From English to Prolog via Discourse Representation Theory ACMC Research Report 01-0024

Michael A. Covington
Donald Nute
Nora Schmitz
David Goodman

Advanced Computational Methods Center
University of Georgia
Athens, Georgia 30602

April 1988

1 Introduction/Abstract

This is a preliminary report on a set of techniques for translating the discourse representation structures (DRSes) of Kamp (1981) into semantically equivalent clauses in a slightly extended form of Prolog. Together with discourse representation theory (DRT) itself, these techniques yield a system for translating English into Prolog. A working prototype has been built using Quintus Prolog on a VAX workstation[1].

[1]This work was supported by National Science Foundation Grant Number IST-85-02477. Opinions and conclusions expressed here are solely those of the authors.

2 Methodology

Both computational linguistics and natural language semantics have reached the stage where precise solutions to small problems are of greater value than vague proposals for solutions to large problems. This paper is offered in that spirit.

Discourse representation theory (DRT) covers only a subset of the semantic phenomena of English, not including, for example, plurals or definite noun phrases. Prolog, even with our extensions, is less powerful than classical logic, which in turn is less powerful than natural language. So any system that translates English to Prolog via DRT can accept no more than a subset of English. Nonetheless, this subset is large enough to display clearly the solutions to a variety of problems of semantic representation — solutions that will carry over into a more comprehensive theory when one is developed.

The following are among the design goals of our system:

(1) To preserve the main advantage of Prolog — its ability to perform inference more rapidly than a full resolution theorem prover by constraining its search space. This contrasts with the approaches taken by Guenthner (1986), Kolb (1985, 1987), and a group at Imperial College, London (Gabbay, personal communication), who built inference engines for discourse representation structures.

(2) To preserve the main advantage of DRT — the ability to represent a text as a single unit by focusing on its semantic and logical structure rather than on individual sentences or propositions, and hence incorporating the textual context that is necessary for interpreting each subsequent sentence.

(3) To represent knowledge and its semantic structure in a form as close as possible to ordinary Prolog clauses.

We extend Prolog by adding conditional queries (the ability to ask whether an "if-then" relation holds), explicit negation (rather than negation as failure), and a table of identity that enables non-unifiable terms to be treated as equivalent for some purposes.

3 Discourse representation theory

3.1 Overview of DRT

Discourse representation theory (DRT), introduced by Kamp (1981), is a more satisfactory representation of natural language semantics than earlier representations such as classical logic or Montague grammar. The key properties of DRT include the following:

(1) It is not sentence-based. DRT constructs representations of discourses, not sentences. These representations are called discourse representation structures (DRSes). Rather than building a DRS for each sentence, we build a single DRS to which each sentence contributes some material, using the previously present material as context.

(2) It is not tied closely to the syntax of English nor to a particular theory of syntax. Any syntactic analysis suitable for determining meaning can be used in an implementation of DRT.

(3) Structural restrictions on the accessibility of anaphoric antecedents are predicted correctly.

(4) A theory of truth-conditions is built in. Truth is defined in terms of embedding the DRS in a model. Thus DRT preserves the advantages of predicate logic as a representation language while bringing the formalization closer to the structure of natural language.

3.2 Discourse representation structures

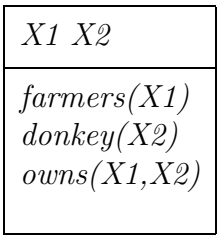
A discourse representation structure is an ordered pair `<U,Con>` where`U` is a universe of discourse, i. e., a set of discourse entities, and `Con` is a set of conditions, i. e., predicates or equations that these entities must fulfill. For example, the sentence

```
A farmer owns a donkey.
```

is represented by the DRS

```
U={X1,X2}
Con={farmer(X1),donkey(X2),owns(X1,X2)}
```

or in more usual notation:



The truth of a DRS is defined by trying to embed it in a model. A model comprises a domain `D`, i. e., a set of entities that can be discussed, and an interpretation function I that maps every n-place predicate of the language onto a set of n-tuples of elements of `D`, and maps every logical constant of the language onto an individual in `D`.

Intuitively, `D` is the set of things you can talk about (including imaginary as well as real entities); `I(farmer)` is the set of elements of `D` that are farmers; and `I(owns)` is the set of pairs of elements of `D` such that the first one owns the second one.

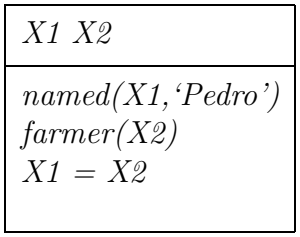
To embed a DRS into a model, assign each discourse variable (`X1` and `X2` above) to an element of `D`. The DRS is true in the model if and only if it can be embedded in such a way that all of the conditions are satisfied. Thus, the DRS above is true in a particular model if, within that model, it is possible to assign `X1` to a farmer and `X2` to a donkey which is owned by that farmer.

3.3 Equations

Our implementation treats is as a predicate that requires two discourse variables to refer to the same individual. For example, the sentence

```
Pedro is a farmer.
```

is represented as:



This is a temporary measure that will be replaced by a fully developed table of identity (section 6.1 below).

