Unification-Based Semantic Interpretation

Robert C. Moore
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025

Abstract

We show how unification can be used to specify the semantic interpretation of natural-language expressions, including problematical constructions involving long-distance dependencies. We also sketch a theoretical foundation for unification-based semantic interpretation, and compare the unification-based approach with more conventional techniques based on the lambda calculus.

1 Introduction

Over the past several years, unification-based formalisms (Shieber, 1986) have come to be widely used for specifying the syntax of natural languages, particularly among computational linguists. It is less widely realized by computational linguists that unification can also be a powerful tool for specifying the semantic interpretation of natural languages. While many of the techniques described in this paper are fairly well known among natural-language researchers working with logic grammars, they have not been extensively discussed in the literature, perhaps the only systematic presentation being that of Pereira and Shieber (1987). This paper goes into many issues in greater detail than do Pereira and Shieber, however, and sketches what may be the first theoretical analysis of unification-based semantic interpretation.

We begin by reviewing the basic ideas behind unification-based grammar formalisms, which will also serve to introduce the style of notation to be used throughout the paper. The notation is that used in the Core Language Engine (CLE) developed by SRI's Cambridge Computer Science Research Center in Cambridge, England, a system whose semantic-interpretation component makes use of many of the ideas presented here.

Fundamentally, unification grammar is a generalization of context-free phrase structure grammar in which grammatical-category expressions are not simply atomic symbols, but have sets of features with constraints on their values. Such constraints are commonly specified using sets of equations. Our notation uses equations of a very simple format—just `feature=value`—and permits only one equation per feature per constituent, but we can indicate constraints that would be expressed in other formalisms using more complex equations by letting the value of a feature contain a variable that appears in more than one equation. The CLE is written in Prolog, to take advantage of the efficiency of Prolog unification in implementing category unification, so our grammar rules are written as Prolog assertions, and we follow Prolog conventions in that constants, such as category and feature names, start with lowercase letters, and variables start with uppercase letters. As an example, a simplified version of the rule for the basic subject-predicate sentence form might be written in our notation as

```
(1) syn(s_np_vp,
        [s:[type=tensed],
         np:[person=P,num=N],
         vp:[type=tensed,
             person=P,num=N]]).
```

The predicate `syn` indicates that this is a syntax rule, and the first argument `s_np_vp` is a rule identifier that lets us key the semantic-interpretation rules to the syntax rules. The second argument of `syn` is a list of category expressions that make up the content of the rule, the first specifying the category of the mother constituent and the rest specifying the categories of the daughter constituents. This rule, then, says that a tensed sentence (`s:[type=tensed]`) can consist of a noun phrase (`np`) followed by a verb phrase (`vp`), with the restrictions that the verb phrase must be tensed (`type=tensed`), and that the noun phrase and verb phrase must agree in person and number—that is, the `person` and `num` features of the noun phrase must have the same respective values as the `person` and `num` features of the verb phrase.

These constraints are checked in the process of parsing a sentence by unifying the values of features specified in the rule with the values of features in the constituents found in the input. Suppose, for instance, that we are parsing the sentence

Mary runs using a left-corner parser. If *Mary* is parsed as a constituent of category

```
np:[person=3rd,num=sing],
```

then unifying this category expression with

```
np:[person=P,num=N]
```

in applying the sentence rule above will force the variables `P` and `N` to take on the values `3rd` and `sing`, respectively. Thus when we try to parse the verb phrase, we know that it must be of the category

```
vp:[type=tensed,person=3rd,num=sing].
```

Our notation for semantic-interpretation rules is a slight generalization of the notation for syntax rules. The only change is that in each position where a syntax rule would have a category expression, a semantic rule has a pair consisting of a "logical-form" expression and a category expression, where the logical-form expression specifies the semantic interpretation of the corresponding constituent. A semantic-interpretation rule corresponding to syntax rule (1) might look like the following:

```
(2) sem(s_np_vp,
        [(apply(Vp,Np),s:□),
         (Np,np:□),
         (Vp,vp:□)]).
```

