Unification Encodings of Grammatical Notations

Stephen G. Pulman
SRI International Cambridge Computer
Science Research Centre* and
University of Cambridge Computer
Laboratory

This paper describes various techniques for enriching unification-based grammatical formalisms with notational devices that are compiled into categories and rules of a standard unification grammar. This enables grammarians to avail themselves of apparently richer notations that allow for the succinct and relatively elegant expression of grammatical facts, while still allowing for efficient processing for the analysis or synthesis of sentences using such grammars.

1. Introduction

Formalisms equivalent to, or based on, unification grammars of the type exemplified by PATR (Shieber 1984) are very widely used in computational linguistics (Alshawi 1992; van Noord et al. 1990; Briscoe et al. 1987; Bobrow, Ingria, and Stallard 1991, etc.) A unification-based formalism has many well-known virtues: it is declarative, monotonic, reversible (in principle at least); it has a well-understood formal interpretation (Shieber 1986, Smolka 1992, Johnson 1988); and there exist well-understood and relatively efficient parsing and generation algorithms for grammars using such a formalism (Shieber 1988; Haas 1989; Alshawi 1992; Shieber et al. 1990, inter alia).

However, a pure unification formalism is often thought to be a somewhat restricted grammatical formalism, especially when compared with the rich devices advocated by many grammarians. The recent literature (Pollard and Sag 1987, 1993, etc.) uses many devices that go beyond pure unification: set valued features; negation and disjunction; tests for membership; list operations like "append" and "reverse"; multiple inheritance hierarchies; as well as Kleene operators like * or +1, which are familiar from linguistics textbooks although they have no direct equivalent in a unification grammar.

Unfortunately, there is a price to pay for this increase in expressive power: a decrease in efficiency. Whereas there exist several relatively efficient implemented systems for parsing and generating with wide-coverage "pure" unification grammars (Alshawi 1992; Briscoe et al. 1987), more complex formalisms have not so far led to systems of comparable efficiency. At the present time, I am not aware of any practical implementations using these more complex descriptive devices remotely comparable to the (relative) efficiency of pure unification-based systems when used with wide-coverage grammars and large lexica. This is not a claim that the efficiency problem is solved even for pure unification grammars, but it is at least less of a problem than for these richer formalisms.

* Suite 23, Miller's Yard, Mill Lane, Cambridge CB2 1RQ, UK

© 1996 Association for Computational Linguistics

It would therefore be desirable to combine the efficiency of pure unification-based systems with the availability of richer grammatical formalisms. One route to this happy state of affairs would be to develop efficient processing mechanisms for the richer devices directly. However, this route involves a research program of uncertain length and outcome, given the known complexity properties of many of the richer descriptive devices.

This paper describes an alternative approach towards such a combination, via the compilation of apparently richer grammatical notations into expressions whose satisfaction can be checked by unification alone. Compilation of these apparently richer devices into expressions that can be processed just using unification will generally allow the grammarian to use them freely, without necessarily sacrificing the advantages of efficiency that pure unification systems offer. I say "without necessarily sacrificing" efficiency, because some compilation strategies may actually make matters worse. For example, a naive compilation of disjunction into many alternative rules and lexical entries, combined with an equally naive parsing algorithm, may produce worse behavior than an implementation that interprets the disjunctions directly.

The paper describes a variety of apparently richer descriptive devices that can be compiled into unification grammars in ways that under normal circumstances will result in efficient processing. Some I believe to be original; others have been described elsewhere in the literature in some form, although often in a way that makes it difficult for computational linguists to appreciate their significance. Still others are known mostly by word of mouth, in the unification grammar community. The intention of the present paper is to describe them all in an accessible form (hence the more tutorial tone than is usually found in this journal) and thus attempt to narrow the gap between rich grammatical formalisms and efficient practical implementations.

Of course, you don't get anything for nothing in this game. There will still be cases where the full power of the richer formalisms is necessary. The various techniques described here are often limited in their applicability, applying to only a subset of the problems that one would like to solve. Furthermore, some of the techniques described can lead, in the worst case, to overwhelmingly large structures and consequent processing inefficiency. Nevertheless, practical experience has shown that (with care and some experimentation) it is possible to develop linguistic descriptions that are succinct and relatively elegant, while still lending themselves to efficient (and most importantly, bidirectional) processing.

2. A Unification Formalism

To begin with, we will define a basic unification grammar formalism. For convenience, we will use many of the notational conventions of Prolog.

A **category** consists of a set of feature equations, written:

```
{f1=v1,f2=v2, . . . ,fN=vN}
```

Feature names are atoms; **feature values** can be variables (beginning with an uppercase character), atoms (beginning with a number or a lowercase character) or categories. For example:

```
{f1=X,f2=yes,f3={f4=1,f5=X}}
```

Coreference is indicated by shared variables: in the preceding example, `f1` and `f5` are constrained to have the same value. We often use underscore (_) as a variable if we are not interested in its value.

For convenience and readability, we shall also allow as feature values **lists** of values, **n-tuples** of values, and Prolog-like **terms**:

```
{f1=[{f2=a},{f3=b}],f4=(c,d,e),f5=foo(X,Y,Z)}
```

These constructs can be regarded as "syntactic sugar" for categories. For example, a term `foo(X,Y,Z)` could be represented as a category `{functor=foo,arg1=X,arg2=Y, arg3=Z}`. Tuples can be thought of as fixed-length lists, and lists can be defined as categories with features `head` and `tail`, as in Shieber (1986). We will use the Prolog notation for lists: thus `[bar|X]` stands for the list whose head is `bar` and whose tail (a list) is `X`.

A **lexical item** can be represented by a category. For example:

```
{cat=n,count=y,number=sing,lex=dog}
{cat=det,number=sing,lex=a}
{cat=verb,number=sing,person=3,subcat=[],lex=snores}
```

A **rule** consists of a **mother category** and a list of zero or more **daughter categories**. For example:

```
{cat=s} ==> [{cat=np,number=N,person=P},
             {cat=vp,number=N,person=P}]
```

A rule could equivalently be represented as a category, with distinguished features `mother` and `daughters`:

```
{mother={cat=s},
 daughters=[{cat=np,number=N,person=P},
            {cat=vp,number=N,person=P}]}
```

However, we will stay with the more traditional notation here.

Various simple kinds of **typing** can be superimposed on this formalism. We can distinguish a particular feature (say `cat`) as individuating different types and associate with each different value of the `cat` feature a set of other dependent features. This will only be a sensible thing to do if we know that the value of the `cat` feature will always be instantiated when types are checked. We will write such declarations as:

```
category(np,{person,number}).
category(verb,{person,number,subcat}).
```

The intent of declarations like this is to ensure that an NP or a verb always has these and only these feature specifications. One of the practical advantages of such a regime is that different categories can now be compiled into terms whose functor is the value of the `cat` feature, and whose other feature values can be identified positionally: for example, `{cat=np,number=sing,person=3}` would compile to `np(3,sing)`. And in turn the advantage of this is that ordinary first order term unification (i.e., of the type (almost) provided by Prolog implementations) can be used in processing, guaranteeing almost linear performance in category matching.

It is often convenient to use a slightly different notation when adopting such a regime, to make clear that one particular feature value has a privileged status. Thus we will frequently write:

```
np:{person=3,number=sing}
```

to mean:

```
{cat=np,person=3,number=sing}.
```

We can also provide type declarations for features. We will assume a set of primitive types like `atom` or `category`, and allow for complex types also:

```
feature(person, atom({1,2,3})).  % value must be an atom in declared set
feature(lex, atom).              % value must be any atom
feature(subcat, list(category)). % value must be a list of categories
```

We can also, if required, use a simple type of feature default to make the categories written by a grammarian more succinct:

```
default(person,3).
default(number, noun, sing).
```

The effect of the first statement would be to ensure that at compile time, the feature `person` will be instantiated to 3 if it does not already have a value (of any kind). The second statement restricts the application of the default to members of the category `noun`. We will often assume that such defaults have been declared to make the various example rules and entries more succinct.

It is also very often convenient to allow for **macros**, expanded at compile time, to represent in a readable form commonly occurring combinations of features and values. We will assume that such macros are defined in ways suggested by the following examples, and that at compile time, the arguments (if any) of the defined macro are unified with the arguments of the instance of it in a rule or lexical item. In some cases, the results of macro evaluation may need to be spliced into a category: for example, when the result is a set of feature specifications.

```
macro(transitive_verb(Stem),
  v:{lex=Stem,subcat=[np:{}]}).

macro(phrasal_verb(Stem,Particle),
  v:{lex=Stem,subcat=[np:{},p:{lex=Particle}]})

macro(thread_gaps(Mother,LeftDaughter,RightDaughter)) :-
     Mother        =   {gapin=In,gapout=Out},
     LeftDaughter  =   {gapin=In,gapout=Nxt},
     RightDaughter =   {gapin=Nxt,gapout=Out}.
```

Thus the grammarian might now write:

```
transitive_verb(kick).
phrasal_verb(switch,off).

s:{A,f1=v1,...} ==> [np:{B,f2=v2,..}, vp:{C,f3=v3,...}]
 where thread_gaps(A,B,C).
```

and these will be expanded to:

```
v:{lex=kick,subcat=[np:{}]}
v:{lex=switch,subcat=[np:{},p:{lex=off}]}

s:{gapin=I,gapout=O,f1=v1,...} ==>
  [np:{gapin=I,gapout=N,f2=v2,...},
  vp:{gapin=N,gapout=O,f3=v3,...}]
```

Notice that the values of variables in categories like `s:{A,...}` need to be spliced in when the macro is evaluated at compile time.

Finally we will point out that multiple equations for the same feature on a category are permitted (where they are consistent). Thus a rule like:

```
a:{f=V} ==> [b:{},c:{f=V,f=d:{f1=a}}]
```

is valid, and means that the value on c of f, which may be only partly specified, will be the same on category a.

This completes our definition of a basic unification grammar formalism. While the notational details vary, the basic properties of such formalisms will be very familiar. We turn now to descriptive devices not present in the formalism as defined so far, and to ways of making them available.