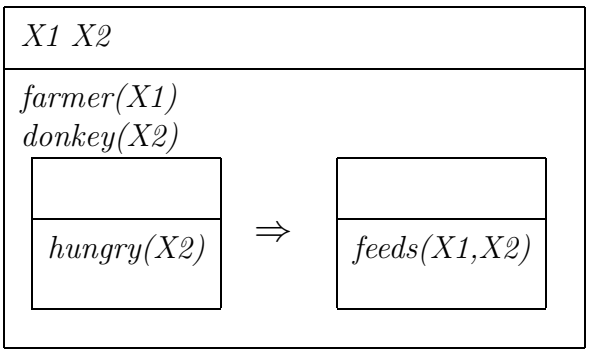
3.4 Conditionals ("if-thens")[2]

An "if-then" relationship between two propositions is expressed by a special type of DRS condition. Thus

```
A farmer owns a donkey. If it is hungry he feeds it.
```

is represented as:

[2]We will call these "if-thens" rather than "conditionals" because the term "conditional" is too easily confused with "condition," and because some DRT "if-then" structures correspond to sentences such as All men are mortal, which are called universals rather than conditionals in ordinary philosophical discourse.



Crucially, one of the conditions of this DRS consists of two more DRSes joined by `'==>'`. In general, the condition

```
DRS1 ==> DRS2
```

is satisfied (in a particular model) if and only if

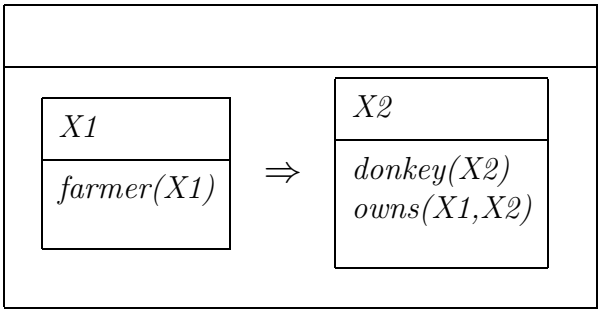
> every embedding that satisfies `DRS1` also satisfies, or can be extended to satisfy, `DRS2`.

By saying 'extended' we leave open the possibility that `DRS2` may contain new variables of its own, which will need to be assigned to individuals in the universe of discourse. This extends the embedding of `DRS1` and of the surrounding larger DRS, which did not assign these variables.

Note that the new variables of `DRS1`, if any, have implicit universal quantifiers, and the new variables of `DRS2` have implicit existential quantifiers. To satisfy the whole condition, we must find some embedding of `DRS2` to go with every embedding of `DRS1`. A universal sentence such as

> Every farmer owns a donkey.

is actually a kind of if-then; it means "If X is a farmer, then X owns a donkey" and is represented as:



That is: "Every embedding that assigns `X1` to a farmer can be extended to form an embedding that assigns `X2` to a donkey owned by `X1`." Or in ordinary language, "For every farmer `X1`, there is some donkey `X2` owned by `X1`."

Because the quantifiers are implicit and there are no sentence boundaries, some familiar problems of quantifier scope no longer arise. Consider for example the discourse

```
A farmer owns a donkey. He feeds it.
```

In classical logic the first sentence would be

```
(Some X)(Some Y) farmer(X) & donkey(Y) & owns(X,Y).
```

The second sentence must somehow have access to the same donkey and the same farmer, but if it is treated as a separate proposition, its variables cannot be under the scope of the same quantifiers. Any translation into classical logic will therefore need a special, explicit rule for combining two propositions into one. In DRT, on the other hand, this combining is implicit and automatic; sentences keep being added to the same DRS until there is a reason not to do so.

3.5 Anaphora and accessibility

A DRS can use its own discourse variables and those of DRSes which are superordinate to it. Whenever a DRS contains another DRS, the outer DRS is superordinate to the inner one. Further, the left side of an if-then is superordinate to the right side, and superordinateness is transitive.

When one DRS is superordinate to another, we will say that the second DRS is subordinate to the first.

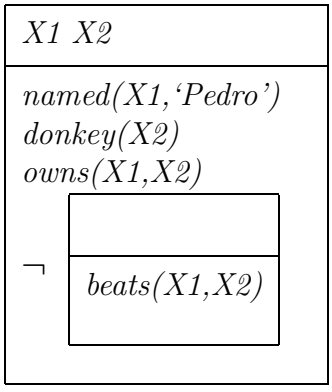
This restriction makes correct predictions about the accessibility of antecedents to anaphors (Kamp 1981).

3.6 Negation

DRT represents negated assertions as DRSes within DRSes, preceded by the negation operator (here written NEG). Thus

```
Pedro owns a donkey. He does not beat it.
```

is represented by:



A condition of the form `NEG DRSx` is satisfied if and only if there is no extension of the current embedding that makes DRSx true.

Thus the above DRS is true in a model if there is an assignment of `X1` and `X2` such that `X1` is Pedro and `X2` is a donkey owned by Pedro, and this assignment cannot be extended to satisfy the condition that `X1` beats `X2`.

We note in passing a logical problem: what if Pedro owns two donkeys? DRT allows us to choose the "wrong" donkey — the one he does not beat — and use it to satisfy the above DRS even though it is not clear that this move would be faithful to the meaning of the original English-language discourse. This is related to the problem that present versions of DRT do not address plurals at all.

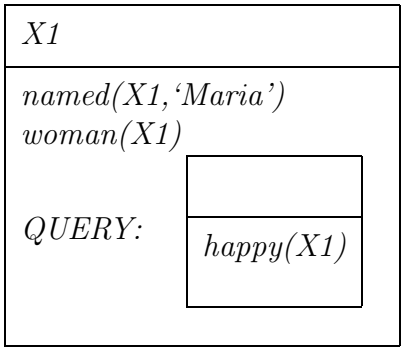
3.7 Questions

Kamp's DRT provides no way to distinguish statements from questions. We treat yes-no questions as DRSes within DRS-conditions, marked with the operator `QUERY`. Conditions of this type can appear only in the topmost DRS, not in subordinate DRSes.

The most common use of questions in a natural language understanding system is to request information from the computer. Accordingly, we posit that a question does not contribute to the knowledge base, but rather directs the hearer to test the truth of the included DRS and report the result. Thus the discourse:

```
Maria is a woman. Is she happy?
```

is represented as



A more general representation of questions would also be able to specify variables whose values should be reported (e.g., "Who owns a donkey?").