The predicate `sem` means that this is a semantic-interpretation rule, and the rule identifier `s_np_vp` indicates that this rule applies to structures built by the syntax rule with the same identifier. The list of pairs of logical-form expressions and category expressions specifies the logical form of the mother constituent in terms of the logical forms and feature values of the daughter constituents. In this case the rule says that the logical form of a sentence generated by the `s_np_vp` rule is an applicative expression with the logical form of the verb phrase as the functor and the logical form of the noun phrase as the argument. (The dummy functor `apply` is introduced because Prolog syntax does not allow variables in functor position.) Note that there are no feature restrictions on any of the category expressions occurring in the rule. They are unnecessary in this case because the semantic rule applies only to structures built by the `s_np_vp` syntax rule, and thus inherits all the restrictions applied by that rule.

2 Functional Application vs. Unification

Example (2) is typical of the kind of semantic rules used in the standard approach to semantic interpretation in the tradition established by Richard Montague (1974) (Dowty, Wall, and Peters, 1981). In this approach, the interpretation of a complex constituent is the result of the functional application of the interpretation of one of the daughter constituents to the interpretation of the others.

A problem with this approach is that if, in a rule like (2), the verb phrase itself is semantically complex, as it usually is, a lambda expression has to be used to express the verb-phrase interpretation, and then a lambda reduction must be applied to express the sentence interpretation in its simplest form (Dowty, Wall, and Peters, 1981, pp. 98–111). To use (2) to specify the interpretation of the sentence *John likes Mary*, the logical form for *John* could simply be `john`, but the logical form for *likes Mary* would have to be something like `X\like(X,mary)`. [The notation `Var\Body` for lambda expressions is borrowed from Lambda Prolog (Miller and Nadathur, 1986).] The logical form for the whole sentence would then be `apply(X\like(X,mary),john)`, which must be reduced to yield the simplified logical form `like(john,mary)`.

Moreover, lambda expressions and the ensuing reductions would have to be introduced at many intermediate stages if we wanted to produce simplified logical forms for the interpretations of complex constituents such as verb phrases. If we want to accommodate modal auxiliaries, as in *John might like Mary*, we have to make sure that the verb phrase *might like Mary* receives the same type of interpretation as *like(s) Mary* in order to combine properly with the interpretation of the subject. If we try to maintain functional application as the only method of semantic composition, then it seems that the simplest logical form we can come up with for *might like Mary* is produced by the following rule:

```
(3) sem(vp_aux_vp,
        [(X\apply(Aux,apply(Vp,X)),
          vp:□),
         (Aux,aux:□),
         (Vp,vp:□)]).
```

Applying this rule to the simplest plausible logical forms for *might* and *like Mary* would produce the following logical form for *might like Mary*:

```
    X\apply(might,
            (apply(Y\like(Y,mary),X)))
```

which must be reduced to obtain the simpler expression `X\might(like(X,mary))`. When this expression is used in the sentence-level rule, another reduction is required to eliminate the remaining lambda expression. The part of the reduction step that gets rid of the `apply` functors is to some extent an artifact of the way we have chosen to encode these expressions as Prolog terms, but the lambda reductions are not. They are inherent in the approach, and normally each rule will introduce at least one lambda expression that needs to be reduced away.

It is, of course, possible to add a lambda-reduction step to the interpreter for the semantic rules, but it is both simpler and more efficient to use the feature system and unification to do explicitly what lambda expressions and lambda reduction do implicitly—assign a value to a variable embedded in a logical-form expression. According to this approach, instead of the logical form for a verb phrase being a logical predicate, it is the same as the logical form of an entire sentence, but with a variable as the subject argument of the verb and a feature on the verb phrase having that same variable as its value. The sentence interpretation rule can thus be expressed as

```
(4) sem(s_np_vp,
        [(Vp,s:[]),
         (Np,np:[]),
         (Vp,vp:[subjval=Np])]),
```

which says that the logical form of the sentence is just the logical form of the verb phrase with the subject argument of the verb phrase unified with the logical form of the subject noun phrase. If the verb phrase *likes Mary* is assigned the logical-form/category-expression pair

```
    (like(X,mary),vp:[subjval=X]),
```

then the application of this rule will unify the logical form of the subject noun phrase, say `john`, directly with the variable `X` in `like(X,mary)` to immediately produce a sentence constituent with the logical form `like(john,mary)`.

