An Algorithm for Generating Quantifier Scopings

Jerry R. Hobbs and Stuart M. Shieber
Artificial Intelligence Center
SRI International
and
Center for the Study of Language and Information
Stanford University

October 1, 1986

Abstract

The syntactic structure of a sentence often manifests quite clearly the predicate-argument structure and relations of grammatical subordination. But scope dependencies are not so transparent. As a result, many systems for representing the semantics of sentences have ignored scoping or generated scopings with mechanisms that have often been inexplicit as to the range of scopings they choose among or profligate in the scopings they allow.

In this paper, we present an algorithm, along with proofs of some of its important properties, that generates scoped semantic forms from unscoped expressions encoding predicate-argument structure. The algorithm is not profligate as are those based on permutation of quantifiers, and it can provide a solid foundation for computational solutions where completeness is sacrificed for efficiency and heuristic efficacy.

1 Introduction

A principal focus of computational linguistics, as a branch of computer science, ought to be the design of algorithms. A large number of algorithms have undoubtedly been devised for dealing with problems every researcher has to face in constructing a natural language system, but that simply have not received wide circulation. These algorithms are part of the "folk culture", buried in the most technical, unreadable portions of theses, passed among colleagues informally at best, and often reinvented. It should be a practice to publish these algorithms in isolation, independent of a particular implementation or system.

A version of this paper appears in *Computational Linguistics*, volume 13, numbers 1–2, pages 47–63, January–June, 1987. This version incorporates minor typographical revisions as of February 17, 1994.

This paper constitutes an effort to initiate such a practice. A problem that many natural-language efforts have faced is the recovery of implicit semantic scope dependency possibilities—such as those manifest in quantifiers and modals—from predicate-argument relations and relations of grammatical subordination, which are more or less transparently conveyed by the syntactic structure of sentences. Previous computational efforts typically have not been based on an explicit notion of the range of possible scopings. In response to this problem, we present an algorithm that generates quantifier scopings for English sentences.

1.1 The Problem of Generating Quantifier Scopings

The naive algorithm for generating quantifier scopings is to generate all permutations of the quantifiers. For a sentence with n quantified noun phrases this will generate $n!$ different readings. But for the sentence

(1) Every representative of a company saw most samples.

there are not six different readings, but only five. The reading that is missing is the one in which "most samples" is outscoped by "every representative" but outscopes "a company". A model for the disallowed reading could include a different company not only for each representative but also for each sample.

The reduction in number of readings for a given sentence is not significant for sentence (1), but in the sentence

(2) Some representative of every department in most companies saw a few samples of each product.

there are only 42 valid readings, as opposed to the 120 readings the naive algorithm would generate, and this constitutes a significant difference indeed. The recent trend in computational linguistics has been to view more and more noun phrases, as well as other constituents, as introducing quantifiers, so that sentences with this much quantificational complexity are not at all unusual. (The immediately preceding sentence, for example, has six or seven quantifiers.)

This observation of "illegitimate readings" is not intended as a new or controversial claim about an idiosyncrasy of English. It accords well with semantic judgments about the possibility of such readings. For instance, we find it impossible to view sentence (1) as expressing that for each representative there was a group of most samples which he saw, and furthermore, for each sample he saw, there was a company he was a representative of.

We can find the same problem of illegitimate readings in the standard account of the "Cooper storage" mechanism for generating quantifier scopings (Cooper, 1983). Cooper's method generates an expression in intensional logic for the illegitimate readings, but the expression contains an unbound variable and a vacuous quantifier.[1]

[1] William Keller (1986) has also noted this problem with Cooper's method. His independent solution to the problem, stated in terms of "nested Cooper storage", resembles the one presented here.

Finally, the observation follows merely syntactically from the ill-formedness of certain logical form expressions. Let us examine why this is so. The propositional content of a sentence can be seen as combining specifications that restrict the range of quantified entities, together with assertions about the entities so specified. This intuition is often made formal in the use of logical languages which syntactically separate the notion of the range of a quantified expression from its scope by placing the information about the range in a part of the expression which we call the *restriction* and the assertions in a part called the *body*. (Henceforth, we will uniformly use the terms restriction and body.) The separation of these two semantic roles of range and scope into restriction and body as an important fact of the logical structure of English can be seen for example, in Woods's four-part quantifier structures (Woods 1977), in the recommendations of Moore (1981), and in the generalized quantifier research of Barwise and Cooper and others. The latter have demonstrated the necessity of such a separation for quantifiers other than the standard first-order ones (Barwise and Cooper, 1981; Cushing, 1976).

But under this understanding of English logical structure, it follows that no sixth reading exists for sentence (1) above. Consider the reading in which the universal outscopes the 'most' which outscopes the existential in the logical form for this sentence. Then, using the notation of Moore (1981) for four-part quantifier structures, the logical form must have the following structure:

```
all(r, representative(r) ... , ...)
```

since the universal is outermost. Now the existential is within the scope of the universal by hypothesis, and since it provides a restriction on the range of the variable r, it must occur in the restriction of the quantifier. Thus, we have:

```
all(r, representative(r) & some(c, company(c), of(c,r)),
       ...saw(r,s)...)
```

But where can the quantifier 'most' be put to bind the variable s corresponding to the samples seen? It must outscope its occurrence in the body of the universal, but it must also by hypothesis outscope the existential in the restriction of the universal. To outscope both, it must outscope the universal itself, but this violates the assumed scope relations. Thus, no such reading is possible. By a similar argument, it follows from the logical structure of English that *in general* a quantifier from elsewhere in a sentence cannot come after the quantifier associated with a head noun and before the quantifier associated with a noun phrase in the head noun's complement.

Most research in linguistic semantics, e.g., Montague (1973) and Cooper (1983), has concentrated on explicitly defining the range of possible scope relationships that can be manifested in sentences. But, to our knowledge, all fall prey to the profligacy of generation just outlined.

1.2 Other Issues in Quantifier Scoping

1.2.1 Other Spurious Scopings

We are concerned here only with suppressing readings that are spurious for purely structural reasons, that is, for reasons that follow from the general relationship between the structure of sentences and the structure of their logical forms and independent of the meanings of the particular sentences. For instance, we are not concerned with *logical* redundancies, such as those due to the commutativity of successive universal quantifiers. When we move beyond the two first-order logical quantifiers to deal with the so-called generalized quantifiers such as "most", these logical redundancies become quite rare. Similarly, we are not concerned with the infelicity of certain readings due to lexical semantic or world knowledge, such as the fact that "a child" cannot outscope "every man" in the sentence

I've met a child of every man in this room.

1.2.2 Heuristically Primary Scopings

Computational research on quantifier scoping has emphasized generating a single scoping, which can be thought of as heuristically primary, as discussed by, e.g., Woods (1977), Pereira (1983), Grosz, *et al.* (1985). We are concerned not with generating the *best* reading, but with generating *all* readings. The reader may object that it is inappropriate in a practical natural language system to generate scopings one by one for testing against semantic and pragmatic criteria. Instead, one should appeal to various heuristics to generate only the most likely reading, or at least to generate readings in order of their plausibility. These include the following: lexical heuristics, e.g., "each" usually outscopes "some"; syntactic heuristics, e.g., a noun phrase in a relative clause is usually outscoped by the head noun, and a noun phrase in a prepositional phrase complement of a relational head noun usually outscopes the head noun; and ordering heuristics, such as the principle that left-to-right order at the same syntactic level is generally preserved in the quantifier order.[2]

We are sympathetic with this view. Nevertheless, there are several reasons that codifying a complete algorithm remains useful. First, a complete and sound algorithm provides a benchmark against which other approaches can be tested. Second, one may actually wish to use a generate-and-test mechanism in simpler implementations, and it should be correct and as efficient as possible. It should not generate scopings that can be ruled out on purely structural grounds. Finally, the algorithm we present might be modified to incorporate heuristics to generate scopings in a certain order or only certain of the scopings. The soundness and correctness of the underlying algorithm provide a guarantee of soundness for a heuristically guided version. We include a few comments below about incorporating ordering heuristics into our scoping generation algorithm, although we should point out that the possibilities are somewhat limited due to the local nature of where the heuristics can be applied. A full discussion of heuristically-guided scoping generation is, of course, beyond the scope of this paper.