4 DRS construction

The DRS-building algorithm used in this project is that described by Covington and Schmitz (1988), a unification-based grammar modeled closely on that of Johnson and Klein (1986).

A DRS is represented by a Prolog term `drs(U,Con)` where `U` is a list of discourse variables and `Con` is a list of structures containing these variables. In Prolog, the fastest way to build a list is to add material at the beginning. As a result, the elements of `U` and `Con` appear in the opposite of the order in which they occur in the discourse; this obviously has no effect on their truth conditions. Further, `Con` contains additional predicates specifying the gender of every discourse variable, to make it possible to find the antecedents of anaphoric pronouns.

In the Covington and Schmitz implementation, discourse variables were represented by unique integers, but here they are unique Prolog variables. This makes it possible to perform skolemization (described below) by simply instantiating heretofore uninstantiated Prolog variables.

For example, A man owns a donkey is represented as

```
drs([X2,X1],[owns(X1,X2),
             gender(X2,n),
             donkey(X2),
             gender(X1,m),
             man(X1)].
```

Proper names are represented by the predicate named, as in

```
named(X3,'Pedro')
```

thus leaving open the possibility of two or more people with the same name, or two or more names for the same person.

Special DRS-conditions of the following forms represent, respectively, negated assertions, questions, and if-thens:

```
neg(drs(...,...))

query(drs(...,...))

ifthen(drs(...,...))
```

These are elements of Con just like ordinary conditions.

The implementation has the ability to resolve anaphors, and for this purpose, it uses the `gender` predicate to specify the gender of each discourse variable. The antecedent of he, she, or it is the most recently mentioned discourse variable of the appropriate gender which is found in the current DRS or a DRS superordinate to it.

The discourse variable for the anaphor and the discourse variable for the antecedent are unified by the anaphora resolver so they are for all practical purposes identical.

5 Translation into Prolog

5.1 Discarding irrelevant information

The first step in translating a DRS into a set of Prolog clauses and/or queries is to "clean up" the output of the DRS-builder by discarding irrelevant information. This involves discarding all the DRS-conditions that describe gender.

Further, the verb is introduces conditions that say that two discourse variables must be the same individual. For example, the sentence

```
Pedro is a farmer.
```

is represented (ignoring gender information) as:

X1 X2
named(X1,'pedro') *farmer(X2)* *X1 = X2*

In the present implementation, the clean-up routine simply unifies `X1` with `X2`, giving:

X1
named(X1,'Pedro') *farmer(X1)*

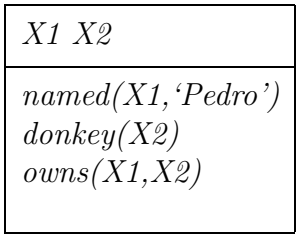
We will see in Section 6.1 that in some cases, two equated discourse variables may not be unifiable after skolemization. A more adequate implementation would leave the equation in the DRS and later mark the equated discourse variables as equivalent in the table of identity.

5.2 Simple questions

The easiest sentences to translate into Prolog are simple questions such as

```
Does Pedro own a donkey?
```

which are, in effect, instructions to test the truth of a DRS in the model defined by the current knowledge base. The queried DRS is:



We want to know whether there is an assignment of values to `X1` and `X2` such that `named(X1,'Pedro')`, `donkey(X2)`, and `owns(X1,X2)` will all be satisfied.

Ex hypothesi, the interpretations of named, donkey, and owns — that is, the sets of argument tuples that satisfy them — are given by the definitions of these predicates in Prolog. Thus our task is exactly equivalent to solving the Prolog query:

```
?- named(X1,'Pedro'), donkey(X2), owns(X1,X2).
```

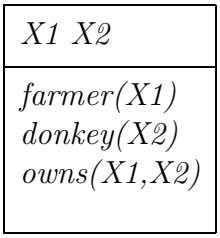
Variables in Prolog queries have implicit existential quantifiers; so do free variables in DRSes whose truth is being tested. For this type of sentence, then, the semantics of Prolog and of DRT match closely.

5.3 Simple assertions

Assertions — statements of fact — are slightly more complicated to handle. Clearly, the DRS for

```
A farmer owns a donkey.
```

— namely



should be translated by adding something to the knowledge base. But what should we add?

We can't just leave the variables free, generating the clauses

```
farmer(X1).
donkey(X2).
owns(X1,X2).
```

because free variables in Prolog facts and rules have implicit universal quantifiers. In satisfying a Prolog query, free variables match anything, so these clauses would mean "Anything is a farmer; anything is a donkey; anything

owns anything." In fact, because like-named variables in different clauses are distinct, we haven't even succeeded in saying that the donkey is the same as the thing that is owned.

Rather, we must provide "dummy names" for existentially quantified entities — one-element lists with distinct integers inside. We will call the hypothetical farmer [1] and call the hypothetical donkey [2], and assert the facts:

```
farmer([1]).
donkey([2]).
owns([1],[2]).
```

Now if we ask "Is there a farmer that owns a donkey?" — i. e., the query

```
?- farmer(X), donkey(Y), owns(X,Y).
```

we get the answer that there is, and that the farmer is known as [1] and the donkey is known as [2].

This is a special case of skolemization, to be dealt with below. In a later implementation, we will use a table of identity to ensure that distinct names can be recognized as referring to the same individual when necessary.

