda(X,P)) :- all(element(E,C), e(E,lambda(X,P))).
```

The first of these is built in, as `forall`, in many Prolog systems. Note that `all` and `forall` succeed when there are *no* ways to satisfy the first goal, whereas natural–language *all* presupposes *some.* This discrepancy between natural language and logic could be removed by modifying `forall`, but in order to avoid complicating the rest of my exposition I shall ignore it. Note also that `lambda` has no special meaning in Prolog; it is a functor that I chose arbitrarily to tie together a variable and a goal in which that variable is to be instantiated.

Franconi's collective and distributive Beatles examples go into Prolog straightforwardly. First we define the Beatles and give some information about them:

```
element(john,beatles).
element(paul,beatles).
element(george,beatles).
element(ringo,beatles).

led_by(beatles,john).

born_in(john,liverpool).
born_in(paul,liverpool).
born_in(george,liverpool).
born_in(ringo,liverpool).
```

A collective query is trivial:

```
%  Are the Beatles led by John?
?- led_by(beatles,john).
yes
```

A distributive query calls the meta–predicate `dist/2`:

```
%  Were the Beatles born in Liverpool?
?- dist(beatles,lambda(X,born_in(X,liverpool))).
yes
```

Actually, in place of `yes`, Prolog systems typically return an answer like "`X = _0123`", which means, "Yes, and `X` is still uninstantiated," because there is no value of `X` to report. The practical effect is the same.

Finally, here is a cumulative example, from Franconi ([3], p. 457):

```
element(paul,c1).
element(john,c1).
element(elvis,c1). % not a Beatle

element(paul,c2).
element(george,c2).
element(ringo,c2).

sang(c1,yesterday).
sang(c2,yesterday).

%  Did the Beatles sing 'Yesterday'?
% (cumulative reading)
?- cumu(beatles,lambda(X,sang(X,yesterday))).
yes
```

Using `e/2` (i.e., $\mathcal{E}$), the system can also handle singular queries that are satisfied through membership in collections:

```
%  Did Ringo sing 'Yesterday'
%  (in a group somewhere)?
?- e(ringo,lambda(X,sang(X,yesterday))).
yes
```

Here most Prolog systems will report that `X = c2`, identifying the collection involved.

6 Complexity of distributives and cumulatives

The complexity of `dist` is linearly proportional to the size of the collection. To prove `dist(beatles,lambda(X,born_in(X,liverpool))` the inference

engine looks up each Beatle and proves `born_in(X,liverpool)` for that individual. Thus, for an n-element collection, the predicate is queries n times.

The complexity of `cumu` is proportional to the size of the collection in question and to the number of other collections to which each element belongs. Let c be a collection of n elements each of which belongs to a total of m collections. Then in the worst case, the queried predicate is called n times (for the individuals) plus nm times (for each individual as a member of each collection), making a total of $n + mn$ calls.

If n and m are proportional to the size of the database, then relative to the size of the database, the complexity of `dist` is linear and that of `cumu` is quadratic.

7 Multiple plurals in a single sentence

This analysis correctly predicts that if a sentence contains more than one plural, each of the plurals can be collective, distributive, or cumulative.

Leaving the Beatles aside, consider for example the sentence *The farmers fed the donkeys,* where there are three farmers and three donkeys. The sentence is true in at least the following nine situations:

1. The farmers as a group fed the donkeys as a group.

   $fed(farmers, donkeys)$

   ```
   fed(farmers,donkeys)
   ```

2. Each of the three farmers fed the whole group of donkeys.

   $\lhd(farmers, \lambda x.fed(x, donkeys))$

   ```
   dist(farmers,lambda(X,fed(X,donkeys)))
   ```

3. Two farmers together and one farmer separately fed the whole group of donkeys.

   $\trianglelefteq(farmers, \lambda x.fed(x, donkeys))$

   ```
   cumu(farmers,lambda(X,fed(X,donkeys)))
   ```

4. The farmers as a group fed each of the three donkeys individually.

   $\lhd(donkeys, \lambda y.fed(farmers, y))$

   ```
   dist(donkeys,lambda(Y,fed(farmers,Y)))
   ```

5. Each of the three farmers fed each of the three donkeys. (Nine acts of feeding.)

   $\lhd(farmers, \lambda x. \lhd (donkeys, \lambda y.fed(x, y)))$

   ```
   dist(farmers,lambda(X,dist(donkeys,lambda(Y,fed(X,Y)))))
   ```

6. Two farmers together and one farmer separately fed each of the three donkeys. (Six acts of feeding.)

   $\trianglelefteq(farmers, \lambda x. \lhd (donkeys, \lambda y.fed(x, y)))$

   ```
   cumu(farmers,lambda(X,dist(donkeys,lambda(Y,fed(X,Y)))))
   ```

7. The farmers as a group fed two of the donkeys together and one donkey separately.

   $\trianglelefteq(donkeys, \lambda y.fed(farmers, y))$

   ```
   cumu(donkeys,lambda(Y,fed(farmers,Y)))
   ```

8. Each of the three farmers fed two of the donkeys together and one donkey separately. (Six acts of feeding.)

   $\lhd(farmers, \lambda x. \trianglelefteq (donkeys, \lambda y.fed(x, y)))$

   ```
   dist(farmers,lambda(X,cumu(donkeys,lambda(Y,fed(X,Y)))))
   ```

9. Two farmers together fed two donkeys together and one donkey separately; then the third farmer fed two donkeys together and one donkey separately.

   $\trianglelefteq(farmers, \lambda x. \trianglelefteq (donkeys, \lambda y.fed(x, y)))$

   ```
   cumu(farmers,lambda(X,cumu(donkeys,lambda(Y,fed(X,Y)))))
   ```

(For each of the cumulative readings, other combinations of individuals and groups are of course possible.)

Notice that all the other readings are special cases of the last one. Thus, a system could answer the question "Did the farmers feed the donkeys?" by trying only the last query, not the preceding eight.

8 Multiple distribution

The farmers fed the donkeys is also true when farmer 1 fed donkey 1, farmer 2 fed donkey 2, and farmer 3 fed donkey 3, respectively — a situation not encompassed in any of the nine examples above. I call this the RESPECTIVE reading.

The collective, distributive, and cumulative readings relate the *whole collection* of farmers (as a collection, as three individuals, or mixed) to the *whole collection* of donkeys (likewise). The respective reading relates *one* farmer to *one* donkey, the next farmer to another donkey, and so on. Thus, there is no way to construct a respective reading from two instances of $\lhd$ or $\trianglelefteq$. Here "plural quantifiers" are particularly unlike real quantifiers, because neither of them has scope over the other; instead, the respective reading performs a distribution over the two collections simultaneously. In effect, it is DOUBLY DISTRIBUTIVE.