1.2.3 Scope of Opaque Predicates

In addition to handling the scoping of quantifiers relative to each other, the algorithm we present also allows quantifiers to be scoped within or outside of opaque arguments of higher-order predicates. For instance, the algorithm generates two readings for the sentence

Everyone isn't here.

[2] These heuristics should themselves be made available in a public forum.

corresponding to the two relative scopings of the universal quantifier and the negation.

2 The Algorithm

In the discussion below, we assume that parsing has made explicit the predicate-argument relations and the relations of grammatical subordination in the form of a logical encoding in an input language. A well-formed formula (wff) in the input language is a predicate or other operator applied to one or more arguments. An argument can be a constant or variable, another wff, or what we will call a *complex term*. A complex term is an ordered triple consisting of a quantifier, a variable, and a wff (called the *restriction*), which represents the predication that is grammatically subordinated to the variable. The input representation for sentence (2) is, then, the following (ignoring tense):

```
see(<some r and(rep(r),
                of(r,<every d and(dept(d),
                                  in(d,<most c co(c)>))>))>
    <a-few s and(samp(s),
                 of(s,<each p prod(p)>))>)
```

A complex term can be read "*quantifier variable* such that *restriction*", e.g., "most `c` such that `c` is a company".

The output language is identical to the input language, except that it does not contain complex terms. Quantifiers are expressed in the output language as operators which take three arguments: the variable bound by the quantifier, a wff restricting the range of the quantified variable, and the body scoped by the quantification, schematically

quantifier(variable, restriction, body)

This encoding of quantification is the same as that found in Woods (1977) and Moore (1981). We will refer to such expressions as quantified wffs. Thus, one reading for sentence (2) is represented by the following quantified wff:

```
some(r, and(rep(r),
            every(d, and(dept(d),
                         most(c, co(c),
                                 in(d, c))),
                     of(r, d))),
        a-few(s, and(samp(s),
                     each(p, prod(p),
                             of(s, p))),
                 see(r, s)))
```

Intermediate structures built during the course of scoping include both complex terms and quantified wffs. We use the term *full scoping* for an expression in the output language, i.e., one that has no complex terms.

We also will use the terms *bound* and *free* as follows: An expression *binds* a variable v if the expression is of the form $< q \ v \ r >$ or $q(v, r, s)$ where q is a quantifier. The variable v is said to be *bound in* the expressions r, or r and s, respectively. A variable v is *unbound* or *free* in an expression α if there is an occurrence of v in α which is not also an occurrence in a subexpression of α binding v. Note that here quantified wffs and complex terms are both thought of as expressions binding a variable.

2.1 Summary of the Algorithm

We present both nondeterministic and deterministic versions of the algorithm[3] in an ALGOL-like language. Both algorithms, however, have the same underlying structure, based on the primitive operation of "applying" a complex term to a wff in which it occurs: a complex term in a wff is replaced by the variable it restricts, and that variable is then bound by wrapping the entire form in the appropriate quantifier. Thus, applying the term $< q \ x \ r(x) >$ to a wff containing that complex term, say, $p(< q \ x \ r(x) >)$ yields the quantified wff $q(x, r(x), p(x))$. This is the primitive operation by which complex terms are removed from a wff and quantified wffs are introduced. It is implemented by the function *apply*.

The generation of a scoping from a wff proceeds in two stages. First, the opaque argument positions within the wff are scoped. The function *pull-opaque-args* performs this task by replacing wffs in opaque argument positions by a (full or partial) scoping of the original wff. For instance, if p were a predicate opaque in its only argument, then, for the wff $p(s(< q \ x \ r(x) >))$, *pull-opaque-args* would generate the wff $p(q(x, r(x), s(x)))$ or the unchanged wff $p(s(< q \ x \ r(x) >))$. In the former, the opaque predicate p outscopes the quantifier q. In the latter, the quantifier q has not been applied yet and the wff will subsequently yield readings in which q has wider scope than p.

Second, some or all of the remaining terms are applied to the entire wff. The function *apply-terms* iteratively (through a tail recursion) chooses a complex term in the wff and applies it. Thus *apply-terms* acting upon the wff $p(< q_1 \ x \ r_1(x) >, < q_2 \ y \ r_2(y) >)$ will yield one of the five wffs

$$
\begin{array}{l}
p(< q_1 \ x \ r_1(x) >, < q_2 \ y \ r_2(y) >) \\
q_1(x, r_1(x), p(x, < q_2, y, r_2(y) >)) \\
q_2(y, r_2(y), p(< q_1, x, r_1(x) >, y)) \\
q_2(y, r_2(y), q_1(x, r_1(x), p(x, y))) \\
q_1(x, r_1(x), q_2(y, r_2(y), p(x, y)))
\end{array}
$$

depending on how many quantifiers are applied and in what order. The choice of a complex term is restricted to a subset of the terms in the wff, the so-called *applicable* terms. The principal restriction on applicable terms is that they not be embedded in any other complex term in the wff. Section 4.1 discusses a further restriction. The function *applicable-term* returns an applicable term in a given wff.

These two stages are manifested in the function *pull* which generates all partial or full scopings of a wff by invoking *pull-opaque-args* and *apply-terms*. Since ultimately only full scopings are

[3] A nondeterministic version of the algorithm, formulated by both authors, was presented by Hobbs (1983).

desired, an additional argument to *pull* and *apply-terms* controls whether partial scopings are to be returned. When this flag, *complete?*, is *true*, *apply-terms*, and hence *pull* will return only expressions in which no more complex terms remain to be applied, for example, only the last two of the five readings above.

Finally, the restrictions of the complex terms may themselves contain complex terms and must be scoped themselves. The *apply* function therefore recursively generates the scopings for the restriction by calling *pull* on that restriction, and a quantified wff is generated for each possible partial or complete scoping of the restriction. Schematically, in the simplest case, for the expression $p(< q_1\ x\ r_1(x, < q_2\ y\ r_2(y) >) >)$ and its complex term $< q_1 \cdots >$,[4] *apply* generates the complete scoping

$$q_1(x, q_2(y, r_2(y), r_1(x, y)), p(x))$$

(having called *apply* recursively on $< q_2 \cdots >$), and the partial scoping

$$q_1(x, r_1(x, < q_2\ y\ r_2(y) >), p(x))$$

A subsequent application of the remaining complex term will yield the "wide scope" reading

$$q_2(y, r_2(y), q_1(x, r_1(x, y), p(x)))$$

The disallowed readings that are produced by the "all permutations" algorithm are never produced by this algorithm, because it is everywhere sensitive to the four-part quantifier structure of the target logical form.

The difference between the nondeterministic and deterministic versions lies only in their implementation of the choice of terms and returning of values. This is done either nondeterministically, or by iterating through and returning explicit sets of possibilities. A nondeterministic Prolog version and a deterministic COMMON LISP version of the algorithm are given in Appendices A and B. The full text of these versions (including auxiliary functions not listed here) is available from the authors. A variant of the COMMON LISP version is currently being used at SRI International to generate scopings in the KLAUS system.