Even proper names are not treated as logical constants. Thus, `Pedro owns a donkey` goes into Prolog as:

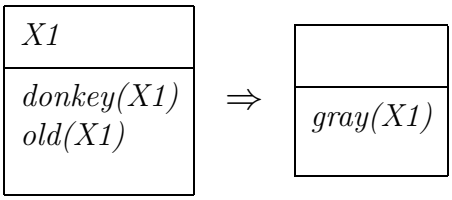
```
named([1],'Pedro').
owns([1],[2]).
donkey([2]).
```

There would be little advantage in using proper names as logical constants because no proper names are available for most individuals. Further, by using the predicate `named` we allow for ambiguous names and for individuals with more than one name.

5.4 Simple if-thens

In the simplest case, an if-then DRS condition is equivalent to a Prolog rule. Consider the DRS condition:

```
Every old donkey is gray.
```



This condition is true if and only if every embedding that satisfies its antecedent — that is, every embedding that assigns `X1` to an old donkey — also assigns `X1` to a gray donkey.

If this if-then condition is known to be true, then, for any X, we can infer that X is gray if we can prove that X is old and is a donkey. This is exactly equivalent to the Prolog rule:

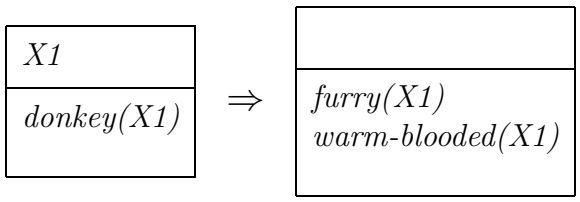
```
gray(X) :- donkey(X), old(X).
```

Here again Prolog semantics exactly matches DRT.

5.5 Distributing consequents

A minor syntactic problem arises if the consequent contains more than one predicate, as in the following:

```
Every donkey is furry and warm-blooded.
```



A Prolog rule cannot have two predicates in its consequent; rules of the form

```
furry(X1), warm-blooded(X1) :- donkey(X1).
```

are not permitted. We deal with this by defining, for purposes of internal representation, an operator ::- which is like the usual 'if' (:-) except that:

```
(1) Both the antecedent and the consequent can be compound goals;

(2) Queries can be headed by ::-, i. e., one can ask whether an if-
 then relation holds.
```

Rules headed by ::- are never asserted directly into the knowledge base. Instead, when an if-then is to be asserted, the consequent is broken up and a series of ordinary Prolog rules is generated. A rule of the form

```
a, b, c ::- d, e, f.
```

is added to the knowledge base as the three rules:

```
a :- d, e, f.
b :- d, e, f.
c :- d, e, f.
```

This is known as distributing consequents. For very complex consequents, a more compact representation could be obtained by creating a new symbol x (taking as arguments are all the arguments of d, e, and f, if any) and asserting, instead, the four rules:

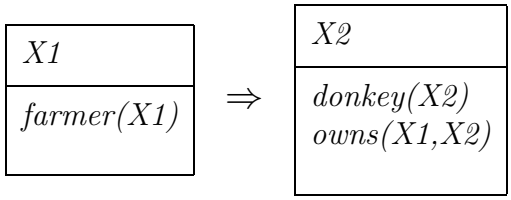
```
x :- d, e, f.
a :- x.
b :- x.
c :- x.
```

This was not done in the current implementation.

5.6 Skolemization

If the consequent of an if-then introduces new variables, these variables have implicit existential quantifiers, which Prolog cannot represent. Thus

```
Every farmer owns a donkey.
```



is true if and only if, for every embedding that assigns `X1` to a farmer, there is some embedding that assigns `X2` to a donkey owned by `X1`.

In Prolog (extended with ::-), if we were to say

```
donkey(X2), owns(X1,X2) ::- farmer(X1).
```

we would be saying that every farmer owns every donkey. If we gave the donkey a dummy name, say [92], we would not be much better off, because

```
donkey([92]), owns(X1,[92]) ::- farmer(X1).
```

would mean that every farmer owns the same donkey (this particular one identified as [92]).

What we want to say is that every farmer owns a `different` donkey. This is achieved by giving the donkey a dummy name that contains X1 so that, in effect, the name depends on the assignment of X1:[3]

```
donkey([92|X1]), owns(X1,[92,X1]) ::- farmer(X1).
```

Then if we assign X1 to an individual named [83] who is a farmer, we get a donkey named [92,83]; if we assign X1 to another farmer named [17], we get a donkey named [92,17]; and so on. The table of identity described below will enable us to equate [92,17] with a donkey already known under another dummy name if necessary.

This is a form of skolemization, the method of eliminating existential quantifiers proposed by Skolem (1928). Skolemization replaces every existentially quantified variable with a function whose value is an individual that satisfies the formula. For instance, in classical logic,

```
(All X) (Some Y) g(X,Y)
```

can be replaced by

```
(All X) g(X,f(X))
```

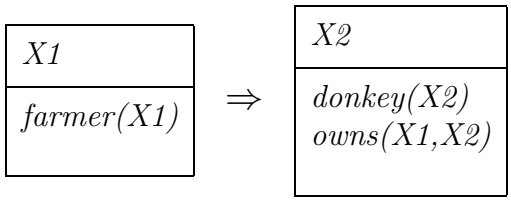
where f is a function which, given a value of X, yields a value of Y that would satisfy the original formula. The existential quantifier is nothing more than a claim that such a function exists. Its arguments are all the universally quantified variables in the scope of whose quantifier the existentially quantified variable occurs. Thus if every farmer owns a donkey, there is (or can be) a

[3]Recall that | divides a list into head and tail in Prolog, so that [a | [b,c,d]] is equivalent to [a,b,c,d].