Double distribution also holds in situations where the mapping is not one–to–one. For example, *The farmers fed the donkeys* is still true in the situation where farmer 1 fed donkeys 1 and 2 and farmers 2 and 3 both fed donkey 3. The respective reading is the special case of the doubly distributive reading in which the mapping is one–to–one.

To account for double distribution, we can define the following second–order predicate:

$$\lhd^2(c_1, c_2, P) \quad \text{iff}$$
$$\lhd(c_1, \lambda x.\ \exists y.\ y \in c_2 \wedge P(x, y)) \wedge$$
$$\lhd(c_2, \lambda y.\ \exists x.\ x \in c_1 \wedge P(x, y))$$

This can, of course, also be implemented as a Prolog meta–predicate.

But even double distribution is not enough. The sentence

The mothers gave the books to the children.

has a reading — perhaps even the preferred reading — in which each mother gives one book to one child, and

The dealers gave the keys for the cars to the purchasers.

has a quadruply distributive reading. The number of variables that are distributed does not seem to be subject to a definite limit, nor are such variables confined to semantic arguments of the main verb or any other single

natural–language predicate. Multiple distribution is apparently related to branching quantification [1], but it is not the same phenomenon because crossing Skolem–dependencies are not involved.

9 Implementing multiple distribution

Here are formulas for triple and quadruple distributives, respectively:

$$\begin{aligned}
&\lhd^3(c_1, c_2, c_3, P) \quad \text{iff} \\
&\quad \lhd(c_1, \lambda x.\ \exists y.\ y \in c_2 \wedge \exists z.\ z \in c_3 \wedge P(x, y, z)) \wedge \\
&\quad \lhd(c_2, \lambda y.\ \exists x.\ x \in c_1 \wedge \exists z.\ z \in c_3 \wedge P(x, y, z)) \wedge \\
&\quad \lhd(c_3, \lambda z.\ \exists x.\ x \in c_1 \wedge \exists y.\ y \in c_2 \wedge P(x, y, z))
\end{aligned}$$

$$\begin{aligned}
&\lhd^4(c_1, c_2, c_3, c_4, P) \quad \text{iff} \\
&\quad \lhd(c_1, \lambda x.\ \exists y.\ y \in c_2 \wedge \exists z.\ z \in c_3 \wedge \exists w.\ w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_2, \lambda y.\ \exists x.\ x \in c_1 \wedge \exists z.\ z \in c_3 \wedge \exists w.\ w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_3, \lambda z.\ \exists x.\ x \in c_1 \wedge \exists y.\ y \in c_2 \wedge \exists w.\ w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_4, \lambda w.\ \exists x.\ x \in c_1 \wedge \exists y.\ y \in c_2 \wedge \exists z.\ z \in c_3 \wedge P(x, y, z, w))
\end{aligned}$$

The emerging pattern can be made clearer by moving the quantifiers to the beginning, and then adding a redundant $\in$ clause to each of the conjuncts:

$$\begin{aligned}
&\lhd^4(c_1, c_2, c_3, c_4, P) \quad \text{iff} \\
&\quad \lhd(c_1, \lambda x.\ \exists y \exists z \exists w.\ x \in c_1 \wedge y \in c_2 \wedge z \in c_3 \wedge w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_2, \lambda y.\ \exists x \exists z \exists w.\ x \in c_1 \wedge y \in c_2 \wedge z \in c_3 \wedge w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_3, \lambda z.\ \exists x \exists y \exists w.\ x \in c_1 \wedge y \in c_2 \wedge z \in c_3 \wedge w \in c_4 \wedge P(x, y, z, w)) \wedge \\
&\quad \lhd(c_4, \lambda w.\ \exists x \exists y \exists z.\ x \in c_1 \wedge y \in c_2 \wedge z \in c_3 \wedge w \in c_4 \wedge P(x, y, z, w))
\end{aligned}$$

(Recall that, for example, $\lhd(c_1, \lambda x.\ x \in c_1)$ is a tautology; thus the added $\in$ clauses do not affect the truth value of the whole formula.) Now the n conjuncts of an n-way distributive are alike except that, in each of them, a different argument is chosen for distribution and all the rest of the arguments are existentially quantified.

Because existential quantifiers are implicit in Prolog, these cumbersome logical formulas have a straightforward, general Prolog implementation. Suppose we use $\lhd$ to mark collections over which a predicate is to be distributed, writing $P(\lhd c_1, \lhd c_2, \lhd c_3, \lhd c_4)$ to indicate that the arguments of P are to come from simultaneous distribution over c_1, c_2, c_3 and c_4 respectively. (Arguments that are not to be distributed would not be marked with $\lhd$.) Then $P(\lhd c_1, \lhd c_2, \lhd c_3, \lhd c_4)$ can be converted into the above four–way distributive

formula before the query is executed in Prolog. The following code performs the translation:

```
% multiply_distribute(+Term,-Result)
%  transforms Pred(Arg1,Arg2...ArgN) to a query in which
%  simultaneous multiple distribution has been performed
%  on all arguments of the form dist(Collection), and
%  other arguments are left unchanged.

multiply_distribute(Term,Result) :-
  make_conjunct(Term,Conjunct,Args,Vars),
  join_conjuncts(Args,Vars,Conjunct,R),
  (R == true ->
     Result = Term    % special case, no args distributed
  ;
     Result = R       % normal case
  ).

% join_conjuncts(+Args,+Vars,+Conjunct,-Result)
%  given lists of variables and corresponding
%  arguments, and the schema for one conjunct,
%  constructs whole formula.

join_conjuncts([],_,_,true).

join_conjuncts([dist(Arg)|Args],[Var|Vars],Conjunct,
               (dist(Arg,lambda(Var,Conjunct)),Rest)) :-
  !,
  join_conjuncts(Args,Vars,Conjunct,Rest).

join_conjuncts([Arg|Args],[Var|Vars],Conjunct,Result) :-
  join_conjuncts(Args,Vars,Conjunct,Result).

% make_conjunct(+Term,-Result,-Args,-Vars)
%  converts p(c1,c2,c3) or the like into a schema
%  for one of the conjuncts of the multiply distributed
%  formula, and provides lists of argument variables and
%  the corresponding arguments.

make_conjunct(Term,(ElementTests,Result),Args,Vars) :-
```

```
  Term =.. [Pred|Args],
  list_of_variables(Args,Vars),
  Result =.. [Pred|Vars],
  make_element_tests(Args,Vars,ElementTests).

% make_element_tests(+Args,+Vars,-Result)
%  supplies the "element-of" tests to go into a
%  conjunct of a multiply distributed formula.

make_element_tests([],[],true).
  % Adds a redundant "true" to the formula.

make_element_tests([dist(Arg)|Args],[Var|Vars],
                      (element(Var,Arg),Tests)) :-
  !,
  make_element_tests(Args,Vars,Tests).

make_element_tests([Arg|Args],[Var|Vars],Tests) :-
  % this argument is not marked for distribution,
  % so instantiate the variable to a fixed value
  Var = Arg,
  make_element_tests(Args,Vars,Tests).

% list_of_variables(+L1,-L2)
%  given a list, produces an equal-length list
%  of variables. (For simplicity, we generate
%  variables for all arguments, even those that
%  are not going to be distributed.)

list_of_variables([_|T],[_|T1]) :-
  list_of_variables(T,T1).

list_of_variables([],[]).
```

Here $P(\lhd c_1, \lhd c_2, \lhd c_3)$ (for example) is represented in Prolog as

`p(dist(c1),dist(c2),dist(c3))`

and is converted by the predicate `multiply_distribute/2` into:

```
(dist(c1,lambda(X,((element(X,c1),element(Y,c2),
```

```
                  element(Z,c3),true),p(X,Y,Z)))),
 dist(c2,lambda(Y,((element(X,c1),element(Y,c2),
                  element(Z,c3),true),p(X,Y,Z)))),
 dist(c3,lambda(Z,((element(X,c1),element(Y,c2),
                  element(Z,c3),true),p(X,Y,Z)))),
 true)
```

Here the redundant instances of `true` (a query that always succeeds) allow the use of a simpler translation algorithm; it is less work to leave them in, and eventually execute them, than to do extra computation to take them out.

10 Example of a Multiply Distributive Query

Here is an example of multiple distribution in use. Consider the knowledge base:

```
% a collection of 3 tenants
element(t1,tenants).
element(t2,tenants).
element(t3,tenants).

% a collection of 4 keys
element(k1,keys).
element(k2,keys).
element(k3,keys).
element(k4,keys).

% a collection of 2 landlords
element(l1,landlords).
element(l2,landlords).

% who gave what to whom
gave(l1,k1,t1).
gave(l2,k2,t2).
gave(l1,k3,t3).
gave(l2,k4,t2).
```

To ask the question "Did the landlords give the keys to the tenants?" in the multiply distributive sense, one encodes it as

```
gave(dist(landlords),dist(keys),dist(tenants))
```

and then uses `multiply_distribute` to translate it into an executable query, and finally passes the query to the Prolog system:

```
% "Did the landlords give the keys to the tenants?"

?- multiply_distribute(
     gave(dist(landlords),dist(keys),dist(tenants)),
     Query
   ),
   Query.