2.2 Language Constructs

In the specifications below, the **let** construct implements local variable assignment. All assignments are done sequentially, not in parallel. The syntax is

let ⟨*assignments*⟩
in ⟨*body*⟩

The entire expression returns what the body returns. Destructuring by pattern matching is allowed in the assignments; for example,

[4]Note that this term is applicable according to the criterion discussed above, whereas the embedded term binding y is not. The fact that we still get both scopings even without the possibility of applying the embedded term first demonstrates that the restriction on applicable terms does not affect completeness of the algorithm.

let <*quant var restrict*> := *term*
in $\langle$*body*$\rangle$

simultaneously binds *quant*, *var*, and *restrict* to the three corresponding components in *term*. The symbol ":=" is used for assignment. **lambda** is an anonymous-function-forming operator. Its syntax is

lambda($\langle$*variable*$\rangle$).
$\langle$*body*$\rangle$

where $\langle$*variable*$\rangle$ is free in $\langle$*body*$\rangle$. We assume lexical scoping in lambda expressions. The statement "**return** *value*" returns a value from a function. The binary function *map* (similar to LISP's *mapcar*) applies its second argument (a lambda expression) to each of the elements of its first argument (a list). It returns a corresponding list of the values of the individual applications. The function *integers*(*lower*, *upper*) returns a list of the integers in the range *lower* to *upper*, inclusive and in order (corresponding to APL's *iota*). The function *length*(*list*) is obvious. The expression *list*[*n*] returns the nth element of the list *list*. The function *subst*(*x*,*y*,*expr*) substitutes x for all occurrences of y in *expr*.

The unary function *predicate*(*wff*) returns the main predicate in a wff. The unary function *arguments*(*wff*) returns a list of the arguments in a wff. Applied to two arguments, *wff* is a binary function that takes a predicate name and a list of arguments, and returns the wff consisting of the application of the predicate to the arguments. Applied to four arguments, *wff* is a quaternary function that takes a quantifier name, a variable name, a restriction, and a body, and returns the quantified wff consisting of the binding of the variable by the quantifier in the restriction and body. The binary predicate *opaque*(*predicate*, n) returns *true* if and only if the predicate is opaque in its nth argument. It is naturally assumed that opaque argument positions are filled by wff expressions, not terms. Each of the unary predicates *wff?*, *term?*, and *quantifier?* returns true if and only if its argument is a wff, a complex term, or a quantifier operator, respectively.

2.3 The Nondeterministic Algorithm

In the nondeterministic version of the algorithm, there are three special language constructs. The unary predicate *exists*(*expression*) evaluates its argument nondeterministically to a value and returns true if and only if there exist one or more values for the expression. The binary operator "$a||b$" nondeterministically returns one of its arguments (a or b). The function *term(form)* nondeterministically returns a complex term in *form*. Finally, the function *applicable-term(form)* nondeterministically returns a complex term in *form* that can be applied to *form*.

The nondeterministic version of the algorithm is as follows. The function *gen*(*form*) nondeterministically returns a valid full scoping of the formula *form*.

function *gen(form);*
return *pull(form,true).*

The function *pull*(*form*, *complete*?) nondeterministically returns a valid scoping of the formula *form*. If *complete*? is *true*, then only full scopings are returned; otherwise, partial scopings are allowed as well.

```
function pull(form, complete?);
    return apply-terms(pull-opaque-args(form), complete?).
```

The function *pull-opaque-args*(*form*), when applied to a wff returns a wff generated from *form* but with arguments in opaque argument positions replaced by a valid scoping of the original value. Since the recursive call to *pull* has *complete?* set to *false*, the unchanged argument is a nondeterministic possibility, even for opaque argument positions. When applied to any other type of expression (i.e., a term of some sort), *form* is unchanged.

```
function pull-opaque-args(form);
    if not(wff?(form))
      then return form
      else
          let predicate := predicate(form);
              args := arguments(form)
          in return
                wff( predicate,
                     map( integers(1,length(args)),
                          lambda(arg-index).
                              if opaque(predicate,arg-index)
                                  then pull(args[arg-index],false)
                                  else pull-opaque-args(args[arg-index]) ) ).
```

The function *apply-terms*(*form, complete?*) chooses several terms in *form* nondeterministically and applies them to *form*. If *complete?* is *true* then only full scopings are returned.

```
function apply-terms(form, complete?);
    if not(exists(term(form)))
      then return form
      else
          let scoped-form :=
                  apply-terms(apply(applicable-term(form),
                                    form),
                              complete?)
          in
             if complete?
                then return scoped-form
                else return scoped-form || form.
```

The function *apply*(*term*, *form*) returns a wff consisting of the given complex term *term* applied to a form *form* in which it occurs. In addition, the restriction of the complex term is recursively scoped.

```
function apply(term,form);
    let ⟨quant var restrict⟩ := term
    in return
        wff(quant,
            var,
            pull(restrict,false),
            subst(var,term,form)).
```

2.4 The Deterministic Algorithm

For the deterministic version of the algorithm, there are five special language constructs. The unary predicate *empty*(*set*) returns *true* if and only if *set* is empty. Paired braces "{ ... }" constitute a set-forming operator. The binary function *union* applies its second argument (a lambda expression) to each of the elements of its first argument (a set). It returns a corresponding set of the values of the individual applications. The binary infix operator $\cup$ returns the union of its two arguments (both sets). The function *cross-product* takes a list of sets as its argument and returns the set of lists corresponding to each way of taking an element from each of the sets in order. For example,

$$\textit{cross-product}(\ [\{a,b\},\{c,d,e\}]\ ) =$$
$$\{\ [a,c],\ [a,d],\ [a,e],\ [b,c],\ [b,d],\ [b,e]\ \}$$

The function *terms*(*form*) returns the set of all complex terms in *form*. The function *applicable-terms*(*form*) returns the set of all complex terms in *form* that can be applied to *form*.

The deterministic version of the algorithm is identical in structure to the nondeterministic version. Each function operates in the same way as its nondeterministic counterpart, except that they uniformly return sets rather than nondeterministically returning single values.

The algorithm is as follows. The function *gen*(*form*) returns a set of all valid full scopings of the formula *form*.

```
function gen(form);
    return pull(form,true).
```

The function *pull* returns a set of all valid scopings of the formula *form*. If *complete?* is *true*, only full scopings are returned; otherwise, partial scopings are allowed as well.

```
function pull(form, complete?);
    return union( pull-opaque-args(form),
                  lambda(pulled-opaque).
                      apply-terms(pulled-opaque, complete?)).
```

The function *pull-opaque-args*(*form*) returns a set of all wffs generated from *form*, but with arguments in opaque argument positions replaced by a valid scoping of the original value. Since the recursive call to *pull* has *complete?* set to *false*, the unchanged argument is a possibility even for opaque argument positions. When applied to any other type of expression (i.e., a term of some sort), the argument is unchanged.

```
function pull-opaque-args(form)
    if not(wff?(form))
       then return {form}
       else
          let predicate := predicate(form);
               args := arguments(form)
          in return
               union( cross-product(
                          map( integers(1,length(args)),
                               lambda(arg-index).
                                  if opaque(predicate,arg-index)
                                     then pull(args[arg-index], false)
                                     else pull-opaque-args(args[arg-index]))),
                     lambda(args-possibility).
                        {wff(predicate, args-possibility)} ).
```

The function *apply-terms*(*form, complete?*) returns a set of scopings of *form* constituting all of the ways of choosing several terms in *form* and applying them to *form*. If *complete?* is *true* then only the full scopings are returned.

```
function apply-terms(form, complete?);
    if empty(terms(form)))
      then return {form}
      else
          let scoped-forms :=
                  union(applicable-terms(form),
                        lambda(term).
                           union(apply(term, form),
                                 lambda(applied-form).
                                    apply-terms(applied-form, complete?))).
          in
             if complete?
                then return scoped-forms
(3)             else return {form} ∪ scoped-forms.
```

The function *apply*(*term, form*) returns a set of all wffs consisting of the given complex term *term* applied to the form *form* in which it occurs, with the restriction of the complex term recursively scoped in all possible ways.

```
    function apply(term,form);
        let ⟨quant var restrict⟩ := term
        in return
(4)         union( pull(restrict,false),
                   lambda(pulled-restrict).
                      {wff(quant,
                           var,
                           pulled-restrict,
                           subst(var,term,form))}).
```

3 Two Examples

Since the algorithm is not completely transparent, it may be useful to work through the deterministic version for a detailed example.