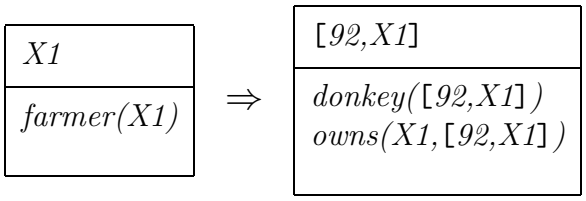
different donkey for each farmer, and the Skolem function for the existentially quantified donkey must take as an argument the universally quantified farmer.

In large formulae, there are often some universally quantified variables that can be shown to be irrelevant to a particular predicate; these need not be arguments of the Skolem function, though it does no harm to use them. Andrews (1986:123–127) compares various methods of skolemization and shows how to eliminate unneeded arguments.

To skolemize an if-then structure in a DRS, we replace all the variables pin the consequent with Skolem functions of the variables in the antecedent. A Prolog term such as `[92|X1]` has a value that is a function of the value of `X1`; this makes it an appropriate way to encode a Skolem function. Thus, for example, the DRS-condition



is transformed into:



which goes into extended Prolog as

```
donkey([92,X1]), owns(X1,[92,X1]) ::- farmer(X1)
```

and is asserted as the two Prolog rules:

```
donkey([92,X1]) :- farmer(X1).
owns(X1,[92,X1]) :- farmer(X1).
```

The number 92 here is of course arbitrary, produced by a routine that returns a heretofore unused integer every time it is called.

5.7 Queried if-thens

"Is every old donkey gray?" may mean either of two things. It may mean, "If you are told that something is an old donkey, can you deduce that it is gray?" Or it may mean "Are all the old donkeys that you know of gray?" We will call these the deductive and inductive approaches to querying an if-then.

The deductive approach is easily implemented using a technique suggested by Gabbay and Reyle (1984): invent a hypothetical individual, temporarily assert that it is an old donkey, and test whether it can be deduced to be gray. We put this into our implementation by defining `::-` as a Prolog predicate so that queries of the form `A ::- B` can be answered:

```
(A ::- B) :- skolemize(B,[]),
             asserta(B),
             test(A,Result),
             retract(B),
             Result = yes.
```

That is, to test whether `A ::- B` holds, first replace all the variables in `B` with dummy names, then temporarily add `B` to the knowledge base,[4] then try to deduce `A`. Here `test(A,Result)` is a metalogical predicate that instantiates `Result` to `yes` or `no` depending on whether `A` succeeds; `test` itself succeeds in either case.

[4] A practical implementation would allow `B` to be a compound goal and would replace `asserta` and `retract` with procedures that assert and retract all of the conjuncts of which `B` is composed.

Note that this works only as long as `A ::- B` does not rely on negation as failure. If we were trying to test the validity of

```
gray(X) ::- donkey(X)
```

and the knowledge base contained only the rule

```
gray(X) :- donkey(X), not young(X).
```

we would get wrong results. We would assert something like `donkey([27])` and then query `gray([27])`. The hypothetical donkey would be taken to be "not young" simply because we did not assert that it is young, and `gray([27])` would therefore be derivable, leading to the mistaken conclusion that `gray(X)` is derivable from `donkey(X)` in all cases.

We avoid this problem by using an explicit (positive) way of representing negation (Section 6.2).

The inductive approach says that `A ::- B` is true if (a) there are no cases in the knowledge base that satisfy `B` without satisfying `A`, and (b) there are enough cases that satisfy `B` to warrant making an inductive generalization. That is, all donkeys are gray if the knowledge base contains a sufficient number of donkeys and none of them fail to be gray.

The value of this "sufficient number" is open to question; in a human thinker it depends on, among other things, the size of the overall sample, the rarity of the phenomenon being described, the expected regularity, and the thinker's training in statistics. Here we will take it to be 1. Thus we add another rule to the definition of the predicate `::-` as follows:

```
(A ::- B)  :-  not (B, not A),   /* No counterexamples */
               B.                /* One positive example */
```

N

This provides all that is needed to handle queries that ask whether an if-then is true.

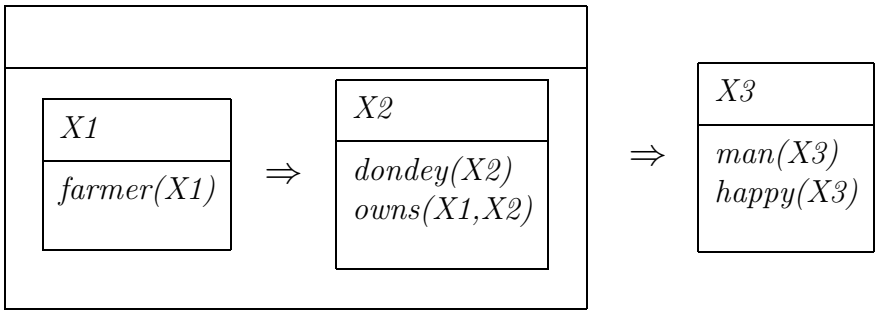
Note that queried if-thens are not skolemized. In the context of a query, free Prolog variables are taken to be existentially quantified, exactly as the DRT semantics requires.

5.8 Nested if-thens

5.8.1 Nested antecedents: Prolog subgoals

Because if-thens can be queried on the Prolog level, they can appear as subgoals in a Prolog rule. This provides a way to handle an if-then within the antecedent of an if-then. Consider for example the DRS:

If every farmer owns a donkey, a man is happy.