3. Kleene Operators

Kleene operators like * (0 or more) or + (1 or more) are frequently used in semi-formal linguistic descriptions. In a context-free-based formalism they must actually be interpreted as a notation for a rule schema, rather than as part of the formalism itself: something like `A -> B C* D` is a shorthand for the infinite set of rules:

```
A -> B D, A -> B C D, A -> B C C D, etc.
```

While not essentially changing the weak generative capacity of a CFG, the use of Kleene operators does change the set of trees that can be assigned to sentences: N-ary branching trees can be generated directly.

In some unification-based formalisms (e.g. Briscoe et al. 1987; Arnold et al. 1986) Kleene operators have been included. However, in the context of a typed unification formalism like ours, the exact interpretation of Kleene operators is not completely straightforward. Some examples will illustrate the problem. In a formalism like that in Arnold et al. (1986), grammarians write rules like the following, with the intent of capturing the fact that an Nbar can be preceded by an indefinite number of Adjective Phrases provided that (in French, for example) they agree in gender, etc., with the Nbar:

```
np:{agr=A} ==>
   [...,adjp:{agr=A}*, nbar:{agr=A}]
```

This is presumably intended to mean that if an AdjP is present, with agr instantiated to some value, then succeeding instances of AdjP must have agr bound to the same value, as must the Nbar. But a rule like this does not make clear what is intended for the values of any features on an AdjP not mentioned on the rule. Presumably it is not intended that all such values are shared, for otherwise such a rule would parse the first two of the following combinations, but not the third, which simply contains the concatenation of the adjectives appearing in the first two:

```
adjp:{agr=m, foo=a} nbar:{agr=m}
adjp:{agr=m, foo=b} nbar:{agr=m}
adjp:{agr=m, foo=a} adjp:{agr=m, foo=b} nbar:{agr=m}
```

Alternatively, the intention might be that only features explicitly mentioned on the rule are to be taken account of when "copying" the Kleene constituent. But this is still not an interpretation that is likely to be of much practical use. Unification formalisms like ours are intended to be capable of encoding semantic as well as syntactic descriptions. In order to properly combine the meaning of the AdjP* with that of the Nbar (as a conjunction, say), to give the meaning of the mother NP, some feature on the AdjP* like sem=... will at least have to be mentioned in building the NP meaning. But this very fact will mean that the interpretation of all the AdjPs encountered will be constrained to have the same value for sem as the first one processed. This is clearly not what the grammarian would have intended. The grammarian presumably wanted the value of the sem feature to depend on the AdjP actually present, while wanting

the value of the agr feature to be set ultimately by the Nbar. Unfortunately, it is not possible to combine these conflicting requirements.

At this point the reader might well wonder why Kleene operators were wanted in the first place. In most grammars, Kleene * is used for two different reasons. In the first type of case, like that just illustrated, it is used when it is not known how many instances of a category will be encountered. (PP or adverbial modification of VP is a similar case.) Under these circumstances, it is in fact very often the case that a recursive analysis is empirically superior. For example, an English NP rule like:

```
np:{}==> [det:{}, adjp:{}*, nbar:{}]
```

actually makes it impossible to capture Nbar co-ordination (unless it is treated as ellipsis). In phrases like:

```
there is no alternative analysis or clever trick
```

in order to get the correct syntax and interpretation, *alternative analysis or clever trick* has to be treated as a conjunction of premodified Nbars. On an analysis that treats the construction recursively, this is no problem.

The second reason for which Kleene * is used is to get a flat structure, where there is no evidence for recursion. Examples of this might be, on some analyses, the German "middle field"; and some types of coordination. For these cases, it is genuinely important to have some way of achieving the effect of Kleene operators.

In our formalism, there are several ways of achieving an equivalent effect. The easiest and most obvious way is to turn the iteration into recursion, with the necessary flat structure being built up as the value of a feature on the highest instance of the recursive expansion. The following schematic rules show how this can be done:

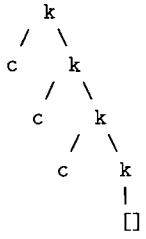
```
kleene:{kcat=C,kval=[]} ==>  []
                  % terminate the recursion

kleene:{kcat=C,kval=[C|T]} ==>  [C, kleene:{kcat=C,kval=T}]
                  % find a C, followed by C*
```

For the other Kleene operators (+, +2, etc.), instead of the first Kleene rule terminating the recursion with an empty category, it terminates with one, two, or however many instances of the category are required. With a suitable macro definition for *, a grammarian can now write rule 1 in the form of rule 2, which will be expanded to 3:

1. `a:{} ==>  [b:{}, c:{}*, d:{}]`
2. `a:{} ==>  [b:{}, *(c:{},C), d:{}]`
3. `a:{} ==>  [b:{}, kleene:{kcat=c:{},kval=C}, d:{}]`

A sequence of three cs will be parsed with a structure:



This structure is, of course, recursive. However, a flat list of the occurrences of c is built up as the value of kval on the topmost Kleene category. Anything that the flat constituent structure was originally needed for can be done with this list, the extra levels introduced by the recursion being ignored.

It is, however, possible to get a flatter *tree* structure more directly, and also to overcome the problem with features used for semantic composition. In order to do this we take advantage of the fact that our formalism allows us to write rules with variables over lists of daughters.[1] We assume a category **kleene** with three category valued features: finish, kcat (kleene category), and next. We enrich the grammatical notation with a * which can appear as a suffix on a daughter category in a rule. Thus our grammarian might write something like:

```
np:{agr=A} ==> [det:{agr=A}, adj:{agr=A}*, n:{agr=A}]
```

This is then compiled into a set of rules as follows:

1.
```
np:{agr=A} ==>
   [det:{agr=A},
    kleene:{finish=[n:{agr=A}],kcat=adj:{agr=A},next=N}
    | N]
```

The original category appears as the value of the kcat feature, and the categories that followed this one in the original rule appear as the value of the finish feature. The value of the feature next is a variable over the tail of the daughters list, in a way reminiscent of many treatments of subcategorisation.

2.
```
kleene:{finish=F,kcat=adj:{agr=A,f1=V1,....},
        next=[kleene:{finish=F,kcat=adj:{agr=A},next=N}|N]}  ==>
        [adj:{agr=A,f1=V1,....}]
```

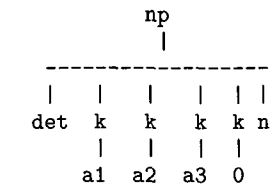
In rule 2, the kleene category is rewritten as an adj, which will share *all* its features with the value of kcat. The value of next is another instance of the kleene category, which shares the value of the finish feature, and where the value of the *kcat* feature is the adj category *as it appeared on the original rule.* This ensures that only the features mentioned on the kleene category will be identically instantiated across all occurrences, enabling the semantic problem mentioned earlier to be solved (at least in principle: the current illustration does not do so). Clearly when the mother of this rule is unified with the corresponding daughter of rule 1, the effect will be to extend the list of daughters of rule 1 by adding the value of next. Since this value is itself a list, now consisting of a kleene category and a variable tail, the resulting structure can again be combined with a following kleene category having the appropriate values. This process can continue *ad infinitum.*

3.
```
kleene:{finish=F,next=F} ==> []
```

The third rule (which is general and so only need occur once in the compiled grammar) terminates the iteration by extending the daughters of rule 1 by the sequence of categories that appeared in the original rule.

1 A referee has pointed out that this is akin to the metavariable facility of some Prolog systems (Clark and McCabe 1984), and that a somewhat similar technique, in the context of DCGs, is described by Abramson 1988).

Now a sequence `det adj1 adj2 adj3 n` will be parsed having the following structure:



The values of the `adj` daughters to `kleene` will be present as the value of `kcat`, and so for all practical purposes this tree captures the kind of iterative structure that was wanted.

In some cases, the extra level of embedding that this method gives might actually be linguistically motivated. In this case, the idea behind the compilation just described can be incorporated into the analysis directly. To give an illustration, the following grammar generates indefinitely long, flat, NP conjunctions of the "John, Mary, Bill, . . . , and Fred" type.

1. `np:{flatconj=y} ==>`
   `[np:{flatconj=n,next=MoreNPs} | MoreNPs]`
2. `np:{flatconj=n,next=[np:{flatconj=n,next=MoreNPs} | MoreNPs]} ==>`
   `[np:{ . . . }, comma:{}]`
3. `np:{flatconj=n, next=[]} ==>`
   `[conj:{}, np:{}]`

These rules will give a structure:

`[NP [NP ,][NP ,][NP ,] . . . [and/or NP]]`

The trick is again in the unification of the value of the feature `next` on the daughter of rule 1 and the mother of rule 2. This unification extends the number of daughters that rule 1 is looking for. Rule 3 terminates the recursion. The feature `flatconj` stops spurious nestings, if they are not wanted.

In English, at least, this type of conjunction is the only construction for which a Kleene analysis is convincing, and they can all be described satisfactorily in this manner.

4. Boolean Combinations of Feature Values

Our formalism does not so far include Boolean combinations of feature values. The full range of such combinations, as is well known, can lead to very bad time and space behavior in processing. Ramsay (1990) shows how some instances of disjunction can be avoided, but there are nevertheless many occasions on which the natural analysis of some phenomenon is in terms of Boolean combinations of values.

One extremely useful technique, although restricted to Boolean combinations of atomic values, is described by Mellish (1988). He gives an encoding of Boolean combinations of feature values (originally attributed to Colmerauer) in such a way that satisfiability is checked via unification. This technique is used in several systems (e.g. Alshawi 1992; the European Community's ALEP (Advanced Linguistic Engineering Platform) system; Alshawi et al. 1991). We describe it again here because we will need to know how it works in detail later on.

Given a feature with values in some set of atoms, or product of sets of atoms, any Boolean combination of these can be represented by a term. The encoding proceeds as follows, for a feature f with values in {1,2} * {a,b,c}. We want to write feature equations like:

```
f=1
f=b
f=1&b                ; 1 and b
f=(a;b)&2          ; either a or b, and 2
f=~2               ; not 2
f=(1->b)&(~1->c) ; if 1 then b, else c
f=2<->c            ; 2 if and only if c
```

To encode these values we build a term with a functor, say bv (for Boolean vector) with N+1 variable arguments, where N is the size of the product of the sets from which f takes its values. In the example above, N=6, so bv will have seven arguments. Intuitively, we identify each possible value for f with the position between arguments in bv:

```
bv(_ , _ , _ , _ , _ , _ , _).
     1   1   1   2   2   2
     a   b   c   a   b   c
```

In building the term representing a particular Boolean combination of values, what we do is work out, for each of these positions, whether or not it is excluded by the Boolean expression. The simple way to do this is to build the models as sets of atoms, and then test the expression to see if it holds of each one. For example, take f=(a;b)&2. The models are

{{1,a},{1,b},{1,c},{2,a},{2,b},{2,c}}.