Modal auxiliaries can be handled equally easily by a rule such as

```
(5) sem(vp_aux_vp,
        [(Aux,vp:[subjval=S]),
         (Aux,aux:[argval=Vp]),
         (Vp,vp:[subjval=S])]).
```

If *might* is assigned the logical-form/category-expression pair

```
    (might(A),aux:[argval=A]),
```

then applying this rule to interpret the verb phrase *might like Mary* will unify `A` in `might(A)` with `like(X,mary)` to produce a constituent with the logical-form/category-expression pair

```
    (might(like,X,mary),vp:[subjval=X]),
```

which functions in the sentence-interpretation rule in exactly the same way as the logical-form/category-expression pair for *like Mary*.

3 Are Lambda Expressions Ever Necessary?

The approach presented above for eliminating the explicit use of lambda expressions and lambda reductions is quite general, but it does not replace all possible uses of lambda expressions in semantic interpretation. Consider the sentence *John and Bill like Mary*. The simplest logical form for the distributive reading of this sentence would be

```
    and(like(john,mary),like(bill,mary)).
```

If the verb phrase is assigned the logical-form/category-expression pair

```
    (like(X,mary),vp:[subjval=X]),
```

as we have suggested, then we have a problem: Only one of `john` or `bill` can be directly unified with `X`, but to produce the desired logical form, we seem to need two instances of `like(X,mary)`, with two different instantiations of `X`.

Another problem arises when a constituent that normally functions as a predicate is used as an argument instead. Common nouns, for example, are normally used to make direct predications, so a noun like *senator* might be assigned the logical-form/category-expression pair

```
    (senator(X),nbar:[argval=X])
```

according to the pattern we have been following. (Note that we do not have "noun" as a syntactic category; rather, a common noun is simply treated as a lexical "n-bar.") It is widely recognized, however, that there are "intensional" adjectives and adjective phrases, such as *former*, that need to be treated as higher-level predicates or operators on predicates, so that in an expression like *former* *senator*, the noun *senator* is not involved in directly making a predication, but instead functions as an argument to *former*. We can see that this must be the case, from the observation that a former senator is no longer a senator. The logical form we have assigned to *senator*, however, is not literally that of a predicate, however, but rather of a complete formula with a free variable. We therefore need some means to transform this formula with its free variable into an explicit predicate to be an argument of *former*. The introduction of lambda expressions provides the solution to this problem, because the transformation we require is exactly what is accomplished by lambda abstraction. The following rule shows how this can be carried out in practice:

```
(6) sem(nbar_adj_nbar,
        [(Adjp,nbar:[argval=A]),
         (Adjp,adjp:[type=intensional,
                     argval1=X\Nbar,
                     argval2=A]),
         (Nbar,nbar:[argval=X])]).
```

This rule requires the logical-form/category-expression pair assigned to an intensional adjective phrase to be something like

```
(former(P,Y),
 adjp:[type=intensional,
       argval1=P,argval2=Y]),
```

where `former(P,Y)` means that `Y` is a former `P`. The daughter `nbar` is required to be as previously supposed. The rule creates a lambda expression, by unifying the bound variable with the argument of the daughter `nbar` and making the logical form of the daughter `nbar` the body of the lambda expression, and unifies the lambda expression with the first argument of the `adjp`. The second argument of the `adjp` becomes the argument of the mother `nbar`. Applying this rule to *former senator* will thus produce a constituent with the logical-form/category-expression pair

```
(former(X\senator(X),Y),
 nbar:[argval=Y]).
```