Because the antecedent of an if-then is like a query, the inner if-then will not be skolemized.[5] `X3` is skolemized by a Skolem function with no arguments; its arguments should be the variables of the antecedent as a whole, and in this example there are none. So the resulting Prolog clauses are:

```
man([57])   :- ( (donkey(X2), owns(X1,X2)) ::- farmer(X1) ).
```

[5]The present implementation erroneously skolemizes the antecedents of all if-then structures, even in queried contexts.

```
happy([57]) :- ( (donkey(X2), owns(X1,X2)) ::- farmer(X1) ).
```

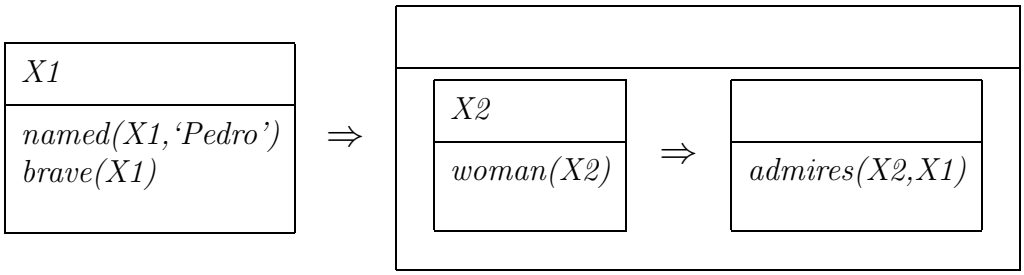
That is: `[57]` is a man and is happy if it is the case that for every farmer `X1`, there is a donkey `X2` that is owned by `X1`.

5.8.2 Nested consequents: Exportation

Nesting in the consequent of an if-then is more complicated. Consider the sentence:

```
If Pedro is brave then every woman admires him.
```

The corresponding DRS is:



Naively, this should go into Prolog as something like

```
( admires(X2,X1) ::- woman(X2) ) :- named(X1,'Pedro'),
                                    brave(X1).
```

But this will not do. The above DRS can be used to infer that a particular woman — Maria, for instance — admires Pedro. The Prolog rule cannot; at best it would match a query asking whether the if-then relation `admires(X2,X1) ::- woman(X2)` holds.

In cases like this we employ the procedure of exportation, familiar from classical logic, which enables us to transform

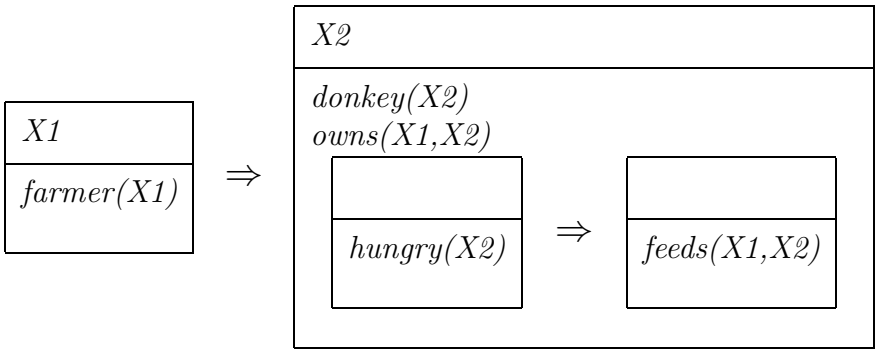
```
if P then (if Q then R)
```

into

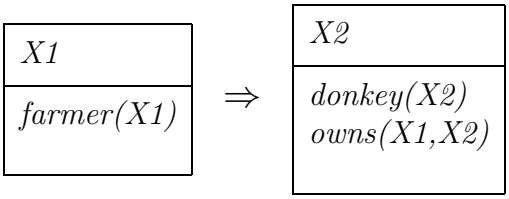
```
if P and Q then R.
```

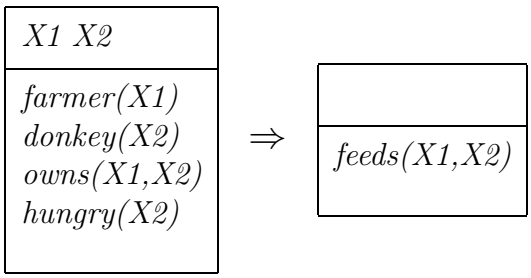
Actual DRSes are not as neat as the formulae of classical logic, and practical questions arise, illustrated by the following example:

```
Every farmer owns a donkey, and if it's hungry he feeds it.
```



Like all if-thens, this is not a self-standing DRS, but rather a condition in a larger DRS. Exportation should break it into two conditions:





The first of these comprises the material in the original if-then that was not affected by exportation. The second is the result of exporting the inner if-then. Its consequent is the consequent of the inner if-then. Its antecedent is a combination of the antecedent of the outer if-then, the antecedent of the inner if-then, and, crucially, the consequent of the outer if-then, except for the part actually being exported. If `donkey(X2)` were not in the antecedent of the second structure, nothing would say what kind of animal the farmer was feeding.

This procedure, if followed ruthlessly, leads to excessively complex formulas. Suppose the outer if-then has two if-thens in its consequent, and we are exporting one of them. The antecedent of the result should contain, inter alia, all the other conditions from the consequent of the original outer if-then. And one of these is itself an if-then. This is not forbidden — after all, nesting of if-thens in the antecedent is permitted. But the two if-thens in the original consequent cannot use each other's variables, since neither is superordinate to the other. So it seems unlikely, if not impossible, for one of them to play an essential role in identifying the entities used by the other.

The actual implementation takes a more modest approach. The if-then structure created by exportation has in its antecedent all the variables and conditions of the antecedents of the inner and outer if-thens, and all the variables, but only some of the conditions, of the consequent of the outer if-then. Specifically, it has the conditions which (a) are simple, not involving nesting of if-thens, and (b) occurred prior to the inner if-then in the discourse. This seems to be adequate for natural langauge in actual use.