An atomic expression like a holds of a model if it is a member, and fails otherwise: a here only therefore holds of the two models containing a. Truth functions of atoms can be interpreted in the obvious way. The feature value of f above holds only of {2,a} and {2,b}. Thus all other combinations are excluded.

For each position representing an excluded combination we unify the variable arguments on each side of it. In our example this gives us:

```
bv(A , A , A , A , B , C , C).
     1   1   1   2   2   2
     a   b   c   a   b   c
```

Finally, we instantiate the first and last argument to different constants, say 0 and 1. Because of the shared variables, this will give us:

```
bv(0 , 0 , 0 , 0 , B , 1 , 1).
     1   1   1   2   2   2
     a   b   c   a   b   c
```

The reasoning behind this last step is that if all the possibilities are excluded, then all the variables will be linked. But if all the possibilities are excluded, then we have an impossible structure and we want this to be reflected by a unification failure. If we know that the first and last arguments are always incompatible, then an attempt to link up all the positions will result in something that will be trying to unify 0 and 1, and this will fail, as required.

Notice that the number of arguments in the term that we build for one of these Boolean expressions depends on the size of the sets of atomic values involved. This can grow rather big, of course.

Sometimes, it happens that although the set of possible values for a feature is very large, we only want to write Boolean conditions on small subsets of those values. A typical case might be a feature encoding the identifier of a particular lexical item: in English, for example, the various forms of *be* often require extra constraints (or relaxation of constraints) which do not apply to other verbs. However, we would not want to build a term with N+1 arguments where N is the number of verbs in English.

Under these circumstances there is a simple extension of this encoding. Assume the feature is called `stem`. We encode the set of values as something like: `{be,have,do,anon}`, where `anon` is some distinguished atomic value standing for any other verb. Then we can write things like:

```
stem=be
stem=~(be;have)
stem=have;do
etc.
```

However, to express the constraints we need to express, the encoding has to be a little more complex. We could build a term of N+1 arguments, as before, where N=4. But now all the items that fall under `anon` will be encoded as the same term. This means that we are losing information: we cannot now use the `stem` feature to distinguish these verbs. What we have to do is to give the `bv` functor another argument, whose values are those of the original feature: in our example, all the different verb stems of English. In other respects we encode the values of the `stem` feature as before, but with the extra argument the encodings now look like:

```
be:      stem=bv(be, 0 , 1 , 1 , 1 , 1)
                     b   h   d   anon

have:    stem=bv(have, 0 , 0 , 1 , 1 , 1)
                       b   h   d   anon

expect:  stem=bv(expect, 0 , 0 , 0 , 0 , 1)
                         b   h   d   anon

decide:  stem=bv(decide, 0 , 0 , 0 , 0 , 1)
                         b   h   d   anon
```

The extra argument can now distinguish between the `anon` verbs. Everything else works just as before.

This extension can also be generalized to products of large sets. For example, we might want a feature whose value was in the product of the set of letters of the alphabet and positive whole numbers. And let us suppose that we want to exclude some particular combinations of these. The particular constraints we need to write might figure in the grammar as:

```
id=~(c&(12;13))
```

That is, everything except `c&12` and `c&13`. At compile time, when we have examined the whole grammar and lexicon, we know which values are actually mentioned, and we can represent the value space of this feature as: `{c,anon1} * {12,13,anon2}`, where `anon1` and `anon2` are again atoms standing in for all the other values. We need two extra arguments this time, and then expressions like `g&444`, `c&13`, and `(c&(12;13))` will be coded as:

```
f2=bv(g,444,0 , 0 , 0 , 0 , 0 , 0 , 1)
              c   c   c   a1  a1  a1
              12  13  a2  12  13  a2
```

```
f2=bv(c,13,0 , 0 , 1 , 1 , 1 , 1 , 1)
             c   c   c   a1  a1  a1
             12  13  a2  12  13  a2

f2=bv(c, _ ,0 , A , B , 1 , 1 , 1 , 1)
              c   c   c   a1  a1  a1
              12  13  a2 12  13   a2
```

Notice that for the original Boolean expressions, we may not be able to fill in all the extra argument places.

4.1 Implementation

Implementation of this technique requires the grammar writer to declare a particular feature as being able to take values in some Boolean combination of atoms, for example, something like:

```
bool_comb_feature(agr,[[1,2,3],[sing,plur]]).
```

Lists of lists of atoms represent the subsets whose product forms the space of values.

To compile the value of a particular `bool_comb_feature` when in the grammar, first, using the declarations, precompute the set of models (i.e., the space of values). Assume this set has N members. Then, for each feature=value equation, construct for the value an N+1 vector whose first member is 1 and whose last is 0, and where all the other members are initially distinct variables. Now encode the feature value into this vector as follows:

```
for i = 1 to N-1,
  if feature value does not hold of the i'th model in the set
  then unify vector positions i and i+1.
```

If the models in the set are represented as lists of atoms, then a single atom as feature value holds of (is true in) a model if it is a member of the list representing the model, a conjunction of atoms holds if both conjuncts hold, etc.

To implement the extensions just described requires only the addition of the right number of extra argument places to hold the original atoms, where relevant.

5. Type Hierarchies and Inheritance

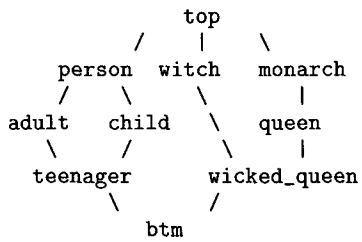
Type hierarchies are becoming as ubiquitous in computational linguistics as they have been in knowledge representation. There are several techniques for compiling certain kinds of hierarchy into terms checkable by unification: Mellish (1988) describes them. The version presented here derives from a more general approach to the implementation of lattice operations by Aït-Kaci et al. 1989, which shows how to implement not only efficient unification of terms (greatest lower bound, "glb") in a type lattice but also of generalization (least upper bound, "lub") and complement.

We will restrict our attention to hierarchies of the type described by Carpenter (1992, Chapter 1), i.e., bounded complete partial orders, (but using the terminology of Aït-Kaci (1986). Carpenter's lattices are upside down, and so for him unification is "least upper bound" and so on.) We further restrict our attention to hierarchies of atomic types. (While in principle the encoding below would extend to non-atomic (but still finite) types, in practice the resulting structures are likely to be unmanageably large.)

In our presentation, we make these hierarchies into lattices: they always have a top and bottom element and every pair of types has a glb and lub. Having a glb of

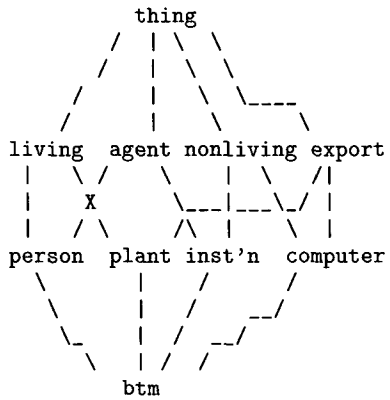
btm is read as failure of unification. Having a lub of top means that the two types do not share any information.

One example of such a lattice is Aït-Kaci (1986, 223)



A teenager is both an adult and a child; a queen is a monarch, etc. The glb of adult and child is teenager; the lub is person.

The lattice that we will use for illustration is:



Notice how easy it is to get a lattice that does not obey our constraints. By adding a line from exports to person (either the slave trade or the brain drain) we get a situation where exports and living no longer has a greatest lower bound, although this would be a perfectly natural inheritance link to want to add.

To encode the information in this lattice in a form where glb and lub can be computed via unification we first make an array representing the reflexive transitive closure of the "immediately dominates" relation, which is pictured in the diagram above by lines.

```
[b,t,a,i,l,n,p,p,c,e]
[1,0,0,0,0,0,0,0,0,0]btm
[1,1,1,1,1,1,1,1,1,1]thing
[1,0,1,1,0,0,1,0,0,0]agent
[1,0,0,1,0,0,0,0,0,0]institution
[1,0,0,0,1,0,1,1,0,0]living
[1,0,0,1,0,1,0,0,1,0]non_living
[1,0,0,0,0,0,1,0,0,0]person
[1,0,0,0,0,0,0,1,0,0]plant
[1,0,0,0,0,0,0,0,1,0]computer
[1,0,0,0,0,0,0,1,1,1]exports
```

In each row we put a 1 if the row element dominates the column element, (i.e., column is a subtype of row) and a 0 otherwise. Since everything is a subtype of itself, and btm is a subtype of everything, there is a 1 in each of the diagonal cells, and in the cell for btm on each row. Taking the agent row, we also have a 1 for the institution column

and a 1 for the person column. We will refer to such a row as a "bitstring," although as we have represented it, it is a list rather than a string. (The sharp-eyed reader will see various other list and term representations of things that are logically bitstrings in what follows. I apologize for this abuse of terminology, but have got into the habit of calling them bitstrings.)

This is the first step of the encoding technique described by Aït-Kaci et al. (1989). They point out that what the rows of this array represent is the set of lower bounds of the row element, via a bitstring encoding of sets. Thus the AND of two rows will represent the set of lower bounds they have in common. This will in fact be the case whether or not the lattice has the properties we are assuming. If it does not, then it will be possible for the bitstring representing the lower bounds of the two types to be distinct from any row. In our case, however, the bitstring will always coincide with one row exactly. This row will represent the glb of the two types.

Unfortunately, however, ANDing of bitstrings is not the kind of operation that is directly available within the unification formalism we are compiling into. So we have to encode it into a unification operation. For this we can turn again to the Colmerauer encoding of Boolean combinations of values.