This solution to the second problem also solves the first problem. Even in the standard lambda-calculus-based approach, the only way in which multiple instances of a predicate expression applied to different arguments can arise from a single source is for the predicate expression to appear as an argument to some other expression that contains multiple instances of that argument. Since our approach requires turning a predicate into an explicit lambda expression if it is used as an argument, by the time we need multiple instances of the predicate, it is already in the form of a lambda expression. We can show how this works by encoding a Montagovian (Dowty, Wall, Peters, 1981) treatment of conjoined subject noun phrases within our approach. The major feature of this treatment is that noun phrases act as higher-order predicates of verb phrases, rather than the other way around as in the simpler rules presented in Sections 1 and 2. In the Montagovian treatment, a proper noun such as *John* is given an interpretation equivalent to `P\P(john)`, so that when we apply it to a predicate like `run` in interpreting *John runs* we get something like `apply(P\P(john),run)` which reduces to `run(john)`. With this in mind, consider the following two rules for the interpretation of sentences with conjoined subjects:

```
(7) sem(np_np_conj_np
        [(Conj,np:[argval=P]),
         (Np1,np:[argval=P]),
         (Conj,conj:[argval1=Np1,
                     argval2=Np2]),
         (Np2,np:[argval=P])]).

(8) sem(s_np_vp,
        [(Np,s:[]),
         (Np,np:[argval=X\Vp]),
         (Vp,vp:[subjval=X])]).
```

The first of these rules gives a Montagovian treatment of conjoined noun phrases, and the second gives a Montagovian treatment of simple declarative sentences. Both of these rules assume that a proper noun such as *John* would have a logical-form/category-expression pair like

```
(apply(P,john),np:[argval=P]).
```

In (7) it is assumed that the conjunction *and* would have a logical-form/category-expression pair like

```
(and(P1,P2),
 conj:[argval1=P1,argval2=P2]).
```

In (7) the logical forms of the two conjoined daughter `np`s are unified with the two arguments of the conjunction, and the arguments of the daughter `np`s are unified with each other and with the single argument of the mother `np`. Thus applying (7) to interpret *John and Bill* yields a constituent with the logical-form/category-expression pair

```
    (and(apply(P,john),apply(P,bill)),
     np:[argval=P]).
```

In (8) an explicit lambda expression is constructed out of the logical form of the **vp** daughter in the same way a lambda expression was constructed in (6), and this lambda expression is unified with the argument of the subject **np**. For the sentence *John and Bill like Mary*, this would produce the logical form

```
    and(apply(X\like(X,mary),john),
        apply(X\like(X,mary),bill)),
```

which can be reduced to

```
    and(like(john,mary),like(bill,mary)).
```

4 Theoretical Foundations of Unification-Based Semantics

The examples presented above ought to be convincing that a unification-based formalism can be a powerful tool for specifying the interpretation of natural-language expressions. What may not be clear is whether there is any reasonable theoretical foundation for this approach, or whether it is just so much unprincipled "feature hacking." The informal explanations we have provided of how particular rules work, stated in terms of unifying the logical form for constituent X with the appropriate variable in the logical form for constituent Y, may suggest that the latter is the case. If no constraints are placed on how such a formalism is used, it is certainly possible to apply it in ways that have no basis in any well-founded semantic theory. Nevertheless, it is possible to place restrictions on the formalism to ensure that the rules we write have a sound theoretical basis, while still permitting the sorts of rules that seem to be needed to specify the semantic interpretation of natural languages.

The main question that arises in this regard is whether the semantic rules specify the interpretation of a natural-language expression in a compositional fashion. That is, does every rule assign to a mother constituent a well-defined interpretation that depends solely on the interpretations of the daughter constituents? If the interpretation of a constituent is taken to be just the interpretation of its logical-form expression, the answer is clearly "no." In our formalism the logical-form expression assigned to a mother constituent depends on both the logical-form expressions and the category expressions assigned to its daughters. As long as both category expressions and logical-form expressions have a theoretically sound basis, however, there is no reason that both should not be taken into account in a semantic theory; so, we will define the interpretation of a constituent based on both its category and its logical form. Taking the notion of interpretation in this way, we will explain how our approach can be made to preserve compositionality. First, we will show how to give a well-defined interpretation to every constituent; then, we will sketch the sort of restrictions on the formalism one needs to guarantee that any interpretation-preserving substitution for a daughter constituent also preserves the interpretation of the mother constituent.