(5) Some representative of every department in most companies saw a few samples.

The predicate-argument structure of this sentence may be represented as follows:

```
(6)  see(<some r and(rep(r),
                     of(r,<every d and(dept(d),
                                       in(d,<most c co(c)>))>))>,
         <a-few s samp(s)>)
```

Suppose *gen* is called with expression (6) as *form*. Since this is the representation of the whole sentence, *pull* will be called with *complete*? equal to *true*. The call to *pull-opaque-args* will return the original wff unchanged since there are no opaque operators in the wff. We therefore call *apply-terms* on the wff.

In *apply-terms*, the call to *applicable-terms* returns a list of all of the unnested complex terms. For (6), there will be two:

```
(7)  <some r and(rep(r),
                 of(r,<every d and(dept(d),
                                   in(d,<most c co(c)>))>))>
```

```
(8)  <a-few s samp(s)>
```

Each of these complex terms will ultimately yield the wffs in which its variable is the more deeply nested of the two.

The function *apply* is called for each of these complex terms, and inside *apply*, there is a recursive call to *pull* on the restriction of the complex term. This generates all the possible scopings for the restriction. When *apply* is called with (6) as *form* and (7) as *term*, the result of scoping the restriction of (7) will be the following four wffs:

```
(9)     and(rep(r), of(r,<every d and(dept(d),
                                  in(d,<most c co(c)>))>))

(10)    every(d, and(dept(d), in(d, <most c co(c)>)),
                 and(rep(r), of(r, d)))

(11)    most(c, co(c), every(d, and(dept(d), in(d,c)),
                             and(rep(r), of(r,d))))

(12)    every(d, most(c, co(c), and(dept(d), in(d,c))),
                 and(rep(r), of(r, d)))
```

Because this call to *pull* has *complete?* equal to *false*, the unprocessed restriction itself, wff (9), as well as the partially scoped wff (10), is returned along with the fully scoped forms of the restriction. Wff (9) will ultimately generate the two readings in which variables `d` and `c` outscope `r`. Wff (10) is also partial as it still contains a complex term. It will ultimately yield a reading in which `r` outscopes `d` but is outscoped by `c`; the complex term for `c` is still available for an application that will give it wide scope. Wffs (11) and (12) will ultimately yield readings in which `d` and `c` are outscoped by `r`.

Each of these wffs becomes the restriction in a quantified wff constructed by *apply*. Thus, from restriction (10), *apply* will construct the quantified wff

```
(13)    some(r, every(d, and(dept(d), in(d, <most c co(c)>)),
                         and(rep(r), of(r, d))),
                see(r, <a-few s samp(s)>))
```

In *apply-terms* the tail recursion turns the remaining complex terms into quantifiers with wide scope. Thus, in (13) `c` and `s` will be given wider scope than `r` and `d`. For example, one of the readings generated from wff (13) will be

```
most(c, co(c),
        a-few(s, samp(s),
                 some(r, every(d, and(dept(d), in(d, c)),
                                  and(rep(r), of(r, d))),
                         see(r, s))))
```

Sentence (5), by the way, has 14 different readings.

As an example of the operation of the algorithm on a wff with opaque operators, we consider the sentence

Everyone isn't here.

This has the predicate-argument structure

```
not(here(<every x person(x)>))
```

where `not` is an operator opaque in its only argument. The call to *pull-opaque-args* returns the two scopings

```
not(here(<every x person(x)>))
not(every(x,person(x),here(x)))
```

The call to *apply-terms* then turns the first of these into

```
every(x,person(x),not(here(x)))
```

Thus, the following two full scopings are generated:

```
every(x,person(x),not(here(x)))
not(every(x,person(x),here(x)))
```

Note that because of the recursive call in *pull-opaque-args* these two readings will be generated even if this form is embedded within other transparent predicates.

4 Modifications and Extensions

4.1 Restricting Applicable Terms

The notion of *applicable term* used above was quite simple. A complex term was applicable to a wff if it was embedded in no other complex term within the wff. The restriction is motivated by the following consideration. Suppose the input wff is

$$p(< q_1\ x\ r_1(x, < q_2\ y\ r_2(y) >) >)$$

If the embedded term were first applied, yielding

$$q_2(y, r_2(y), p(< q_1\ x\ r_1(x, y) >)$$

the remaining complex term would include a free occurrence of y so that when it is later applied, resulting in the formula

$$q_1(x, r_1(x, y), q_2(y, r_2(y), p(x)))$$

the variable y occurs free in the restriction of q_1.

Thus, it is critical that a term never be applied to a form when a variable that is free in the term is bound outside of it in the form. The simple definition of applicability goes part of the way towards enforcing this requirement.

Unfortunately, this simple definition of applicability is inadequate. If x had itself been free in the embedded complex term, as in the wff

$$p(< q_1\ x\ r_1(x, < q_2\ y\ r_2(x, y) >) >)$$

the application of the outer term followed by the inner term would still leave an unbound variable, namely x. This is because the inner term which uses x has been applied outside the scope of the binder for x. Such structures can occur, for instance, in sentences like the following, where an embedded noun phrase requires reference to its embedding noun phrase.[5]

Every man that I know a child of has arrived.
Every man with a picture of himself has arrived.

In these two sentences the quantifier "a" cannot outscope "every" because the noun phrase beginning with "a" embeds a reference to "every man". If "a" were to outscope "every", then "himself" or the trace following "child of" would be outside the scope of "every man".

The definition of *applicable term* must be modified as follows. A term in a wff is applicable to the wff if and only if all variable occurrences that are free in the term are free in the wff as well. Our previous definition of applicability, that the term be unembedded in another term in the wff, is a simple consequence of this restriction. The versions of the algorithm given in Appendices A and B define the functions *applicable-term* and *applicable-terms* in this way. Given this definition, the algorithm can be shown never to generate unbound variables. (See Appendix C.)

4.2 Adding Ordering Heuristics

A full discussion of heuristic rules for guiding generation of quantifier scopings is outside of the aims of this paper. However, certain ordering heuristics can be incorporated relatively easily into the algorithm merely by controlling the way in which nondeterministic choices are made. We discuss a few examples here, merely to give the flavor for how such heuristics might be added.

For instance, suppose we want to favor the original left-to-right order in the sentence. The function *applicable-terms* should return the complex terms in *right-to-left* order, since quantifiers are extracted from the inside out. The union in line (3) should return $form$ after *scoped-forms*.

If we want to give a noun phrase wide scope when it occurs as a prepositional phrase noun complement to a function word, e.g., "every side of a triangle", then $form$ should come before *scoped-forms* in line (3) when *pull* has been called from line (4) in *apply* where the first argument to *apply* is a complex term for a noun phrase satisfying those conditions, e.g., the complex term for "every side of a triangle".

The modifications turn out to be quite complicated if we wish to order quantifiers according to lexical heuristics, such as having "each" outscope "some". Because of the recursive nature of the algorithm, there are limits to the amount of ordering that can be done in this manner. At the most, we can sometimes guarantee that the best scoping comes first. Of course, one can always associate a score with each reading as it is being generated and sort the list afterwards.