Informally, we regard a bitstring like those in the rows of the array above as a representation of the disjunction of the members of the set of lower bounds of the type. So the row for agent:

```
[b,t,a,i,l,n,p,p,c,e]
[1,0,1,1,0,0,1,0,0,0]agent
```

is regarded as meaning "btm or agent or institution or person." Then we can encode the bitstring directly into a Boolean vector term of the kind we discussed earlier. The term will have N+1 arguments, where N is the length of the bitstring, and adjacent arguments will be linked if their corresponding bitstring position is zero, and otherwise not linked. The term corresponding to the bitstring for agent will then be:

```
bv(A,B,B,C,D,D,D,C,C,C,C)
bv(0,B,B,C,D,D,D,1,1,1,1)
    b t a i l n p p c e
```

before and after instantiation of the first and final arguments to 0 and 1, respectively, respectively.

The term corresponding to the living bitstring will be:

```
bv(A,B,B,B,B,C,C,D,E,E,E)
bv(0,B,B,B,B,C,C,D,1,1,1)
    b t a i l n p p c e
```

Unifying the two terms together:

```
bv(0,B,B,C,D,D,D,1,1,1,1)
bv(0,E,E,E,E,F,F,G,1,1,1)
=
bv(0,B,B,B,B,B,B,1,1,1,1)
```

When we decode this, by the reverse translation (identical adjacent arguments means 0), we get:

```
bv(0,B,B,B,B,B,B,1,1,1,1)
=  [1,0,0,0,0,0,1,0,0,0]
```

which is the bitstring for person, the greatest lower bound of the two types agent and living, as required.

With this encoding, unification will never fail, since every pair of types has a glb, even if this is `btm`. However, since having a glb of `btm` is usually meant to signal that two types are incomparable and thus do not have a glb, it would be more useful if we could contrive that unification would fail for such cases. In the usual Colmerauer encoding, an impossible Boolean combination is signaled by all the arguments being shared. This will cause an attempt to unify the first and last arguments of the term, which, being 0 and 1, will cause the unification to fail. Such a failure will never happen in our encoding thus far: since the entry for `btm` in each bitstring is 1, there will always be one adjacent argument pair unlinked, and so unification will always succeed.

If, on the other hand, we simply omit `btm` from the list of types, then when two types have no lower bound, the result of ANDing together their corresponding bitstring will be a bitstring consisting entirely of zeros. Thus, unifying any two Boolean vector terms that results in the term encoding such a bitstring will fail: if all the elements are zero, then all the arguments will be linked, and we will be trying to unify 0 and 1. Everything else will work just as before.

We have been dealing with type hierarchies that have the property of being bounded complete partial orders, except that we have added a `btm` element to ensure that every pair of types has a glb. Hierarchies of this sort, when they have a top element, have the defining property of lattices that every pair of types has both a glb and lub. Being complete lattices, they also have the property that they can be inverted, by taking "immediately dominates" to be "immediately dominated by." Furthermore, what in the original lattice was the glb of two types is now the lub and vice versa. Hence, by computing an array based on the inverse relation one can use exactly the same technique for computing least upper bounds, or the generalization of two types.

The array generated for the inverted lattice is:

```
[b,t,a,i,l,n,p,p,c,e]
[1,1,1,1,1,1,1,1,1,1]btm
[0,1,0,0,0,0,0,0,0,0]thing
[0,1,1,0,0,0,0,0,0,0]agent
[0,1,1,1,0,1,0,0,0,0]institution
[0,1,0,0,1,0,0,0,0,0]living
[0,1,0,0,0,1,0,0,0,0]non_living
[0,1,1,0,1,0,1,0,0,0]person
[0,1,0,0,1,0,0,1,0,1]plant
[0,1,0,0,0,1,0,0,1,1]computer
[0,1,0,0,0,0,0,0,0,1]exports
```

The glb of, say, `person` and `plant` is:

```
[0,1,1,0,1,0,1,0,0,0]person
AND
[0,1,0,0,1,0,0,1,0,1]plant
=
[0,1,0,0,1,0,0,0,0,0]living
```

which corresponds to the lub in the original lattice.

However, the notion of generalization captured in this way is not distributive (because the lattice is not). If it were, then we should expect the following combinations to yield the same result, where `g` and `u` represent generalization and unification:

```
g(u(A,B),C) = u(g(A,C),g(B,C))
```

Whereas in the lattice we are using for illustration, some choices for A, B and C, do have this property (e.g., A=agent, B=person, C=living), other choices (e.g., A=person, B=plant, C=computer) do not.

```
g(u(agent,person),living) = u(g(agent,living),g(person,living))
      person                        thing          living
             living                        living

g(u(person,plant),computer) = u(g(person,computer),g(plant,computer))
      btm                             thing            exports
            computer                          exports
```

We would do well to require distributivity, for otherwise, operations on lattices will become order dependent. In order to do this we have to make our original lattice a distributive one, making new disjunctive types. We can achieve this effect by instead taking our original lattice (the right way up) and using bitwise disjunction of elements to represent generalizations.

```
[1,0,0,0,0,0,1,0,0,0]person
OR
[1,0,0,0,0,0,0,1,0,0]plant
=
[1,0,0,0,0,0,1,1,0,0]
```

However, notice that this bitstring does not correspond to any existing row in the original array. It corresponds instead to the disjunctive object {person;plant}. This object is extensionally identical to the type living: in decoding we can recover this fact by finding a bitstring which has a 1 in (at least) every position that the bitstring describing the disjunctive object has a 1, and as few as possible 1s other than this. This bitstring will be the description of living. In general, to identify the equivalent object for some "virtual" type we take the type description X and find the least object Y such that the generalization of X and Y equals Y.

Unfortunately, for these lattices I have not been able to find a way of encoding generalization as disjunction of bitstrings in such a way that the resulting encoding will interact with the previous encoding of unification as conjunction of bitstrings. So it is possible to have either generalization or unification, but not *both* within the same feature system, at least with this encoding.

5.1 Implementation

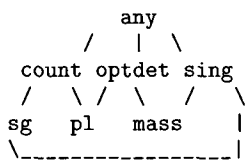
In the context of linguistic descriptions the types concerned are often categories, i.e., non-atomic entities. The compilation technique given here assumes that the types are atomic. Of course, where the ranges of feature values are finite, hierarchies of non-atomic types can always be expanded into hierarchies of atoms. It is likely that the resulting encodings would be rather large (although Aït-Kaci et al. (1989) describe some compaction techniques). It is thus unlikely that the compilation technique would be able to completely compile away the complex non-atomic type hierarchies used in, say, HPSG.

However, a useful compromise is to add to our formalism a new type of feature, whose values are members of an implicit lattice of atomic types. We will illustrate with a partial analysis along these lines of agreement in NPs in English. Traditionally, agreement in NPs is taken to be governed by at least three features: person and number (often combined in a feature "agr") and something like "mass/count." The person feature is only relevant for subject-verb agreement, but at least number and mass/count are necessary to get the right combinations of determiner (or no determiner) and noun in the following:

```
the/a/some man
the/*a/some men
the/*a/some furniture
```

```
*the/*a/*some furnitures
*man
men
furniture
*furnitures
```

We can express the appropriate generalizations quite succinctly by defining a feature whose values are arranged in a hierarchy:



From the basic traditional types of count (sg and pl) and mass nouns we construct two supertypes: sing(ular) and opt(ional)det(erminer).

The grammarian needs to add a declaration describing the set of types and the partial order, expressed as immediate dominance, on them.

```
partial_order_feature(agr,
                [any:[count,optdet,sing],
                 count:[sg,pl],
                 optdet:[pl,mass],
                 sing:[mass,sg]]).
```

From this declaration it is easy to compute the array representing the reflexive transitive closure of immediately dominates:

```
[a,m,c,s,p,s,o]
[1,1,1,1,1,1,1]any
[0,1,0,0,0,0,0]mass
[0,0,1,1,1,0,0]count
[0,0,0,1,0,0,0]sg
[0,0,0,0,1,0,0]pl
[0,1,0,1,0,1,0]sing
[0,1,0,0,1,0,1]optdet
```

Now it is easy to precompute for each atomic type, represented by a row of the array, a vector like that for `bool_comb_feature`. In this case, each vector will have nine elements, and adjacent positions will be linked if the corresponding column element is 0.

Given such a feature, the following rules and lexical entries are sufficient to account for the data above, where in a more traditional feature-based approach we would have had multiple entries for the determiners, and two rules for the determiner-less NPs: one for the case of a mass noun, the other for the plurals.

```
np:{agr=A} ==> [n:{agr=optdet,agr=A}]
np:{agr=A} ==> [det:{agr=A},n:{agr=A}]

the:  det:{agr=any}
a:    det:{agr=sg}
some: det:{agr=sing}
```

Of course, we could achieve a similar result by using the Boolean feature combinations described earlier. We could define a feature with values in `{sing,plur}*{mass,count}` and provide rules and entries with the appropriate Boolean combinations of these. This will always be possible, so, strictly speaking, the encodings we have described are not

necessary. However, there are two reasons for maintaining the type inheritance encoding separately from the Boolean feature combination. Firstly, although in many cases the Boolean encoding might, as here, seem to have a size advantage, in general this need not be the case, especially when the techniques for compaction of bit arrays described by Aït-Kaci et al. (1989) are used. Secondly, and perhaps more importantly for the grammarian, in many cases using the Boolean combination would be a linguistically inaccurate solution. Having a definition like that just given implies that it is just an accident that there are no `mass&plur` NPs, since they are a linguistically valid combination of features, according to the declaration. In this case, and similar ones, the description in terms of type inheritance would be regarded as capturing the facts in a more natural and linguistically motivated way.

6. Threading and Defaults

The technique of gap threading is by now well known in the unification grammar literature. It originates with Pereira (1981) and has been used to implement *wh*-movement and other unbounded dependencies in several large grammars of English (Bobrow, Ingria, and Stallard 1991; Pulman 1992).

The purpose of this section is to point to another use of the threading technique, which is to implement a rather simple, but very useful, notion of default: a notion that is, however, completely monotonic!