yes
```

That is: "Is it true that each landlord gave some key to some tenant, and each key was given by some landlord to some tenant, and each tenant was given some key by some landlord?" The answer, in this situation, is yes even though the numbers of keys, tenants, and landlords are all different.

11 Complexity of multiple distribution

A k-fold distribution is a conjunction of k distributive queries. This conjunction is purely logical; the earlier queries do not instantiate variables for the later ones, and there is no backtracking. Thus, the complexity is k times that of one of the conjoined queries.

Each conjoined query, in turn, picks an element from each of k collections before executing the queried predicate. If each collection has n members, there are n^k combinations of elements to try. Thus, the complexity of the whole computation, comprising k conjoined queries, is proportional to kn^k, exponential in k (which is always small, since it is the number of arguments of a predicate) and polynomial in n (which can be large).

12 Multiple cumulation?

It is not clear to me whether English also has multiple cumulation. Consider a situation where:

- *the farmers* denotes $\{f_1, f_2, f_3\}$;
- *the donkeys* denotes $\{d_1, d_2, d_3\}$;
- farmer f_1 belongs to an unnamed collection that fed d_1;
- farmer f_2 fed an unnamed collection of donkeys that includes d_2;
- farmer f_3 belongs to another unnamed collection of farmers that fed an unnamed collection of donkeys that includes d_3.

In that situation, is *The farmers fed the donkeys* true?

The reading that makes it true is, at best, difficult to get, and there's a reason. Consider the way English plurals relate to events or predicates:

- A collective ascribes a property to a collection;
- An ordinary distributive ascribes a property to all of the elements of a collection;
- A multiple distributive ascribes a relation to unnamed tuples covering all the elements of the specified collections;
- An ordinary cumulative ascribes a property to all the elements of a collection, or other unnamed collections that contain them;
- A multiple cumulative (if there is such a thing) ascribes a relation to unnamed tuples covering all the elements of the specified collections, and/or other unnamed collections containing them.

Of these, only multiple cumulatives use two types of unnamed things at once: not only unnamed tuples composed of elements of the respective collections, but also unnamed collections containing those elements. Apparently, the human mind balks at using a simple plural to refer to two different categories of unnamed entities at once.

13 Conclusion

The semantics of natural language plurals is still an open research question [3, 7, 12]. The implementation presented here is a step toward a more satisfactory interpretation of natural language plurals in queries without sacrificing computational tractability, logical clarity, or conventional database semantics.

References

[1] Jon Barwise, "Branching quantifiers in English," *Journal of Philosophical Logic,* vol. 8, pp. 47–80, 1979.

[2] Jon Barwise and Robin Cooper, "Generalized quantifiers and natural language," *Linguistics and Philosophy,* vol. 4, pp. 159–219, 1981.

[3] Enrico Franconi, "A treatment of plurals and plural quantifications based on a theory of collections," *Minds and Machines,* vol. 3, pp. 453–474, 1993.

[4] Brendan S. Gillon, "The readings of plural noun phrases in English," *Linguistics and Philosophy,* vol. 10, pp. 199–220, 1987.

[5] Jerrold J. Katz, *Propositional Structure and Illocutionary Force,* Hassocks (England): Harvester Press, 1977.

[6] Ruth M. Kempson and Annabel Cormack, "Ambiguity and quantification," *Linguistics and Philosophy,* vol. 4, pp. 259–309, 1981.

[7] Peter Lasersohn, *Plurality, Conjunction, and Events,* Dordrecht: Kluwer, 1995.

[8] Godehard Link and Hinrich Schütze, "The treatment of plurality in L_{LILOG}," in O. Herzog and C.–R. Rollinger, eds., *Text Understanding in LILOG,* pp. 342–352, Berlin: Springer, 1991.

[9] C. S. Mellish, *Computer Interpretation of Natural Language Descriptions,* Chichester (England): Ellis Horwood, 1985.

[10] Remko J. H. Scha, "Distributive, collective, and cumulative quantification," in Jeroen Groenendijk, Theo M. V. Janssen, and Martin Stokhof, eds., *Truth, Interpretation, and Information,* pp. 131–158, Dordrecht: Foris, 1984.

[11] John F. Sowa, "Toward the expressive power of natural language," in John F. Sowa, ed., *Principles of Semantic Networks,* pp. 157–190, San Mateo, Calif.: Kaufmann, 1991.

[12] M. H. van den Berg, "Plurality," in *The Encyclopedia of Language and Linguistics,* vol. 6, pp. 3198–3200, Oxford: Pergamon, 1994.

[13] Jan van Eijck, "Generalized quantifiers and traditional logic," in Johan van Benthem and Alice ter Meulen, eds., *Generalized Quantifiers and Natural Language,* pp. 1–19, Dordrecht: Foris, 1985.