The main problem in giving a well-defined interpretation to every constituent is how to interpret a constituent whose logical-form expression contains free variables that also appear in feature values in the constituent's category expression. Recall the rule we gave for combining auxiliaries with verb phrases:

```
(5) sem(vp_aux_vp,
        [(Aux,vp:[subjval=S]),
         (Aux,aux:[argval=Vp]),
         (Vp,vp:[subjval=S])]).
```

This rule accepts daughter constituents having logical-form/category-expression pairs such as

```
    (might(A),aux:[argval=A])
```

and

```
    (like(X,mary),vp:[subjval=X])
```

to produce a mother constituent having the logical-form/category-expression pair

```
    (might(like,X,mary),vp:[subjval=X].
```

Each of these pairs has a logical-form expression containing a free variable that also occurs as a feature value in its category expression. The simplest way to deal with logical-form/category-expression pairs such as these is to regard them in the way that syntactic-category expressions in unification grammar can be regarded—as abbreviations for the set of all their well-formed fully instantiated substitution instances.

To establish some terminology, we will say that a logical-form/category-expression pair containing no free-variable occurrences has a "basic interpretation," which is simply the ordered pair consisting of the interpretation of the logical-form expression and the interpretation of the category expression. Since there are no free variables involved, basic interpretations should be unproblematic. The logical-form expression will simply be a closed well-formed expression of some ordinary logical language, and its interpretation will be whatever the usual interpretation of that expression is in the relevant logic. The category expression can be taken to denote a fully instantiated grammatical category of the sort typically found in unification grammars. The only unusual property of this category is that some of its features may have logical-form interpretations as values, but, as these will always be interpretations of expressions containing no free-variable occurrences, they will always be well defined.

Next, we define the interpretation of an arbitrary logical-form/category-expression pair to be the set of basic interpretations of all its well-formed substitution instances that contain no free-variable occurrences. For example, the interpretation of a constituent with the logical-form/category-expression pair

```
(might(like,X,mary),vp:[subjval=X])
```

would consist of a set containing the basic interpretations of such pairs as

```
(might(like,john,mary),
 vp:[subjval=john]),

(might(like,bill,mary),
 vp:[subjval=bill]),
```

and so forth.

This provides well-defined interpretation for every constituent, so we can now consider what restrictions we can place on the formalism to guarantee that any interpretation-preserving substitution for a daughter constituent also preserves the interpretation of its mother constituent. The first restriction we need rules out constituents that would have degenerate interpretations: No semantic rule or semantic lexical specification may contain both free and bound occurrences of the same variable in a logical-form/category-expression pair.

To see why this restriction is needed, consider the logical-form/category-expression pair

```
(every(X,man(X),die(X)),
 np:[boundvar=X,bodyval=die(X)]),
```

which might be the substitution instance of a daughter constituent that would be selected in a rule that combines noun phrases with verb phrases. The problem with such a pair is that it does not have any well-formed substitution instances that contain no free-variable occurrences. The variable `X` must be left uninstantiated in order for the logical-form expression `every(X,man(X),die(X))` to be well formed, but this requires a free occurrence of `X` in `np:[boundvar=X,bodyval=die(X)]`. Thus this pair will be assigned the empty set as its interpretation. Since any logical-form/category-expression pair that contains both free and bound occurrences of the same variable will receive this degenerate interpretation, any other such pair could be substituted for this one without altering the interpretations of the daughter constituent substitution instances that determine the interpretation of the mother constituent. It is clear that this would normally lead to gross violations of compositionality, since the daughter substitution instances selected for the noun phrases *every man*, *no woman*, and *some dog* would all receive the same degenerate interpretation under this scheme.