Consider the following problem as an illustration. In an analysis of the English passive, we might want to treat the semantics in something like the following way:

```
Joe was seen
 = exists(e,see(e,something,joe))
Joe was seen by Fred
 = exists(e,see(e,fred,joe))
Joe was seen on Friday by Fred
 = exists(e,see(e,fred,joe) & on(e,friday))
Joe was seen on Friday in Cambridge by Fred
 = exists(e,see(e,fred,joe) & in(e,cambridge) & on(e,friday))
Joe was seen on Friday by Fred in Cambridge
 = exists(e,see(e,fred,joe) & on(e,friday)) & in(e,cambridge)
```

Whether we assume a flat structure for the VP modifiers:

```
[vp pp pp . . . ]
```

or a recursive one:

```
[[[vp pp] pp] pp]
```

the problems for the grammarian writing a compositional semantic rule for sentences like this are:

(i) where the verb meaning is encountered, we do not know whether there is an explicit agent phrase

(ii) if there is no agent, we need to put in an existential quantifier of some sort (`something`).

(iii) if there is an agent phrase, we need to put in the meaning of the NP concerned in subject position

(iv) if there is an agent phrase, it may come at any distance from the verb: in particular, we cannot be guaranteed that it will be either the lowest of

the highest of the VP modifiers in such sentences. (If we could, then the problem could be solved either at the lowest VP level, or at the point where the VP is incorporated into the S).

We can formulate this generalization in terms of defaults: the default subject for a passive is `something`, unless an explicit agent is present, in which case it is that agent. We can encode this analysis literally in terms of threading. The basic idea is that we thread an `agent=(In,Out)` feature throughout the VP. On the passive verb itself, the features will look like this:

```
v:{lf=see(Subj,Obj),agent=(Subj,something),subj=Obj,...}
```

The surface subject is the semantic object (passed down via the feature `subj`. The semantic subject is the value of the In part of the `agent` feature. This feature sends the default value `something` as the `Out` value. We arrange things so that this value is threaded throughout the VP and returns as the value of the `In` part if no agent phrase is present. In other words, any PP that is not an agent phrase simply threads the value of this feature unchanged. If an agent phrase is present, the logical form of the NP in it replaces the `something` and becomes the `Out` value. The topmost VP, which is a constituent of the S, will unify the `In` and `Out` values so that either the default agent meaning, or an actual agent meaning is therefore transmitted eventually back to the verb. (If we require the `In` value of the agent phrase to be the default value for an agent i.e., a meaning like `something`, then this threading analysis has the incidental advantage that only one agent phrase can be present.)

Some schematic rules to achieve this might be:

```
s:{lf=Vsem} ==> [np:{lf=S}, vp:{lf=Vsem,subj=S,agent=(A,A)}]
```

The sentence semantics is taken from VP. The NP `subj` meaning is sent down to the head V to be put in its first argument position. The `Agent` thread is passed on by unifying `In` and `Out` values.

```
vp:{lf=PPsem,subj=S,agent=(in,N2)}  ==>
    [vp:{lf=Vsem,subj=S,agent=(in,N1)},
     pp:{agent=(N1,N2),lf=PPsem,vlf=Vsem}]
```

The mother VP meaning is taken from the PP, which will simply conjoin its own meaning to that of the daughter VP if the PP is a modifier. If the PP is an agent, it will pass up the daughter VP meaning. PP meanings are functions from VP meanings to VP meanings (or more generally, from predicates to predicates).

```
pp:{agent=(A,A),lf=and(Vsem,...), vlf=Vsem}  ==>
   [p:{}, np:{}]
```

A nonagentive PP conjoins its meaning to that of the VP, and passes the agent thread unchanged.

```
pp:{agent=(something,NPsem),lf=Vsem, vlf=Vsem}  ==>
   [p:{}, np:{lf=NPsem}]
```

An agentive PP replaces the default agent value with that of the agentive NP and passes up the daughter VP meaning.

```
vp:{lf=Vsem,subj=S,agent=(In,something)}  ==>
   [v_be:{}, v_passive:{lf=Vsem, subj=In,obj=S}]
```

This rule introduces a passive form of V as complement to `be`, and sends up the default agent meaning.

I described this technique as expressing a limited notion of default. There are several linguistic concepts which seem amenable to an analysis in terms of such a technique. A notable example might be the LFG concepts of functional completeness and coherence. In all implementations of LFG parsers that I am aware of, such checks are built into the parsing algorithm. However, it should instead be possible to achieve the same effect by compiling the unification part of an LFG grammar in such a way that completeness and coherence are checked via unifiability of two features: one going up, saying what a verb is looking for by way of arguments, and one coming down, saying what has actually been found.

6.1 Implementation

The easiest way to implement this use of threading is by defining and using macros such as those given earlier for illustration. Some implementations (e.g., Karttunen 1986) build threading into the grammar compiler directly, but this can lead to inefficiency if features are threaded where they are never used.

7. Threading and Linear Precedence

Threading can also be used as an efficient way of encoding linear precedence constraints. This has been most recently illustrated within the HPSG formalism by Engelkamp, Erbach, and Uskoreit (1992). Given a set of some partial ordering constraints and a domain within which they are to be enforced, the implementation of this as threading proceeds as follows.

Firstly, given a set of constraints of the form `a < c, b < d`, etc., where each of `a, b, c, d` is some kind of category description, then add to each instance of the category that can appear within the relevant domains some extra features encoding what is not permitted to appear to the left or right on each category within that domain. How this is done depends entirely on what features and categories are involved: we could use Boolean combinations of atomic values, category valued features, or, as in the example below, a pair of term-valued features, `left` and `right`.

Secondly, for each relevant rule introducing these categories in the given domains, we need to identify among the daughters some kind of head-complement or governor-governed relation. Exactly what this is does not matter: if there is no intuitively available notion, it can be decided on an ad hoc basis. The purpose of this division is simply to identify one daughter whose responsibility it is to enforce ordering relations among its sisters and to transmit constraints on ordering elsewhere within the domain, both downwards to relevant subconstituents, and upwards to constituents containing this one but still within the domain within which the ordering must be enforced. On each relevant rule we need a feature on the mother and the distinguished daughter, here called `store`, following the terminology of Engelkamp, Erbach, and Uskoreit (1992), and threading features, `in` and `out` or their equivalent, on the relevant sister constituents.

To illustrate the technique in the simplest possible form, here is a small grammar for an artificial language. The language consists of any sequence of four `y`s from the set `{a,b,c,d}` within a constituent labeled `x`, provided that the LP constraints `a < c, b < d` are observed.

First we encode the categories in question with the LP constraints in terms of what can precede and follow them. We represent this as a tuple, with a position for each of the relevant categories: `(a,b,c,d)`. The feature `left` encodes what can precede, and

right what may follow a category. If a member of the tuple cannot precede or follow the current category we put a no in that position of the tuple, otherwise we leave it uninstantiated.

Next we thread a similar tuple through each category to record which category it is. Thus the position in the tuple for a b must have a b in that position in the out value. All the other positions are simply linked by shared variables.

```
/* lexical entries:  LP = a < c, b < d */

y:{lex=a,in=(_,B,C,D),out=(a,B,C,D),left=(_,_,no,_)}
y:{lex=b,in=(A,_,C,D),out=(A,b,C,D),left=(_,_,_,no)}
y:{lex=c,in=(A,B,_,D),out=(A,B,c,D),right=(no,_,_,_)}
y:{lex=d,in=(A,B,C,_),out=(A,B,C,d),right=(_,no,_,_)}

/* rules: */

sigma:{}  ==>  [x:{}]
```

This rule just says that an x is a valid parse.

```
x:{store=S} ==> [y:{out=S}]
```

An x can consist of just a y. The store of the x is the out value of the y. In the other rules, x acts as the distinguished daughter, and y as the subsidiary daughter.

```
x:{store=A} ==>
  [y:{out=A,in=B,right=B}, x:{store=B}]
```

When the distinguished daughter follows the subsidiary, the right value of the subsidiary must be unified with its in value and the store of the distinguished daughter. This means that any y categories following this one will be recorded in the store of the x daughter, and will have to be consistent with the constraints recorded on this y daughter's right feature.

The out value of the subsidiary daughter is passed to the mother category's store. Thus the mother contains a record both of the distinguished daughter's store, and what has been added to it by the subsidiary daughter.

```
x:{store=A} ==>
  [x:{store=B}, y:{out=A,in=B,left=B}]
```

This rule illustrates what to do when the distinguished daughter precedes the subsidiary one. Otherwise, things are exactly analogous. Of course, if both of these rules are used, there will be a lot of ambiguity in these "sentences": they are just to illustrate the different possibilities.

7.1 Implementation

This approach to partial ordering can be implemented by requiring the grammarian to make linear precedence declarations encoding the partial orderings. (If grammars obey the "Exhaustive Constant Partial Ordering" property (Gazdan et al. 1985, 49) one global statement will be sufficient). Then, for each domain, the relevant rules have to be annotated with an indication of the daughter that is to be treated as the distinguished one.

We define (for each domain) five features (earlier called store, left, right, in, and out) whose values will be tuples of length N, where there are N different categories figuring in the partial order declaration. The members of the tuple will be categories, each associated with a fixed position, or a negative element (here represented as no)

which will not unify with any of these categories. Intuitively, `left` and `right` encode what can precede or follow the category they appear on; `in` and `out` encode what actually does precede or follow; and `store` encodes the information to be passed up the tree.

Now when compiling the grammar (and lexicon), for each category figuring in a linear precedence statement $C_a < C_b$, do the following:

1. add to C_a the feature specification
   `left=(...,no,...)`
   where `no` is in the position associated with C_b and all other positions have an anonymous variable;
2. add to C_b the feature specification
   `right=(...,no,...)`
   where `no` is in the position associated with C_a and all other positions have an anonymous variable;
3. add to $C_{a/b}$ the feature specifications

   in = $(X_1, \ldots, \_, \ldots, X_n)$

   out= $(X_1, \ldots, C_{a/b}, \ldots, X_n)$ where _ and $C_{a/b}$ are in the positions associated with $C_{a/b}$ and the other positions in these two features are linked by shared variables $X_1 \ldots X_n$ as indicated.

Finally, for each annotated rule with distinguished daughter D, mother M, and subsidiary daughter S:

1. put `store=X` on M and `out=X` on S
2. put `store=Y` on D
3. if D < S put `in=Y,left=Y` on S; if S < D put `in=Y,right=Y` on S.