4.3 Nonstandard Input Structures

The algorithm as presented will operate correctly only for input structures which are themselves well-formed. For instance, they must contain no unbound variables. Certain natural language phenomena, such as the so-called donkey sentences, exhibit structures that are ill-formed with respect to the assumptions made by this algorithm. For instance, the sentence

[5] This problem was pointed out to us by Fernando Pereira.

Every man who owns a donkey beats it.

has an ill-formed input structure because the pronoun has to reach inside the scope of an existential quantifier for its antecedent. Its predicate-argument structure might be something like

```
beat(<every m and(man(m),
                  own(m, <some d donkey(d)>))>,
     d)
```

An alternative is to leave the pronoun unanalyzed, in which case the closest reading produced by the algorithm is

```
every(m, and(man(m),
             some(d, and(donkey(d), own(m, d))))
         the(x, it(x), beat(m, x)))
```

In fact, this is not bad if we take `it(x)` to mean that `x` is nonhuman and that `x` is mentioned in the prior discourse in a position determined by whatever coreference resolution process is used. There is a problem if we take the quantifier `the` to mean that there is a unique such `x` and take the sentence to mean that a man who owns many donkeys will beat *every* donkey he owns. But we can get around this if, following the approach taken by Hobbs (1983), we take "a donkey" to be generic, take "it" to refer to the unique generic donkey that `m` owns, and assume that to beat a generic donkey is to beat all its instances.

In any case, modifications to the algorithm would be needed to handle such anaphora phenomena in all their complexity.

5 Conclusion

We have presented an algorithm for generating exactly those quantifier scopings that are consistent with the logical structure of English. While this algorithm can sometimes result in a significant savings over the naive approach, it by no means solves the entire quantifier scoping problem, as we have already pointed out. There has already been much research on the problem of choosing the *preferred* reading among these allowable ones, but the methods that have been suggested need to be specified in an implementation-free fashion more precisely than they have been previously, and they need to be evaluated rigorously on large bodies of naturalistic data. More importantly, methods need to be developed for using pragmatic considerations and world knowledge—particularly reasoning about quantities and dependencies among entities—to resolve quantifier scope ambiguities, and these methods need to be integrated smoothly with the other kinds of syntactic, semantic, and pragmatic processing required in the interpretation of natural language texts.

Acknowledgments

We have profited from discussions about this work with Paul Martin and Fernando Pereira, and from the comments of the anonymous reviewers of the paper.

This research was supported by NIH Grant LM03611 from the National Library of Medicine, by Grant IST-8209346 from the National Science Foundation, and by a gift from the System Development Foundation.

References

[1] Barwise, Jon, and Robin Cooper (1981) "Generalized Quantifiers and Natural Language", *Linguistics and Philosophy*, Volume 4, Number 2, pp. 159-219.

[2] Cooper, Robin (1983) *Quantification and Syntactic Theory*, Reidel, Dordrecht.

[3] Cushing, Steven (1976) "The Formal Semantics of Quantification", Ph.D. dissertation, University of California, Los Angeles.

[4] Grosz, Barbara J., Douglas E. Appelt, Paul Martin, Fernando C. N. Pereira and Lorna Shinkle (1985) "The TEAM Natural-Language Interface System", Final Report, Project 4865, Artificial Intelligence Center, SRI International, Menlo Park, California.

[5] Hobbs, Jerry R. (1983) "An Improper Treatment of Quantification in Ordinary English", *Proceedings, 21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, Massachusetts, pp. 57-63.

[6] Keller, William (1986) "Nested Cooper Storage", paper presented at the Workshop on Word Order and Parsing in Unification Grammars, Friedenweiler, West Germany, 7-11 April, 1986.

[7] Montague, Richard (1973) "The Proper Treatment of Quantification in Ordinary English", in R. Thomason, ed., *Formal Philosophy, Selected Papers of Richard Montague*, Yale University Press, New Haven, Connecticut.

[8] Moore, Robert C. (1981) "Problems in Logical Form", *Proceedings, 19th Annual Meeting of the Association for Computational Linguistics*, Stanford, California, pp. 117-124.

[9] Pereira, Fernando C. N. (1983) "Logic for Natural Language Analysis", Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California.

[10] Woods, William (1977) "Semantics and Quantification in Natural Language Question Answering", *Advances in Computers*, Volume 17, Academic Press, New York, pp. 1-87.

A Prolog Implementation of the Algorithm

The following is the core of a Prolog implementation of the nondeterministic algorithm which includes all but the lowest level of routines. The syntax is that of Edinburgh Prologs, e.g., DEC-20 Prolog.

```
/****************************************************************************

          Prolog Implementation of Scope Generation Algorithm

****************************************************************************/


/*-------------------------------------------------------------------------
                        Representation of wffs:

A wff of the form 'p(arg1,...,argn)' is represented as the Prolog term
wff(p,[arg1',...,argn']) where argi' is the encoding of the
subexpression argi.

A constant term is represented by the homonymous Prolog constant.

A complex term is represented by the Prolog term
term(quant,var,restrict') where restrict' is the encoding of the wff
that forms the restriction of the quantifier.
-------------------------------------------------------------------------*/


% gen(Form,ScopedForm)
% ====================
%
%         Form       ==> a wff with in-place complex terms
%         ScopedForm <== a full scoping of Form

gen(Form, ScopedForm) :-
        pull(Form, true, ScopedForm).


% pull(Form, Complete?, ScopedForm)
% =================================
%
%         Form         ==> a wff with in-place complex terms
%         Complete?    ==> true iff only full scopings are allowed
%         ScopedForm   <== a full or partial scoping of Form
%
%         Applies terms at various level of embedding in Form, including
%         applying to the entire Form, and to opaque argument positions
%         inside Form.
```

```
pull(Form, Complete, ScopedForm) :-
        pull_opaque_args(Form, PulledOpaque),
        apply_terms(PulledOpaque, Complete, ScopedForm).


% pull_opaque_args(Form, ScopedForm)
% ==================================
%
%        Form       ==> a term or a wff with in-place complex terms
%        ScopedForm <== Form with opaque argument positions recursively scoped
%
%        Scopes arguments of the given Form recursively.

pull_opaque_args(wff(Pred,Args), wff(Pred, ScopedArgs)) :- !,
        pull_opaque_args(Pred, 1, Args, ScopedArgs).

pull_opaque_args(Term, Term).


% pull_opaque_args(Pred, ArgIndex, Args, ScopedArgs)
% ==================================================
%
%        Pred       ==> the predicate of the wff whose args are being scoped
%        ArgIndex   ==> the index of the argument currently being scoped
%        Args       ==> list of args from ArgIndex on
%        ScopedArgs <== Args with opaque argument positions recursively scoped
%
%        Scopes a given argument if opaque; otherwise, scopes its
%        subparts recursively.

% No more arguments.
pull_opaque_args(_Pred,_ArgIndex,[],[]) :- !.

% Current argument position is opaque; scope it.
pull_opaque_args(Pred, ArgIndex,
          [FirstArg|RestArgs],
          [ScopedFirstArg|ScopedRestArgs]) :-
        opaque(Pred,ArgIndex), !,
        pull(FirstArg,false,ScopedFirstArg),
        NextIndex is ArgIndex+1,
        pull_opaque_args(Pred, NextIndex, RestArgs, ScopedRestArgs).

% Current argument is not opaque; don't scope it.
pull_opaque_args(Pred, ArgIndex,
          [FirstArg|RestArgs],
          [ScopedFirstArg|ScopedRestArgs]) :-
```

```
        pull_opaque_args(FirstArg,ScopedFirstArg),
        NextIndex is ArgIndex+1,
        pull_opaque_args(Pred, NextIndex, RestArgs, ScopedRestArgs).


% apply_terms(Form, Complete?, ScopedForm)
% ========================================
%
%        Form       ==> a wff with in-place complex terms
%        Complete?  ==> true iff only full scopings are allowed
%        ScopedForm <== a full or partial scoping of Form
%
%        Applies one or more terms to the Form alone (not to any embedded
%        forms.

apply_terms(Form, _Complete, Form) :-
        not(term(Form,_Term)), !.

apply_terms(Form, false, Form).

apply_terms(Form, Complete, ScopedForm) :-
        applicable_term(Form, Term),
        apply(Term, Form, AppliedForm),
        apply_terms(AppliedForm, Complete, ScopedForm).


% apply(Term,Form,NewForm)
% ========================
%
%        Term      ==> a complex term
%        Form      ==> the wff to apply Term to
%        NewForm   <== Form with the quantifier wrapped around it

apply(term(Quant,Var,Restrict),
      Body,
      wff(Quant,[Var,PulledRestrict,OutBody])) :-
        pull(Restrict, false, PulledRestrict),
        subst(Var,term(Quant,Var,Restrict),Body,OutBody).


% applicable_term(Form, Term)
% ===========================
%
%        Form ==> an expression in the logical form language
%        Term <== a top-level term in Form (that is, a term embedded in
%                 no other term) which is not free in any variable bound
%                 along the path from Form to the Term.
```

```
applicable_term(Form, Term) :-
        applicable_term(Form, Term, []).


% applicable_term(Form,Term,BlockingVars)
% ======================================
%
%        Form ==> an expression in the logical form language
%        Term <== a top-level term in Form (that is, a term embedded in
%                 no other term) which is not free in any variable bound
%                 along the path from Form to the Term.
%        BlockingVars ==>
%                 a list of variables bound along the path so far

% A term is an applicable top-level term...
applicable_term(term(Q,V,R),term(Q,V,R), BVs) :-
        % if it meets the definition.
        not(free_in(BVs, R)).

% An applicable term of the restriction or body of a quantifier is applicable
% only if the variable bound by the quantifier is not free in the term.
applicable_term(wff(Quant,[Var,Restrict,Body]),Term, BVs) :-
        quantifier(Quant), !,
        (applicable_term(Restrict,Term,[Var|BVs]);
         applicable_term(Body,Term,[Var|BVs])).

% An applicable term of an argument list is an applicable term of the wff.
applicable_term(wff(_Pred,Args),Term, BVs) :-
        applicable_term(Args, Term, BVs).

% An applicable term of any argument is an applicable term of the whole
% list.
applicable_term([F|R],Term, BVs) :-
        applicable_term(F,Term,BVs) ;
        applicable_term(R,Term,BVs).

% Note the absence of a rule looking for applicable terms inside of
% complex terms.  This limits applicable terms to be top-level.
```