This restriction may appear to be so constraining as to rule out certain potentially useful ways of writing semantic rules, but in fact it is generally possible to rewrite such rules in ways that do not violate the restiction. For example, in place of the sort of logical-form/category-expression pair we have just ruled out, we can fairly easily rewrite the relevant rules to select daughter substitution instances such as

```
(every(X,man(X),die(X)),
 np:[bodypred=X\die(X)]),
```

which does not violate the constraint and has a completely straightforward interpretation.

Having ruled out constituents with degenerate interpretations, the principal remaining problem is how to exclude rules that depend on properties of logical-form expressions over and above their interpretations. For example, suppose that the order of conjuncts does not affect the interpretation of a logical conjunction, according to the interpretation of the logical-form language. That is, `and(p,q)` would have the same interpretation as `and(q,p)`. The potential problem that this raises is that we might write a semantic rule that contains both a logical-form expression like `and(P,Q)` in the specification of a daughter constituent and the variable `P` in the logical form of the mother constituent. This would be a violation of compositionality, because the interpretation of the mother would depend on the interpretation of the left conjunct of a conjunction, even though, according to the semantics of the logical-form language, it makes no sense to distinguish the left and right conjuncts. If order of conjunction does not affect meaning, we ought to be able to substitute a daughter with the logical form `and(q,p)` for one with the logical form `and(p,q)` without affecting the interpretation assigned to the mother, but clearly, in this case, the interpretation of the mother would be affected.

It is not clear that there is any uniquely optimal set of restrictions that guarantees that such violations of compositionality cannot occur. Indeed, since unification formalisms in general have Turing machine power, it is quite likely that there is no computable characterization of all and only the sets of semantic rules that are compositional. Nevertheless, one can describe sets of restrictions that do guarantee compositionality, and which seem to provide enough power to express the sorts of semantic rules we need to use to specify the semantics of natural languages. One fairly natural way of restricting the formalism to guarantee compositionality is to set things up so that unifications involving logical-form expressions are generally made against variables, so that it is possible neither to extract subparts of logical-form expressions nor to filter on the syntactic form of logical-form expressions. The only exception to this restriction that seems to be required in practice is to allow for rules that assemble and disassemble lambda expressions with respect to their bodies and bound variables. So long as no extraction from inside the body of a lambda expression is allowed, however, compositionality is preserved.

It is possible to define a set of restrictions on the form of semantic rules that guarantee that no rule extracts subparts (other than the body or bound variable of a lambda expression) of a logical-form expression or filters on the syntactic form of a logical-form expression. The statement of these restrictions is straightforward, but rather long and tedious, so we omit the details here. We will simply note that none of the sample rules presented in this paper involve any such extraction or filtering.

5 The Semantics of Long-Distance Dependencies

The main difficulty that arises in formulating semantic-interpretation rules is that constituents frequently appear syntactically in places that do not directly reflect their semantic role. Semantically, the subject of a sentence is one of the arguments of the verb, so it would be much easier to produce logical forms for sentences if the subject were part of the verb phrase. The use of features such as `subjval`, in effect, provides a mechanism for taking the interpretation of the subject from the place where it occurs and inserting it into the verb phrase interpretation where it "logically" belongs.

The way features can be manipulated to accomplish this is particularly striking in the case of the long-distance dependencies, such as those in WH-questions. For the sentence *Which girl might John like?*, the simplest plausible logical form would be something like

```
which(X,girl(X),might(like(john,X))),
```

where the question-forming operator `which` is treated as a generalized quantifier whose "arguments" consist of a bound variable, a restriction, and a body.

The problem is how to get the variable `X` to link the part of the logical form that comes from the fronted interrogative noun phrase with the argument of `like` that corresponds to the noun phrase gap at the end of the verb phrase. To solve this problem, we can use a technique called "gap-threading." This technique was introduced in unification grammar to describe the syntax of constructions with long-distance dependencies (Karttunnen, 1986) (Pereira and Sheiber, 1987, pp. 125–129), but it works equally well for specifying their semantics. The basic idea is to use a pair of features, **gapvalsin** and **gapvalsout**, to encode a list of semantic "gap fillers" to be used as the semantic interpretations of syntactic gaps, and to thread that list along to the points where the gaps occur. These gap fillers are often just the bound variables introduced by the constructions that permit gaps to occur.