Macros, perhaps automatically generated by the compiler in response to the declaration, can be used to effect these feature constraints economically.

8. Threading and Set Valued Features

The threading technique can also be used to implement some of the effect of set valued features. We represent a set as a tuple of values, e.g. `(a,b,c,d)`. Each member of the set encodes its presence by changes to this tuple on an `in` and `out` feature: thus `a` would have `(no,B,C,D)` as its `in` value and `(a,B,C,D)` as its `out` value. Then on the category representing the domain within which all the members of the set are to be found, we give `(no,no,no,no)` as the value of `in`, and `(a,b,c,d)` as the value of `out`. These values will be satisfied if and only if all the members of the set have been encountered, in any order.

Here is a small grammar which implements a kind of set-valued subcategorization analysis. The language consists of sequences of a verb (`vabcd`, `vbcd`, or `vbd`) followed by the things it is subcategorized for, in any order: e.g.

```
vabcd a b c d
vabcd b a d c, etc.
*vabcd a b c        % d missing
*vabcd a b d c d    % too many ds
```

Here are the categories (for simplicity regarded as lexical) that can appear on subcat lists:

```
y:{lex=a,in=(no,B,C,D),out=(a,B,C,D)}
y:{lex=b,in=(A,no,C,D),out=(A,b,C,D)}
y:{lex=c,in=(A,B,no,D),out=(A,B,c,D)}
y:{lex=d,in=(A,B,C,no),out=(A,B,C,d)}
```

Here are the verbs:

```
x:{lex=vabcd,in=(a,b,c,d),out=(no,no,no,no)}
x:{lex=vbcd,in=(no,b,c,d),out=(no,no,no,no)}
x:{lex=vbd,in=(no,b,no,d),out=(no,no,no,no)}
```

Notice that by putting the negative element in the relevant position on both the `in` and `out` tuple we require that that member of the set should not be found at all.

And now the rules:

```
sigma:{} ==>
      [x:{in=A,out=A}]
```

This rule unifies the `in` and `out` values to make sure that what was found was what was being sought; `x` might typically be a VP, for example, and this identification of feature values would take place on the `s==>[np,vp]` rule.

```
x:{in=In,out=Out} ==>
  [x:{in=In,out=Nxt},y:{in=Nxt,out=Out}]
```

This is like a subcat schema which combines an `x`-projection with a `y`-complement, threading the appropriate information.

This simple technique can be used to implement many of the kinds of analysis that might be thought to require set valued features, although at a small cost of adding some extra features and values to a grammar. It can also be combined with the preceding treatment of linear precedence to enforce a partial ordering on members of the set.

Set valued features are often used in conjunction with a membership test. It is usually possible to achieve the same effect by inventing a new `bool_comb_value` feature and using disjunction. For example, if our original features involved sets whose possible members were `{a b c d e f}` and had feature specifications of the form `f=X, where member(X,{b c d e})`, then the same effect can be achieved by declaring `f` to be a `bool_comb_value` feature with values in `{a,b,c,d,e,f}` and writing `f=(b;c;d;e)`. In the case that the members of the set in question are categories, then some new atomic feature values have to be invented to represent these, as is often necessary in other contexts also (Engelkamp, Erbach, and Uskoreit 1992).

9. Reducing Lexical Disjunction

This section describes two techniques for eliminating multiple lexical entries for the same word. Having multiple lexical entries for the same word is a form of disjunction, and all forms of disjunction entail increased nondeterminism leading to inefficiency in analysis. It is therefore a good idea to eliminate multiple entries as far as is possible.

9.1 Selectors

A frequently occurring case is the following: a particular word, W, has multiple possible realizations of some property, $P_1 \ldots P_n$. Which particular realization is found will

depend on the context: in context C_1 we find P_1, and, more generally, in C_i we will find P_i.

A simple though rather artificial illustration of this phenomenon might be a treatment of the semantics of prepositions that regarded them as ambiguous between different senses, depending on which type of NP they combined with. For example, we might regard *for* as having these meanings:

'for_benefactive' with animate NP: The book is for John

'for_time_period' with temporal NPs: He stayed for an hour

'for_directional' with locative NPs: They changed direction for the coast

Here the P_i are the different meanings, and the C_i are the different types of NP.

The simplest way to achieve the desired result is to have multiple entries for the preposition, one for each sense. We then treat the correlation of the properties with the contexts as a kind of agreement, between some feature on the preposition and one on the NP. Some sample lexical entries, and a rule for combining a P and an NP to make a PP might look like this:

```
p:{lex=for, sem=for_benefactive, type=animate}
p:{lex=for, sem=for_time_period, type=temporal}
p:{lex=for, sem=for_directional, type=locative}

pp:{sem=lambda(X,[S,X,NP])}  ==>  [p:{sem=S,type=T},np:{type=T,sem=NP}]
```

Unfortunately, such a treatment can lead to large numbers of lexical entries, which, especially if they are phrasal heads, as in this case, can each generate a separate parsing hypothesis for any occurrence of *for* in the input.

A better treatment can be obtained by using the fact that it is the NP that determines the P semantics, and encoding this dependency directly. What we need to do is to make the NP *select* the appropriate prepositional semantics, representing all the choices within a single lexical entry. We can do this in the following way:

1. Encode the set of possible semantic values for the preposition as a list or a tuple, where each position in the tuple is going to correspond systematically to a particular type of NP.

   ```
   p:{lex=for,semvalues=(for_benefactive,for_time_period...)...}
   ```

2. Use the original `sem` feature to represent the semantic value that will be chosen when the P is combined with an NP:

   ```
   p:{lex=for,sem=Chosen,
      semvalues=(for_benefactive,for_time_period,for_directional)...}
   ```

3. Associate with each different type of NP and other relevant categories a `selector` feature whose value will pick the appropriate member of the tuple on the P. Some illustrative rules and entries are:

   ```
   np:{type=T,selector=S,...} ==> [det:{...},n:{type=T,selector=S,...}]

   n:{lex=john,type=animate,selector=(X,(X,_,_))}
   n:{lex=week,type=temporal,selector=(X,(_,X,_))}
   n:{lex=coast,type=locative,selector=(X,(_,_,X))}
   ```

4. Now on the rule that combines a P and an NP to form a PP, use the selector feature to choose the appropriate semantics for the P:

```
pp:{sem=lambda(X,[Chosen,X,NP])} ==>
    [p:{sem=Chosen,semvalues=Tuple},
     np:{sem=NP,selector=(Chosen,Tuple)}]
```

The value of the sem feature on the P will be the first, second, or third member of the tuple, depending on the type of the NP. This will arise because the selector on the NP will unify the Chosen variable with the position on the tuple identified by its shared variable, X.

This simple device enables us to have a single entry for each preposition, while still allowing for it to have multiple senses conditional upon the type of NP it combines with. The technique has a wide variety of applications and can be astonishingly effective in reducing the number of explicit alternative entries or rules that need to be written, at the cost of a few extra features that cost nothing in terms of processing time.

9.2 Implementation

As with many of the techniques described here, implementation by way of a compiled out notation can be complex if the features involved interact with other aspects of linguistic description. If we assume that they do not (which can usually be enforced by defining a new "shadow" feature that simply duplicates the information where it is needed) then an attractive and clean way of implementing this technique is as a conditional constraint on feature values.

There are various notations one could employ: one possibility for the above example is the following, where psem and type are assumed not to figure in any other such statement, and where their total range of values is given by the conditionals (such restrictions could be relaxed to some extent given agreed conventions or extra declarations):

```
pp:{sem=lambda(X,[PSem,X,NP])} ==>
   [p:{psem=PSem},
    np:{sem=NP,type=Type)}]
 where
   if Type = animate  then PSem = for_benefactive
   if Type = temporal then PSem = for_time_period
   if Type = locative then PSem = for_directional
```

Now the compiler has enough information to be able to proceed automatically:

1. construct a values feature whose value will be a tuple of the values of psem in a canonical order. Put this feature specification on the P category. More generally, put this specification on that category of the rule introducing the conditional constraints which contains the feature specification figuring in the consequents of the conditional constraints.

2. construct a selector feature whose values will be of the form (X,(...,X,...)) where the second member of the tuple is a tuple of the same length as that in the values feature. On each category where a type feature specification is present, add the selector feature also. If the type feature is instantiated, then the selector feature will be of the form indicated by the conditional constraint: that is, the X in the second component of the tuple will be in the position corresponding to the value

of the psem feature given by the conditional, and all other positions will be anonymous variables. If the type feature is simply passed from one category to another, as it is for example on the NP rule given earlier, then the selector feature must likewise be coindexed on the two categories.

3. On the categories of the rule introducing the constraint, coindex the feature specifications as follows:

```
values   = Values
selector = (Selected,Values)
psem     = Selected
```

Again, macros can be used to make it possible to express all this economically.

9.3 Subcategorization

Perhaps the most obvious source of lexical disjunction is subcategorization. Most verbs can appear with several different types of complement, and some verbs appear with many. For example, the verb *send* in English can occur in at least the following configurations (there is some dialect variation here, but please bear with me for the sake of the example):

```
John sent a letter.

John sent a letter to Mary.

John sent Mary a letter.

John sent out a letter.

John sent a letter out.

John sent out a letter to Mary.

John sent a letter out to Mary

John sent Mary out a letter.

John sent out Mary a letter.
```

There are nine distinct configurations here. Let us ignore the fact that some alternations might be capturable by rule, and let us also ignore the fact that different semantic properties might be involved. Given this, it would be nice to be able to have a single entry for the verb *send* that encapsulated all these alternatives, rather than listing them all as separate lexical entries, as is done in all grammatical formalisms I am familiar with (except of course those that allow explicit disjunction).