B COMMON LISP Implementation of the Algorithm

The following is the core of a COMMON LISP implementation of the deterministic algorithm which includes all but the lowest level of routines.

```
;;;*************************************************************************
;;;
;;;        COMMON LISP Implementation of Scope Generation Algorithm
;;;
;;;************************************************************************/


;;;-------------------------------------------------------------------------
;;;                         Representation of Wffs
;;;
;;; A wff of the form 'p(arg1,...,argn)' is represented as the
;;; s-expression (p arg1' .. argn') where argi' is the encoding of the
;;; subexpression argi.
;;;
;;; A constant term is represented by the homonymous LISP atom.
;;;
;;; A complex term is represented by the s-expression (:term quant
;;; var restrict') where restrict' is the encoding of the wff that forms
;;; the restriction of the quantifier.
;;;-------------------------------------------------------------------------

;;;-------------------------------------------------------------------------
;;; Implementation notes:
;;;
;;; The following simple utility functions are assumed:
;;;
;;; map-union -- implements the binary function UNION
;;; cross-product -- implements the function CROSS-PRODUCT
;;; opaque -- implements the binary function OPAQUE
;;; integers -- implements the binary function INTEGERS
;;;
;;; The infix union is implemented with CL function UNION.
;;; The binary prefix union is implemented under the name MAP-UNION
;;;   to avoid conflict with the CL function UNION.
;;; The function APPLY is implemented under the name APPLY-Q to avoid
;;;   conflict with the CL function APPLY.
;;;-------------------------------------------------------------------------


(defun gen (form)
  (pull form t))
```

```
(defun pull (form complete?)
  (map-union (pull-opaque-args form)
             (function (lambda (pulled-opaque)
                         (apply-terms pulled-opaque complete?)))))


(defun pull-opaque-args (form)
  (if (not (wff? form))
      (list form)
      (let ((predicate (first form))
            (args      (rest form)))
        (map-union (cross-product
                    (mapcar (function (lambda (arg-index)
                              (if (opaque predicate arg-index)
                                  (pull (nth (- arg-index 1) args)
                                        nil)
                                   (pull-opaque-args (nth (- arg-index 1)
                                                          args)))))
                            (integers 1 (length args))))
                   (function (lambda (args-possibility)
                     (list (cons predicate args-possibility)))))))))


(defun apply-terms (form complete?)
  (if (null (terms form))
      (list form)
      (let ((scoped-forms
             (map-union
              (applicable-terms form)
              (function (lambda (term)
                          (map-union
                           (apply-q term form)
                           (function (lambda (applied-form)
                                       (apply-terms applied-form
                                                    complete?)))))))))
        (if complete?
            scoped-forms
          (adjoin form scoped-forms)))))


(defun apply-q (term form)
  (let ((quant     (second term))
        (var       (third term))
        (restrict  (fourth term)))
    (map-union (pull restrict nil)
               (function (lambda (pulled-restrict)
                 (list
```

```
                  (list quant var pulled-restrict
                        (subst var term form)))))))


(defun applicable-terms (form)
  (applicable-terms1 form '()))


(defun applicable-terms1 (form blocking-vars)
  (cond ((atom form)
         '())
        ((and (term? form)
              (not-free-in blocking-vars (fourth form)))
         (list form))
        ((term? form)
         '())
        ((and (wff? form)
              (quantifier? (first form)))
         (union (applicable-terms1 (third form)
                                   (cons (second form) blocking-vars))
                (applicable-terms1 (fourth form)
                                   (cons (second form) blocking-vars))))
        (t (mapcan (function (lambda (arg)
                               (applicable-terms1 arg blocking-vars)))
                   (cdr form)))))
```

C Proofs of Algorithm Properties

This appendix includes informal proofs of some important properties of the nondeterministic version of the presented algorithm. First, we present a proof of the termination of the algorithm. Several criteria of the partial correctness of the algorithm are also informally shown, especially, that the algorithm does not generate wffs with unbound variables.

However, we do not prove correctness in the sense of showing that the algorithm is semantically sound, i.e., that it yields wffs with interpretations consistent with the interpretation of the input expression, simply because we do not provide a semantics for the input language. (The output language, of course, has a standard logical semantics.)

We do not attempt to prove completeness for the algorithm, as the concept of completeness is open to interpretation, depending as it does on just which scopings one deems possible, but we expect that the algorithm is complete in the sense that every permutation of quantifiers respecting the considerations in the introduction is generated. We also do not prove the nonredundancy of the nondeterminism in the algorithm, i.e., that the algorithm will not generate the same result along different nondeterministic paths, although we believe that the algorithm is nonredundant.

C.1 Notation

We will use lower Greek letters (α, β, ...) as variables ranging over expressions in the logical form language.

We inductively define a metric ρ on expressions in the logical form language as follows:

$$\rho(\alpha) \equiv \begin{cases} 2 + \rho(r) & \text{if } \alpha \text{ is a complex term } < q\ v\ r > \\ 1 + \sum_{i=1}^{n} \rho(\alpha_i) & \text{if } \alpha \text{ is a wff } f(\alpha_1, \ldots, \alpha_n) \text{ and } \sum_{i=1}^{n} \rho(\alpha_i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Informally, ρ is a measure of the embedding depth of the complex terms in an expression.