The following semantic rules illustrate how this mechanism works:

```
(9) sem(whq_ynq_np_gap,
        [(Np,s:[gapvalsin=[],
                gapvalsout=[]]),
         (Np,np:[type=interrog,
                 bodypred=A\Ynq]),
         (Ynq,s:[gapvalsin=[A],
                 gapvalsout=[]])]).
```

This is the semantic-interpretation rule for a WH-question with a long-distance dependency. The syntactic form of such a sentence is an interrogative noun phrase followed by a yes/no question with a noun phrase gap. This rule expects the interrogative noun phrase *which girl* to have a logical-form/category-expression pair such as

```
(which(X,girl(X),Bodyval),
 np:[type=interrog,
     bodypred=X\Bodyval]).
```

The feature bodypred holds a lambda expression whose body and bound variable are unified respectively with the body and the bound variable of the which expression. In (9) the body of this lambda expression is unified with the logical form of the embedded yes/no question, and the **gapvalsin** feature is set to be a list containing the bound variable of the lambda expression. This list is actually used as a stack, to accomodate multiply nested filler-gap dependencies. Since this form of question cannot be embedded in other constructions, however, we know that in this case there will be no other gap-fillers already on the list.

This is the rule that provides the logical form for empty noun phrases:

```
(10) sem(empty_np,
        [(Val,np:[gapvalsin=[Val|ValRest],
                  gapvalsout=ValRest])]).
```

Notice that it has a mother category, but no daughter categories. The rule simply says that the logical form of an empty np is the first element on its list of semantic gap-fillers, and that this element is "popped" from the gap-filler list. That is, the **gapvalsout** feature takes as its value the tail of the value of the **gapvalsin** feature.

We now show two rules that illustrate how a list of gap-fillers is passed along to the points where the gaps they fill occur.

```
(11) sem(vp_aux_vp,
        [(Aux,vp:[subjval=S,gapvalsin=In,
                  gapvalsout=Out]),
         (Aux,aux:[argval=Vp]),
         (Vp,vp:[subjval=S,gapvalsin=In,
                 gapvalsout=Out])]).
```

This semantic rule for verb phrases formed by an auxilliary followed by a verb phrase illustrates the typical use of the gap features to "thread" the list of gap fillers through the syntactic structure of the sentence to the points where they are needed. An auxillary verb cannot be or contain a WH-type gap, so there are no gap features on the category **aux**. Thus the gap features on the mother **vp** are simply unified with the corresponding features on the daughter **vp**.

A more complex case is illustrated by the following rule:

```
(12) sem(vp_vp_pp,
        [(Pp,vp:[subjval=S,gapvalsin=In,
                 gapvalsout=Out]),
         (Vp,vp:[subjval=S,gapvalsin=In,
                 gapvalsout=Thru]),
         (Pp,pp:[argval=Vp,gapvalsin=Thru,
                 gapvalsout=Out])]).
```

This is a semantic rule for verb phrases that consist of a verb phrase and a prepositional phrase. Since WH-gaps can occur in either verb phrases or prepositional phrases, the rule threads the list carried by the **gapvalsin** feature of the mother **vp** first through the daughter **vp** and then through the daughter **pp**. This is done by unifying the mother **vp**'s **gapvalsin** feature with the daughter **vp**'s **gapvalsin** feature, the daughter **vp**'s **gapvalsout** feature with the daughter **pp**'s **gapvalsin** feature, and finally the daughter **pp**'s **gapvalsout** feature with the mother **vp**'s **gapvalsout** feature. Since a gap-filler is removed from the list once it has been "consumed" by a gap, this way of threading ensures that fillers and gaps will be matched in a last-in-first-out fashion, which seems to be the general pattern for English sentences with multiple filler-gap dependencies. (This does not handle "parasitic gap" constructions, but these are very rare and at present there seems to be no really convincing linguistic account of when such constructions can be used.)