In a GPSG-like approach to subcategorization (Gazdan et al. 1985), each distinct type of complement has a separate rule. Thus we will have rules, schematically, like:

```
vp ->  v[1] np

vp ->  v[2] np pp

vp ->  v[3] np np

etc.
```

Using the technique described earlier for encoding Boolean combinations of feature values, we could achieve the desired single entry for *send* very simply. Rather than

use numbers to represent the different subcategorization possibilities, we will have an atom with some mnemonic content: {np,np_pp,np_np,np_pnp,...}. Then we define a feature that can take as values Boolean combinations from this set, subcat, and write:

```
v:{lex=send,subcat=(np;np_pp;np_np; . . . ) . . . }
```

The various VP rules are recast using the mnemonic symbols:

```
vp:{} ==> [v:{subcat=np},np:{}]
vp:{} ==> [v:{subcat=np_pp},np:{},pp:{}]
etc.
```

Now one entry for each verb will subsume all the possible subcategorization combinations for it.

This technique certainly reduces the number of items in the lexicon. Unfortunately, it does not necessarily reduce the amount of nondeterminism during analysis. Although there is only a single entry for *send*, it will, on either a left-corner or head-driven approach to parsing, initiate parsing hypotheses for each distinct VP rule whose head unifies with it. That will be exactly the same number of parsing hypotheses as we would have had with the original GPSG treatment, and so there is no obvious advantage here.

Nevertheless, this technique should not be scorned, for in other cases, there will be some advantage gained. For example, in derivational morphology the presence of multiple entries for verbs like *send* can cause unwanted ambiguity. The word *sender*, for example, would be nine ways ambiguous, given a rule like:

```
n:{} ==> [v:{}, affix:{lex=er}]
```

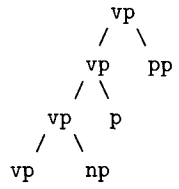
With just one entry for *send* this problem goes away. (Note that one cannot get round the ambiguity problem by just restricting the agent nominalization rule to one or two types of subcategorization: many different types of verbal complement may be involved: *sleeper*, *designer*, *thinker*, etc.)

As we have seen, the GPSG treatment of subcategorization involves many VP rules. A currently more favored approach is to use a single VP rule or schema or subcat principle, and a list of categories subcategorized for by a verb:

```
vp:{subcat=Rest} ==> [vp:{subcat=[Next|Rest]},Next]

vp:{lex=send,subcat=[{cat=np},{cat=p},{cat=pp}]}
etc.
```

Multiple applications of this schema use up subcategorized elements one at a time, with a requirement that when the VP is combined with a subject to form a sentence the subcat list is empty (or contains just one category unifiable with the subject, depending on the approach taken). The tree for a VP will look like:



This approach requires multiple entries for verbs, but has the advantage that it eliminates the need for different VP rules for each type of complement.

It would be nice to find some way of combining this single-schema approach with a single subcategorization entry subsuming multiple possibilities. This would eliminate

nondeterminism completely, even for verbs capable of appearing with many different types of complement. Although the details are rather complex, it turns out that it is possible to achieve this by combining the Boolean encoding technique in conjunction with the use of `selectors` as previously described. Unfortunately, there are some limitations on the amount of subcategorization information that can be expressed by the resulting technique: in particular, categories have to be represented by atoms, which is an inconvenient limitation. Nevertheless, for many purposes where efficiency of processing is at a premium, it could be worth living with this limitation.

First of all, consider how to represent the various subcategorization possibilities of a verb like *send*, using Boolean combinations of atoms. (I have omitted as many parentheses as possible in the interests of readability. Assume that ; takes precedence over & unless parentheses indicate otherwise.) It might seem that something like:

```
{cat=vp,lex=send,
 subcat=(np;
         np & pp;
         np & np;
         p & np;
         np & p;
         p & np & pp;
         np & p & pp;
         np & p & np;
         p & np & np)
}
```

would accurately describe the possibilities. (This Boolean expression can, of course, be written more compactly by using a few more disjunctions).

The VP schema that we need will then have to be of the following form. Note that since we need to be able to generalize over categories, we are reverting to the basic (untyped) category notation.

```
 schema:
   {cat=vp,subcat=S} ==> [{cat=vp,subcat=S]},{cat=S}]
 sample entry:
   {cat=vp,lex=send,subcat=(np; np & pp;  np & np;...)}
```

(Note that this makes `cat` a Boolean combination feature. Given the importance of the `cat` feature for efficient indexing and lookup this might be, for practical purposes, unwise. A better implementation would use a new feature).

A moment's reflection should reveal that this first attempt will not give the correct results, for two reasons. Firstly, the various different orderings are not properly encoded here (because `p & q` is logically equivalent to `q & p`). Secondly, there is no encoding of the requirement to find the correct number of subcategorized entities (because `p & p` is logically equivalent to `p`). Thus nothing would prevent us from successfully analyzing a sentence like *John sent Mary out Mary a letter to Mary a letter*, with too many complements, or *John sent out*, with too few.

Let us tackle the ordering problem first. We can solve this by adding new symbols representing the product of the set of relevant categories `np,p,pp` and the set of positions `1,2,3` after the verb in which they occur. We then define a Boolean feature value type for the feature `subcat` as follows:

```
{np1,p1,pp1}*{np2,p2,pp2}*{np3,p3,pp3}
```

We encode the subcategorization possibilities in the obvious way, using these new symbols. (This time I have used disjunction to give a more compact encoding.)

```
{lex=send,cat=vp,
 subcat=(np1;
         np1 & (pp2 ; np2) ;
         p1 & np2;
         np1 & p2;
         np1 & p2 & (pp3 ; np3);
         p1 & np2 & (pp3 ; np3))
 }
```

We will define a new feature that appears on every category that can be subcategorized for, say `scat`, whose values are tuples. An NP, for example, will have `scat=(np1,np2, np3)`. Notice that the components of the tuple are values that can appear in Boolean combinations, for they must be of the same type as the `subcat` feature. In order to pick the correct value for the position in question, we associate with the verb a feature whose value is a list of the constructs called `selectors` that we used earlier. Each `selector` picks out a position in the complement corresponding to the position of the selector in the list: the first `selector` on the list will pick out `np1` for an NP, `pp1` for a PP; the second will pick out `np2` for an NP, `pp2` for a PP, and so on. The feature and value will be of the form: `selectors=[(A,(A,_,_)),(B,(_,B,_)),(C,(_,_,C))]`.

The VP rule schema now uses the current `selector` to choose the appropriate symbol from the complement it is combining with. It pops `selectors` off the list each time it applies so that the correct positional encoding is available for the next application.

```
{cat=vp,subcat=S,selectors=Rest} ==>
     [{cat=vp,subcat=S,selectors=[(S,X)|Rest]},
      {cat=_,scat=X}]

{cat=np,scat=(np1,np2,np3),...}
{cat=pp,scat=(pp1,pp2,pp3),...}
etc.
{lex=send,cat=vp,subcat=(np1; np1 & pp2;  np1 & np2; etc.),
   selectors=[(A,(A,_,_)),(B,(_,B,_)),(C,(_,_,C))]}
```

Since the `selector` list guarantees that the symbols `np1` etc. are only found in the correct position after the verb, this solves the ordering problem. Although `np1 & np2` is logically equivalent to `np2 & np1`, the `selector` list will not allow the second ordering to be found, because this would involve an attempt to unify `np1` with `np2`. The use of `selectors` to encode position also solves some cases of the problem that our first attempt suffered from, of allowing more than the correct number of complements. The `selector` list will not allow more than three complements of *send* to be found. Unfortunately, the treatment so far will still allow *fewer* than three complements to be found even where another is needed for the sequence to be grammatical. For example, the sentence *John sent out* will be parsed successfully (as indeed will *John sent*) because no *conflict* with any of the subcategorization possibilities has been encountered. The way the Boolean encoding works has to allow for elements to be conjoined one at a time, but it cannot require that all the elements are present simultaneously, for this very reason.

The way to solve this problem is to expand our Boolean combination of `subcat` feature values to include some special `finish` symbols.

```
{np1,p1,pp1}*{np2,p2,pp2}*{np3,p3,pp3}*{f1,f2,f3,f4}
```

There is one symbol for each possible `subcat` position, plus an extra one to mark the end of the list. We have to extend our various `selectors` and the lists they appear in to accommodate this fourth position.

The intuitive motivation behind this move is to regard the completion of a subcat requirement as being signaled by a special finish category. However, that category need not actually be present: its marker can instead be introduced by the rule that combines a completed VP with a subject to make a sentence. (This is analogous with the treatment described earlier in which this rule required a subcat list to be empty at this point).

To implement this analysis, we enter into the various subcategorization possibilities the information about which position marks their completion:

```
{lex=send,cat=vp,
  selectors=[(A,(A,_,_,_)),      % each selector now has 4 positions
             (B,(_,B,_,_)),
             (C,(_,_,C,_)),
             (D,(_,_,_,D))],     % and there are 4 selectors in the list

  subcat=(np1 & f2;
          np1 & (pp2 ; np2) & f3;
          p1 & np2 & f3;
          np1 & p2 & f3;
          np1 & p2 & (pp3 ; np3) & f4;
          p1 & np2 & (pp3 ; np3) & f4)
}
```

An intransitive verb would of course just be subcat=f1.

Our VP rule schema is exactly as before. For the right results to be obtained, however, we now need to assume the presence of some rule like the following to close off the subcategorization:

```
{cat=s}  ==> [{cat=np},
              {cat=vp,subcat=S,selectors=[(S,(f1,f2,f3,f4))|_]}]
```

This will add the finish marker of the appropriate position to the subcat value of the VP. This unification will only succeed if the verb is subcategorized to finish at that point, and we will not have reached this point unless all the other elements subcategorized for have been found in the correct order. So, using selectors and Boolean combinations of feature values together we have developed an analysis that completely eliminates disjunction and hence non-determinism. It will, of course, generalize to any other area having the same structural properties.

9.4 Implementation

The general features of the implementation of this technique are as follows.

1. we need to know the subcategorized-for categories, the symbols used to identify them, and the maximum number that can occur in a single verb-complement construction. This might conveniently be stated by a declaration something like:

   ```
   subcategorization_feature(Name,Categories,Mnemonics,MaximumLength).
   ```

   This will allow us to automatically construct the selector list. For a maximum number of four, this will take the form:

   ```
   [(A,(A,_,_,_)),(B,(_,B,_,_)),(C,(_,_,C,_)),(D,(_,_,_,D))]
   ```

   The declaration will also allow us to work out the mnemonic values (np1,f1, etc) needed for the scat and subcat features (types of

bool_comb_value feature). Rules or lexical entries that build a member of Categories must have the scat feature added with tuple values like (np1,np2,np3) etc.