6 Remaining issues

6.1 The table of identity

Individuals introduced into the discourse under different names or Skolem functions may later be discovered to be identical. For example:

```
Thales observes Hesperus.
Aristotle observes Phosphorus.
Hesperus is (the same as) Phosphorus.
```

The DRS for the first two sentences is:

X1 X2 X3 X4
named(X1,'Thales') *named(X2,'Hesperus')* *observes(X1,X2)* *named(X3,'Aristotle')* *named(X4,'Phosphorus')* *observes(X3,X4)*

Assuming incremental processing, each DRS condition will be skolemized and entered into the knowledge base as soon as its sentence is processed, if not sooner. Thus, at the end of the second sentence, we have:

```
named([1],'Thales').
named([2],'Hesperus').
observes([1],[2]).
named([3],'Aristotle').
named([4],'Phosphorus').
observes([3],[4]).
```

But now we discover from the third sentence that Hesperus and Phosphorus are different names for the same entity. We would like to note this fact in such a way that we can later ask,

```
Does Aristotle observe Hesperus?

?- named(X,'Aristotle'), named(Y,'Hesperus'), observes(X,Y).
```

and get "yes" as a reply. But Hesperus and Phosphorus have been skolemized as non-unifiable entities, `[3]` and `[4]`, so we can't just unify them.

We have two options. One is to go back through the knowledge base and change all occurrences of `[4]` to `[3]`, or vice versa. The other, which we actually adopt, is to maintain a table of identity so that non-unifiable terms can be recognized as equivalent. We will then formulate the query as:

```
?- named(X,'Aristotle'), identical(X,X1),
   named(Y,'Hesperus'),  identical(Y,Y1),
   observes(X1,Y1).
```

Each argument — `X` and `Y` — is now passed through an extra level of indirectness. On the first attempt, the `identical` predicate simply unifies its two arguments, instantiating `X1` to `X` and `Y1` to `Y`, so that the query works exactly like our earlier proposal. But if this fails, it looks at the table of identity and tries to instantiate `X1` and `X` to two different terms that refer to the same object, and likewise with `Y1` and `Y`. Thus, if `[3]` and `[4]` have been entered in the table, they will be treated as equivalent.

This query looks verbose, but in fact it is generated by a simple transformation of the original query: just insert calls to `identical` for any arguments that are used in more than one subgoal. An alternative would be to modify the inference engine so that it consults the table of identity when solving queries.

The table of identity has been prototyped but not yet integrated with the main English-to-Prolog translation system. The problem is that identity is a

symmetric (commutative) and transitive relation. Thus it would require the rules

```
identical(X,Y) :- identical(Y,X).
identical(X,Z) :- identical(X,Y), identical(Y,Z).
```

which would cause loops under some conditions by calling themselves endlessly.

Our solution is to store the table as an identity matrix represented by a set of Prolog facts. The predicate identical is defined by only facts, not rules, and hence cannot loop. All the appropriate facts are generated when an entry is made into the table.

Initially, only the clause

```
identical(X,X).
```

is in the knowledge base; this corresponds to the main diagonal of the matrix and ensures that any term will be treated as identical to itself. Additional entries are made by a predicate `make_identical`. Calling `make_identical(a,b)` , for example, adds not only the facts

```
identical(a,b).
identical(b,a).
```

but also any other facts called for by transitivity. For example, if identical(a,c) were already in the knowledge base, then `make_identical(a,b)` would add four facts:

```
identical(a,b).
identical(b,a).
identical(c,b).
identical(b,c).
```

The Prolog code to do this reflects the 2x2 dimensionality of the identity matrix:

```
make_identical(X,Y) :-
      setof(X1,identical(X,X1),XMatches),
      setof(Y1,identical(Y,Y1),YMatches),
      make_id_list_squared(XMatches,YMatches).

make_id_list_squared([],_).

make_id_list_squared([H|T],List) :-
      make_id_list(H,List),
      make_id_list_squared(T,List).

make_id_list(_,[]).

make_id_list(X,[H|T]) :-
      (identical(X,H) ; assert(identical(X,H))),
      (identical(H,X) ; assert(identical(H,X))),
      make_id_list(X,T).
```

If the table of identity contains information for n individuals, it will contain at most $\underline{n}^2$ clauses for identical, and usually considerably fewer.

6.2 Negation

The current implementation does not handle negation. We propose to handle negation as a metalogical predicate neg, as in d-Prolog (Nute and Lewis 1986:6-7). Thus it will be possible to assert

```
neg donkey([2]).
```

to say "Individual [2] is not a donkey."

Negative statements are not defined in terms of affirmative ones, nor vice versa. From the inference engine's point of view, `donkey([2])` and `neg donkey([2])` are completely unrelated facts and each must be queried separately. Crucially, neither of them follows from the inability to derive the other. The knowledge base could even contain both of them, though it would then express a contradiction. Special routines could of course be written to detect contradictions and identify the premises from which they arise.

This solves the problem with queried if-thens that we mentioned earlier. In this system, the sentence "Every donkey that is not young is gray" translates to:

```
gray(X) :- donkey(X), neg young(X).
```

Now, in order to test whether all donkeys are gray, we assert the hypothesis

```
donkey([23]).
```

and see if we can prove `gray([23])`. Using the above rule, we cannot. The subgoal `donkey([23])` succeeds, but the subgoal `neg young([23])` does not, because we never asserted `neg young([23])` or anything from which it is derivable. Thus the erroneous result does not occur.