Taken altogether, these rules push the quantified variable of the interrogative noun phrase onto the list of gap values encoded in the feature **gapvalsin** on the embedded yes/no question. The list of gap values gets passed along by the gap-threading mechanism, until the empty-noun-phrase rule pops the variable off the gap values list and uses it as the logical form of the noun phrase gap. Then the entire logical form for the embedded yes/no question is unified with the body of the logical form for the interrogative noun phrase, producing the desired logical form for the whole sentence.

This treatment of the semantics of long-distance dependencies provides us with an answer to the question of the relative expressive power of our approach compared with the conventional lambda-calculus-based approach. We know that the unification-based approach is at least as powerful as the conventional approach, because the the conventional approach can be embedded directly in it, as illustrated by the examples in Section 3. What about the other way around? Many unification-based rules have direct lambda-calculus-based counterparts; for example (2) is a counterpart of (4), and (3) is the counterpart of (5). Once we introduce gap-threading, however, the correspondence breaks down. In the conventional approach, each rule applies only to constituents whose semantic interpretation is of some particular single semantic type, say, functions from individuals to propositions. If every free variable in our approach is treated as a lambda variable in the conventional approach, then no one rule can cover two expressions whose interpretation essentially involves different numbers of variables, since these would be of different semantic types. Hence, rules like (11) and (12), which cover constituents containing any number of gaps, would have to be replaced in the conventional approach by a separate rule for each possible number of gaps. Thus, our formalism enables us to write more general rules than is possible taking the conventional approach.

6 Conclusions

In this paper we have tried to show that a unification-based approach can provide powerful tools for specifying the semantic interpretation of natural-language expressions, while being just as well founded theoretically as the conventional lambda-calculus-based approach. Although the unification-based approach does not provide a substitute for all uses of lambda expressions in semantic interpretation, we have shown that lambda expressions can be introduced very easily where they are needed. Finally, the unification-based approach provides for a simpler statement of many semantic-interpretation rules, it eliminates many of the lambda reductions needed to express semantic interpretations in their simplest form, and in some cases it allows more general rules than can be stated taking the conventional approach.

Acknowledgments

The research reported in this paper was begun at SRI International's Cambridge Computer Science Research Centre in Cambridge, England, supported by a grant from the Alvey Directorate of the U.K. Department of Trade and Industry and by the members of the NATTIE consortium (British Aerospace, British Telecom, Hewlett Packard, ICL, Olivetti, Philips, Shell Research, and SRI). The work was continued at the SRI Artificial Intelligence Center and the Center for the Study of Language and Information, supported in part by a gift from the Systems Development Foundation and in part by a contract with the Nippon Telegraph and Telephone Corporation.

References

Dowty, David R., Robert Wall, and Stanley Peters (1981) *Introduction to Montague Semantics* (D. Reidel, Dordrecht, Holland).

Karttunnen, Lauri (1986) "D-PATR: A Development Environment for Unification-Based Grammars," Proceedings of the 11th International Conference on Computational Linguistics, Bonn, West Germany, pp. 74–80.

Miller, Dale A., and Gopalan Nadathur (1986) "Higher-Order Logic Programming," in E. Shapiro (ed.), *Third International Conference on Logic Programming*, pp. 448–462 (Springer-Verlag, Berlin, West Germany).

Montague, Richard (1974) *Formal Philosophy* (Yale University Press, New Haven, Connecticut).

Pereira, Fernando C.N., and Stuart M. Shieber (1987) *Prolog and Natural-Language Analysis*, CSLI Lecture Notes Number 10, Center for the Study of Language and Information, Stanford University, Stanford, California.

Shieber, Stuart M. (1986) *An Introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes Number 4, Center for the Study of Language and Information, Stanford University, Stanford, California.