C.2 Termination

We will give an informal proof of termination for the nondeterministic algorithm by induction on this metric ρ. But first, we present without proof three simple but useful properties of the metric.

Lemma 1 *If α is a wff, then $\rho(\alpha) = 0$ if and only if α contains no complex terms.*

Lemma 2 *If α is a wff and β is a subexpression of α and $\rho(\alpha) > 0$ then $\rho(\beta) < \rho(\alpha)$.*

Lemma 3 *If α is a wff and β is a subexpression of α and $\rho(\alpha) = 0$ then $\rho(\beta) = 0$.*

We now prove the following theorem, and its corollary which gives the termination of the algorithm. We assume that calls to the auxiliary functions *wff, term, wff?, term?, predicate, arguments, opaque, map, exists, not, applicable-term, subst,* and so forth always terminate if the computation of their arguments terminates.

Theorem 1 *For all expressions α the following six conditions hold:*

Condition 1: *pull-opaque-args(α)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$,

Condition 2: for all complex terms t in α, *apply(t,α)* terminates with result β such that $\rho(\beta) < \rho(\alpha)$,

Condition 3: *apply-terms(α,true)* terminates with result β such that $\rho(\beta) = 0$,

Condition 4: *apply-terms(α,false)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$,

Condition 5: *pull(α,true)* terminates with result β such that $\rho(\beta) = 0$,

Condition 6: *pull(α,false)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$.

Proof: We first prove the base case, for $\rho(\alpha) = 0$. By Lemma 1, α must contain no complex terms. Three of the conditions are easily proved.

Condition 2: Since, by Lemma 1, there are no complex terms in α, this condition holds vacuously.

Condition 3: Again, the absence of complex terms in α causes the call to *apply-terms* to return with result α, and $\rho(\alpha) = 0$, so the condition holds.

Condition 4: Similarly, and $\rho(\alpha) \leq \rho(\alpha)$ trivially.

Conditions 1,5, and 6: These conditions follow directly from Lemma 4 given below.

Lemma 4 *For all expressions α such that $\rho(\alpha) = 0$, pull(α, x) and pull-opaque-args(α) terminate with result α.*

Proof: The proof is by a simple induction on the length of the expression, and uses the base case for conditions 3 and 4 proved above.

For the induction step we assume the induction hypotheses that the six conditions hold for all α such that $\rho(\alpha) < n$ and prove the conditions for $\rho(\alpha) = n$, for $n > 0$. The conditions are proved sequentially. In particular, earlier conditions for the case $\rho(\alpha) = n$ are used in the proofs of later ones. (Since there is no use of later conditions in earlier ones, this does not introduce any circularity in the proof.)

Condition 1: We must show that *pull-opaque-args(α)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$. If α is not a wff, then the condition holds vacuously, so we assume that $\alpha = f(\alpha_1, \ldots, \alpha_k)$. By definition of ρ, $\rho(\beta) \leq 1 + \sum_{i=1}^{k} \rho(\beta_i)$. (The inequality is necessary because $\rho(\beta)$ may be zero.) Now β_i is either α_i or *pull(α_i,false)*. In the first case, $\rho(\beta_i) \leq \rho(\alpha_i)$ trivially. In the second case, since α_i is a subexpression of α, by Lemma 2 we have that $\rho(\alpha_i) < \rho(\alpha)$ and we can use the induction hypothesis to show the termination of the call to *pull*. Also by the induction hypothesis, $\rho(\beta_i) \leq \rho(\alpha_i)$.

Thus, we see that in either case, $\rho(\beta_i) \leq \rho(\alpha_i)$. So $\rho(\beta) \leq 1 + \sum_{i=1}^{k} \rho(\beta_i) \leq 1 + \sum_{i=1}^{k} \rho(\alpha_i) = \rho(\alpha)$.

Condition 2: We must show that for all terms t in α, *apply(t,α)* terminates with result β such that $\rho(\beta) < \rho(\alpha)$. Suppose $t =< q\ v\ r >$. Then $\beta =$ *apply(*$< q\ v\ r >$*,*α*)* $= q(v, \gamma, \delta)$ where $\gamma =$ *pull(r,false)* and $\delta =$ *subst(v,*$< q\ v\ r >$*,*α*)*.

Now, let $\rho(r) = m$. By Lemma 2, $m < n$. So by the induction hypothesis, the computation of γ terminates and $\rho(\gamma) \leq m$. Also, the computation of δ is assumed to terminate (as mentioned above) with δ missing the complex term t that occurs in α (and possibly other complex terms embedded within t). So $\rho(\delta) \leq \rho(\alpha) - \rho(t) = n - (2 + \rho(r)) = n - 2 - m$. Finally, by definition of ρ we have $\rho(\beta) \leq 1 + \rho(\gamma) + \rho(\delta) \leq 1 + m + n - 2 - m = n - 1 < n$.

We will use the two conditions just proved in the proofs of the final four conditions.

Condition 3: We must show that *apply-terms(α,true)* terminates with result β such that $\rho(\beta) = 0$. By Lemma 1, we know that complex terms exist in α so the **else** clause is taken. Let $t =$ *applicable-term*(α) and $\gamma =$ *apply*(t, α). By, the second condition just proved above, the latter computation terminates with $\rho(\gamma) \leq \rho(\alpha) - 1 < n$. Now, let $\epsilon =$ *apply-terms*(γ, *true*). Again by the induction hypothesis, this computation terminates with $\rho(\epsilon) = 0$. Since *complete?* = *true* we return ϵ as β, so $\rho(\beta) = 0$ as required.

Condition 4: We must show that *apply-terms(α,false)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$. By Lemma 1, we know that complex terms exist in α so the **else** clause is taken. Let $t =$ *applicable-term*(α) and $\gamma =$ *apply*(t, α). By, the second condition just proved above, this computation terminates with $\rho(\gamma) \leq \rho(\alpha) - 1 < n$. Now, let $\epsilon =$ *apply-terms*(γ, *false*). Again by the induction hypothesis, this computation terminates with $\rho(\epsilon) \leq \rho(\gamma) < n$. Since *complete?* = *false* we return ϵ or α as β. In either case, $\rho(\beta) \leq \rho(\alpha)$ as required.

We will use the four conditions just proved in the proofs of the final two conditions.

Condition 5: We must show that *pull(α,true)* terminates with result β such that $\rho(\beta) = 0$. Let $\gamma =$ *pull-opaque-args*(α). By the first condition just proved above, we know this computation terminates and $\rho(\gamma) \leq n$. Now, let $\epsilon =$ *apply-terms*(γ, *true*). Again by the third condition just proved above, this computation terminates with $\rho(\epsilon) = 0$. Since *complete?* = *true* we return ϵ as β, so $\rho(\beta) = 0$ as required.

Condition 6: We must show that *pull(α,false)* terminates with result β such that $\rho(\beta) \leq \rho(\alpha)$. The argument is similar to that for condition 3. Let $\gamma =$ *pull-opaque-args*(α). By the first condition just proved above, we know this computation terminates and $\rho(\gamma) \leq n$. Now, let $\epsilon =$ *apply-terms*(γ, *false*). Again by the fourth condition just proved above, this computation terminates with $\rho(\epsilon) \leq \rho(\gamma) \leq n$. Since *complete?* = *false* we return either ϵ or α as β. In either case $\rho(\beta) \leq \rho(\alpha)$ as required.

This completes the proof of the six conditions, given the induction hypotheses, and thus completes the inductive proof of the theorem.

Corollary 1 *For all wffs α, gen(α) terminates with result β such that β has no complex terms as subexpressions.*

Proof: This follows immediately from the fifth condition in Theorem 1 and Lemma 1.