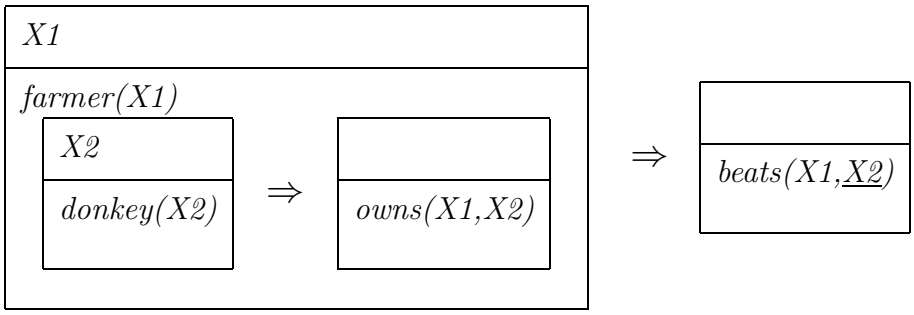
6.3 Modal subordination

The DRT account of pronominal anaphora in conditional sentences presupposes a simple discourse structure in which antecedent clauses are always superordinate to their consequent clauses. For example:

```
(1) If a farmer owns a donkey, he loves it.
(2) If a farmer owns every donkey, he beats it.
```

In (1), it can refer to a donkey, which is in a DRS superordinate to it. But in (2), it cannot refer to every donkey, because the donkey is introduced in a subordinate DRS:

`If a farmer owns every donkey, he beats it.`



Hence the underlined occurrence of X2 is illegal and the anaphoric reference is not permitted.

However, consider the following example (Roberts 1985, 1987):

(3) a. `If a farmer owns a donkey, he feeds it lots of hay.`
    b. `It soon grows fat on this diet.`

Fragment (3) cannot possibly be called ungrammatical or even dubiously grammatical. Clearly, sentence (3b) needs to go into the consequent of the if-then structure established by (3a). Neither the truth-conditions nor the anaphoric reference is handled correctly unless this is done. But the rules for DRT, as formulated so far, do not provide for this. Roberts (1985, 1987) refers to this phenomenon as "modal subordination" — semantically, the mood (modus) of the second sentence makes it behave like a subordinate clause in the first sentence, even though syntactically it is an independent clause. Goodman (1988) independently noticed the phenomenon and called it "multi-sentence consequents."

To handle modal subordination correctly, we must:

(1) formulate rules for recognizing modally subordinate sentences, based on appropriate choices of verb moods and tenses, presence of anaphors, and other indicators;

(2) build these rules into the DRS construction algorithm.

Implicit in this refinement of DRT is the recognition that discourse is more complex than early versions of DRT admitted. Instead of one single topmost DRS into which all independent clauses are inserted, an adequate theory of discourse will need to provide for a complex hierarchical structure including such things as subplots within a main plot, different mainplots at the same level, dialogues, and the like.

Present DRT is indeed discourse-oriented (as opposed to sentence- or proposition-oriented) when dealing with simple declarative sentences, but as soon as if-thens, negations or queries are involved, the DRS construction rules crucially rely on syntactic sentence boundaries (sentence final punctuation) as a trigger for DRS embedding. Non- syntactic intersentential links, for example modal subordination, are ignored.

6.4 Loop removal

In the present implementation, a sentence such as

```
Every gray donkey is an old donkey.
```

goes into extended Prolog as

```
donkey(X), old(X) ::- donkey(X), gray(X).
```

and is asserted as:

```
donkey(X) :- donkey(X), gray(X).
old(X) :- donkey(X), gray(X).
```

The first of these clauses sends Prolog into endless recursion. A simple syntactic readjustment rule needs to be added to remove loops of this type.

References

[1] Andrews, P. B. (1986) *An introduction to mathematical logic and type theory: to truth through proof.* Orlando: Academic Press.

[2] Covington, M. A. (1987) *GULP 1.1: An extension of Prolog for unification- based grammar.* ACMC Research Report 01–0021, University of Georgia.

[3] Covington, M. A.; Nute, D.; and Vellino, A. (1988) *Prolog programming in depth.* Glenview, Ill.: Scott, Foresman.

[4] Covington, M. A., and Schmitz, N. (1988) *An implementation of discourse representation theory.* ACMC Research Report 01–0023, University of Georgia.

[5] Gabbay, D. M., and Reyle, U. (1984) N-PROLOG: an extension of Prolog with hypothetical implications, I. *Journal of Logic Programming* 4:319–355.

[6] Guenthner, F., et al. (1986) A theory for the representation of knowledge. *IBM Journal of Research and Development* 30:1.39–56.

[7] Johnson, M., and Klein, E. (1986) *Discourse, anaphora, and parsing.* CSLI Research Report 86–63, Stanford University.

[8] Kamp, H. (1981) A theory of truth and semantic representation. In J. Groenendijk et al. (eds.) *Formal methods in the study of language,* 277–322. University of Amsterdam. Reprinted in J. Groenendijk et al. (eds.), *Truth, interpretation, and information* (Groningen- Amsterdam Studies in Semantics, 2), 1–40. Dordrecht: Foris.

[9] Kleene, S. C. (1967) *Mathematical logic.* New York: Wiley.

[10] Kolb, H.-P. (1985) *Aspekte der Implementation der Diskursrepräsentationstheorie.* FNS-Script 85–1, University of Tübingen.

[11] — (1987) Diskursrepräsentationstheorie und Deduktion, *Linguistische Berichte* 110:247–282.

[12] Nute, D., and Lewis, M. (1986) *A user's manual for d-Prolog.* ACMC Research Report 01–0017, University of Georgia.

[13] Roberts, C. (1985) *Modal subordination and pronominal anaphora in discourse.* Manuscript, University of Massachusetts at Amherst.

[14] — (1987). *Modal subordination, anaphora and distributivity.* Ph.D. Thesis, University of Massachusetts at Amherst.

[15] Skolem, T. (1928) Über die mathematische Logik. em Norsk matematisk tidskrift 10:125–142. Cited by Kleene (1967).

[16] Spencer-Smith, R. (1987) Semantics and discourse representation.*Mind and Language* 2.1:1–26.