On lexical entries for subcategorizers, the subcat value can be stated as a simple list of possibilities:

```
subcat= [[np],
         [np,pp],
          etc.
```

or some convenient abbreviatory notation could be devised. These values should then be compiled to Boolean combinations of the corresponding mnemonic atoms. The tuple-valued selectors feature needs to be added to these entries with the value already illustrated: this can be done automatically, given the declaration.

2. The rule that encodes the combination of subcategorizer and subcategorized has to be identified, and feature specifications of the following form added:

```
{...subcat=S, selectors=Rest,...} ==>
  [{(subcategorizer) subcat=S, selectors=[(S,X)|Rest]...},
   {(subcategorized) cat=X}]
```

3. The rule that closes off the subcategorization needs to have the relevant selectors value added, as in the example above.

With suitable generation of macros by the compiler, our example might then be written by the grammarian as:

```
% declaration:
subcategorization_feature(subcat,[{cat=np},{cat=p},{cat=pp}],[np,p,pp],4).
% subcat schema rule:
 {cat=vp, Mother} ==>
     [{cat=vp, Subcategorizer]},
      {Subcategorized}]
  where
   subcat_schema_macro(Mother,Subcategorizer,Subcategorized).
% rule terminating subcat:
 {cat=s}  ==> [{cat=np},
               {cat=vp,CloseSubcat}]
  where
   subcat_close_macro(CloseSubcat).
% sample entries:
 {lex=send,cat=vp,subcat=[[np], [np,pp], [np,np] etc.]}
 {lex=give, cat=vp, subcat=[[np,np],[np,pp], etc.]}
```

9.5 Limitations

As mentioned earlier, there are some limitations associated with this technique. Because of the type of Boolean mechanism we are using, we are restricted to atomic symbols to represent the subcategorized-for elements. Putting into lexical entries the kind of refined subcategorization information that we often do using features is not possible, or at least not possible without expanding the vocabulary of symbols like np1, pp2, etc. to induce a finer partition among instances of the categories in question. This is, of course, a serious limitation, especially for theories of grammar that are largely lexically based. However, where all the categories that figure in subcategorization will

have some of the same features (e.g., those used for threading gaps) then these can be incorporated directly into, in our case, the VP subcategorization schema rule.

Another problem is that since we need to be able to generalize over whole categories, we cannot, as things stand, use compilation into terms for feature structures. One way round this is to change the VP schema so that complements are characterized not just by a variable but by an explicit new category, say `xcomp`, with a `bool_comb_value` feature on it that can serve to identify categories. We then introduce rules expanding `xcomp` as the "real" category corresponding to that feature. This may in turn re-introduce some inefficiency, since there will be an extra level of structure that is not linguistically motivated.

A final limitation, which is perhaps more theoretically defensible, is that we are forced to be absolutely and strictly compositional in assembling the semantics of verb phrases grouped under the same subcategorization treatment. Since we have only one entry for a verb, then any semantic differences that are associated with variant subcategorizations will have to be built from the complement constituents in a completely compositional way.

Alternatively, as is done in many wide-coverage systems for efficiency reasons, syntactic and semantic analysis can be separated into consecutive stages. This can have a further advantage in that now the same technique can be used to eliminate disjunction for words where there is sense ambiguity but no syntactic ambiguity. If different lexical entries are assigned to the content words in the following sentence because they differ semantically but not syntactically, then the sentence will have 16 parses (8 * 2 for the attachment ambiguity) to be disambiguated.

```
They saw the ball near the bank.

(saw  = see, or cut wood;
 ball = round thing, or dance;
 bank = edge of river, or financial institution).
```

If sense selection is instead performed when syntactic processing is completed, on the assumption that the words involved do not differ syntactically, then there will only be two parses and three lexical disambiguation decisions. In general we will only be dealing with the sum and not the product of the syntactic and semantic ambiguity. Under such a processing regime the appropriate sense entry for a verb on a particular subcategorization can be simply and cheaply selected (since the complete complement will be there), and the benefits of the preceding analysis for syntactic processing will be retained.

Acknowledgments

The work reported here was supported at SRI Cambridge under contracts with the European Commission, DGXIIIb, Luxembourg (ET6/1, and ET10), and at Cambridge University Computer Laboratory by a grant from the Joint Research Councils Cognitive Science Initiative, held jointly by the author and William Marslen-Wilson (London University), 'Unification based models of lexical access and incremental processing,' SPG 893168. Earlier versions of some of the material included here can be found in the final reports of the ET6/1 (Alshawi et al. 1991) and ET10 projects. (A version of the latter appeared as Markantonatou and Sadler 1994).

This paper was largely written while I was a visitor at the Institut für Maschinelle Sprachverarbeitung, University of Stuttgart, in 1993. I am grateful to the members of the Institute, in particular to Josef van Genabith, Hans Kamp, Esther König, Christian Rohrer, and Sybille Laderer for their help and hospitality during my stay.

For comments on earlier versions of this work I am grateful to Doug Arnold, Gerald

Gazdar, Josef van Genabith, Louisa Sadler, and David Sedlock. Comments from three anonymous referees also helped greatly to improve both content and presentation of the final version.

References

Abramson, Harvey. 1988. Metarules and an Approach to Conjunction in Definite Clause Translation Grammars. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Logic Programming Conference*. MIT Press, pages 233–248.

Ait-Kaci, Hassan. 1986. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351.

Ait-Kaci, Hassan, Bob Boyer, Pat Lincoln, and Roger Nasr. 1989. Efficient implementation of lattice operations. *ACM Transactions of Programming Languages and Systems*. 11(1), January.

Alshawi, Hiyan (editor). 1992. *The Core Language Engine*. MIT Press: Bradford Books, Boston, MA.

Alshawi, Hiyan, Doug Arnold, Rolf Backofen, David Carter, Jeremy Lindop, Klaus Netter, Stephen Pulman, Junichi Tsujii, and Hans Uskoreit. 1991. *ET6/1 Rule Formalism and Virtual Machine Design Study*. CEC Luxembourg.

Arnold, Doug, Stephen Krauwer, Michael A. Rosner, Louis des Tombe, and Giovanni B. Varile. 1986. The <C,A>T framework in EUROTRA: A theoretically committed notation for MT. *COLING-86*, pages 297–303.

Bobrow, Rusty, Robert Ingria, and David Stallard. 1991. The mapping unit approach to subcategorization. In *Proceedings of Darpa Speech and Natural Language Workshop*. Palo Alto. Morgan Kaufman.

Briscoe, Edward, Claire Grover, Branimir Boguraev, and John Carroll. 1987. A formalism and environment for the development of a large grammar of english. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 703–708, Milan, Italy.

Carpenter, Bob. 1992. The logic of typed feature structures. *Cambridge Tracts in Theoretical Computer Science*. CUP.

Clark, Keith and Frank McCabe. 1984. *Micro-Prolog*. Prentice Hall.

Engelkamp, Judith, Gregor Erbach, and Hans Uskoreit. 1992. Handling linear precedence constraints by unification. In *Proceedings of the 30th Annual Meeting, ACL*. pages 201–208, Newark, Delaware.

Gazdar, Gerald, Ewan Klein, Geoffrey K. Pullum and Ivan A. Sag. 1985. *Generalized Phrase Structure Grammar*. Blackwell Publishing, Oxford.

Haas, Andrew. 1989. A parsing algorithm for unification grammar. *Computational Linguistics*. 15(4): 219–232.

Johnson, Mark. 1988. *Attribute-Value Logic and the Theory of Grammar*. CLSI Lecture Notes, Vol. 16. University of Chicago Press.

Karttunen, Lauri. 1986. D-PATR: A development environment for unification-based grammars. In *Proceedings of the 11th International Conference on Computational Linguistics*, pages 74–80, Bonn.

Markantonatou, Stella and Louisa Sadler (editors). 1994. *Grammatical Formalisms: Issues in Migration and Expressivity*. Studies in Machine Translation and Natural Language Processing, Vol 4. Luxembourg: Office for Official Publications of the Commission of the European Communities.

Mellish, Chris. 1988. Implementing systemic classification by unification. *Computational Linguistics*, 14: 40–51.

van Noord, Gertjan, Joke Dorrepaal, Pim van der Eijk, M. Florenza, and Louis des Tombe. 1990. The MiMo2 research system. Third International Conference on Theoretical and Methodological Issues in Machine Translation, Linguistics Research Center, Austin, Texas.

Pereira, Fernando. 1981. Extraposition grammars. *Computational Linguistics*, 7(4): 243–256.

Pereira, Fernando and Stuart Sheiber. 1987. *Prolog and Natural Language Analysis*. CSLI Lecture Notes, Vol. 10, University of Chicago Press.

Pollard, Carl and Ivan Sag. 1987. *Information Based Syntax and Semantics, 1: Fundamentals*. CSLI Lecture Notes, Vol. 13. University of Chicago Press.

Pollard, Carl and Ivan Sag. 1993. *Head Driven Phrase Structure Grammar*. University of Chicago Press.

Pulman, Stephen G. 1992. Unification based syntactic analysis. In Hiyan Alshawi, editor, *The Core Language Engine*, MIT Press, 1992.

Pulman, Stephen G. 1994. Expressivity of Lean Formalisms. In S. Markantonatou and L. Sadler, editors, *Grammatical Formalisms*, Luxembourg, 1994, pages 35–59.

Ramsay, Allan. 1990. Disjunction without

tears. *Computational Linguistics*, 16(3): 171–174.
Shieber, Stuart M. 1984. The design of a computer language for linguistic information. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 362–366, Stanford, CA.
Shieber, Stuart M. 1986. *An Introduction to Unification-Based Approaches to Grammar*. University of Chicago Press.
Shieber, Stuart M. 1988. A Uniform Architecture for Parsing and Generation. *Proceedings of the 12th International Conference on Computational Linguistics*, Budapest.
Shieber, Stuart, Gertjan van Noord, Robert C. Moore, and Fernando C. N. Pereira. 1990. Semantic-head-driven Generation. *Computational Linguistics*.
Smolka, Gert. 1992. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12: 51–87.