C.3 Correctness

We consider several criteria for correctness of the algorithm. Let $U(\alpha)$ = the set of variables which are unbound in α and $V(\alpha)$ = the set of variables which are vacuously quantified in α.[6] We show that if input expression α is well-formed, that is, has no unbound variables and no vacuous quantifiers ($U(\alpha) = V(\alpha) = \emptyset$), and if $\beta = gen(\alpha)$, then

Criterion 1: β has no complex terms;

Criterion 2: β has no unbound variables ($U(\beta) = \emptyset$);

Criterion 3: β has no vacuous quantifiers ($V(\beta) = \emptyset$);

Criterion 4: for every complex term t in α, there is a quantifier in β which binds the same variable as t and has the position held by t in α in its body; and

Criterion 5: for every quantifier q in β, there is either a quantifier in α or a complex term in α which binds the same variable.

Proof of these five statements does not constitute a proof of correctness, but provides motivation for assuming the correctness of the algorithm. As unbound variables in the output are the prime symptom of problems with previous algorithms, we take these criteria to be the most critical for indicating correctness.

The first criterion follows directly from Corollary 1.

The second and third criteria are a consequence of the following theorem which we prove informally.

Theorem 2 *For all expressions α such that $U(\alpha) = u = \{u_1, ..., u_m\}$ and $V(\alpha) = v = \{v_1, ..., v_n\}$, and for $b \in \{true, false\}$ and for β any of gen(α), pull(α,b), pull-opaque-args(α), apply-terms(α,b), and apply(applicable-term(α),α), $U(\beta) = u$ and $V(\beta) = v$.*

Proof: Again, the proof is by induction on $\rho(\alpha)$, but we will be less formal in demonstrating the well-foundedness of the induction. The base case is trivial because, as shown in the proofs of Theorem 1 and Lemma 4, the functions all return their argument unchanged when $\rho(\alpha) = 0$. For the induction step, we will merely show that each function maintains the unbound variables and vacuous quantifiers assuming that all the others do. The previous proof of termination provides the well-foundedness of this proof.

[6] A variable v is vacuously quantified in an expression α if and only if v is bound in a subexpression of α, a quantified wff of the form $q(v, r, s)$ and v does not occur free in r or s. This definition implies that variables bound by complex terms are never vacuously quantified.

apply(applicable-term(α),α): We must show that if $t =< q\ x\ r >$ is an applicable term in α and $U(\alpha) = u$ and $V(\alpha) = v$ then $U(apply(t, \alpha)) = u$ and $V(apply(t, \alpha)) = v$ as well.

The unbound variables u in α can be divided into two (possibly overlapping) sets u_r and u_s, where u_r consists of those variables in u that occur in r and u_s consists of those variables in u that occur outside of t in α. Note that $u = u_r \cup u_s$. Now assume x occurs in r. Then $U(r) = \{x\} \cup u_r \cup u_0$ where u_0 is the set of variables bound within α but outside of t and which occur free in r. But t is an applicable term, and by the definition of "applicable term" u_0 must be empty. So $U(r) = \{x\} \cup u_r$. (If x does not occur in r, a similar argument shows that $U(r) = u_r$.)

Let $r' = pull(r, false)$ and $s = subst(x, t, \alpha)$. By the induction hypothesis, $U(r') = \{x\} \cup u_r$. Since s does not include t (which binds x) but does include x, $U(s) = \{x\} \cup u_s$. In forming the quantified wff $\beta = q(x, r', s)$, the unbound variables in β consist of those in r' and those in s minus x, that is, $U(\beta) = [(\{x\} \cup u_s) \cup (\{x\} \cup u_r)] - \{x\} = u_s \cup u_r = u$. (If x does not occur in r, similar arguments show that $U(r') = u_r$, $U(s) = \{x\} \cup u_s$ and $U(\beta) = [(\{x\} \cup u_s) \cup u_r] - \{x\} = u_s \cup u_r = u$.)

Vacuous quantified variables can be divided similarly into v_r (those bound vacuously in r) and v_s (those bound vacuously outside of t in α). Again, $v = v_r \cup v_s$. Trivially, $V(r) = v_r$. By induction, $V(r') = v_r$ also. Since s does not include t, $V(s) = v_s$. $V(\beta) = V(r) \cup V(s) = v$ unless the quantification of x in β is vacuous. Since x is guaranteed to occur in s (as it replaces t in s), the quantification is clearly not vacuous. So $V(\beta) = v$.

apply-terms(α,b): This follows straightforwardly from the previous subproof for *apply* and the induction hypothesis for *apply-terms*.

pull-opaque-args(α): If α is not a wff, then the proof is trivial. Otherwise, there are two cases, depending on whether the predicate in α, p, is or is not a quantifier. If p is not a quantifier, then the result follows immediately from the induction hypothesis for *pull* and *pull-opaque-args*.

If p is a quantifier, then let $\alpha = p(x, r, s)$. The output β then is $wff(p, pull\text{-}opaque\text{-}args(x), pull\text{-}opaque\text{-}args(r), pull\text{-}opaque\text{-}args(s))$. The first call to *pull-opaque-args* merely returns x. Now by an argument similar to that given in the subproof for *apply*, the unbound variables in α can be exhaustively divided into u_r and u_s depending on whether they occur in r and s. Depending on whether x occurs in r, $U(r) = \{x\} \cup u_r$ or $U(r) = u_r$. Similarly, $U(s) = \{x\} \cup u_s$ or $U(s) = u_s$. Suppose the second and third calls to *pull-opaque-args* return r' and s' respectively. By the induction hypotheses $U(r') = U(r)$ and $U(s') = U(s)$. If the quantification of x in α is not vacuous, then x occurs free in either r or s (and by induction in r' or s') so $U(\beta) = \{x\} \cup u_r \cup u_s - \{x\} = u$. If the quantification of x is vacuous, then $U(r') = u_r$ and $U(s') = u_s$ and $U(\beta) = u$.

Vacuous quantified variables can be divided into v_r and v_s similarly. Suppose the quantification of x is vacuous (i.e., x does not occur free in r or s). Then $V(\alpha) = \{x\} \cup v_r \cup v_s$. By the induction hypothesis, $V(r') = V(r) = v_r$ and $V(s') = V(s) = v_s$. Also by induction, x does not occur free in r' or s'. Therefore, the quantification of x in β is also vacuous and $V(\beta) = \{x\} \cup v_r \cup v_s = V(\alpha)$.

If the quantification of x is not vacuous, then $v = v_r \cup v_s$ and x occurs free in either r or s. By induction, x occurs free in either r' or s' so the quantification of x in β is also non-vacuous. Also by induction as before, $V(r') = v_r$ and $V(s') = v_s$, so $V(\beta) = V(r') \cup V(s') = v_r \cup v_s = V(\alpha)$.

***pull(α,b)*:** This follows directly using the previously proved induction steps for *apply-terms* and *pull-opaque-args*.

***gen(α)*:** This follows directly using the previously proved induction step for *pull*.

This concludes the proof of the induction step and the theorem.

The second and third criteria follow from the presumed well-formedness of α and Theorem 2 which demonstrates that *gen* maintains well-formedness.

The fourth and fifth criteria we argue informally as follows: Since no complex terms occur in β (by Corollary 1) we can assume that every complex term t in α was applied (i.e., the first argument of *apply*) at some time in the processing of α. But if it was applied, then it must have been an applicable term occurring in the wff it was applied to (as the only call to *apply* is of this form). Then the call to *subst* in *apply* will not be vacuous, the quantifier will bind the same variable as t and will outscope the position held by t in α. Thus the fourth criterion holds. Also note that all quantifiers in β are either the result of such an application or were in α originally. Thus the fifth criterion follows immediately as well.