EXTENDED GRAPH UNIFICATION

Allan Ramsay
School of Cognitive Sciences
University of Sussex, Falmer BN1 9QN

Abstract

We propose an apparently minor extension to Kay's (1985) notation for describing directed acyclic graphs (DAGs). The proposed notation permits concise descriptions of phenomena which would otherwise be difficult to describe, without incurring significant extra computational overheads in the process of unification. We illustrate the notation with examples from a categorial description of a fragment of English, and discuss the computational properties of unification of DAGs specified in this way.

1 INTRODUCTION

Much recent work on specifying grammars for fragments of natural languages, and on producing computational systems which make use of these grammars, has used partial descriptions of complex feature structures (Gazdar 1988). Grammars are specified in terms of partial descriptions of syntactic structures; programs that depend on these grammars perform some variant of unification in order to investigate the relationship between specific strings of words and the syntactic structures permitted by the grammar—is some sentence grammatical, what actually is its syntactic structure, how can some partially specified structure be realised as a string of words, and so on. Nearly all existing *unification grammars* of this kind use either *term unification* (the kind of unification used in resolution theorem provers, and hence provided as a primitive in PROLOG) or some version of the *graph unification* proposed by Kay (1985) and Shieber (1984). We propose an extension to the languages used by Kay and Shieber for describing graphs, and to the specification of the conditions under which graphs unify. This extension enables us to write concise descriptions of syntactic phenomena which would be awkward to specify using the original notations. We do not argue that our extension makes it possible to describe any phenomena which could not have been described at all using the existing notations, just that the descriptions using the extension are more concise.

2 GRAPH SPECIFICATIO

We start by defining a language GSL (*graph specification language*) for describing graphs, and by specifying the conditions under which two graphs unify.

2.1 GSL: syntax

The syntax of GSL has been kept as close as possible to that of FUG (Kay 1985) in order to facilitate comparisons. It is not, unfortunately, possible to keep it close to both FUG and PATR (Shieber 1984), but it should be possible for readers familiar with PATR to see roughly what the relation between the two is.

A *node descriptor* consists of either an atomic symbol, e.g. *agr, cat, bar*, or of two atomic symbols separated by a slash, e.g. *cat/C head/OBJECT*. In the first case the symbol is the *value* of the described node, in the second the symbol before the slash is the node's value and the symbol after it is its *name*. We will generally use lower case words for values and upper case ones for names, but the distinction between upper and lower case has no significance in GSL.

A *path descriptor* consists of a sequence of node descriptors separated by equals signs, e.g. *head=major=cat=prep*. The path described by such a descriptor consists of the sequence of described nodes. The first node in a path is called its *initial node* and the final node is called its *terminal node*. The descriptor of the terminal node in a path may be followed by an exclamation mark

as in *head=major=cat=prep!*, in which case the node is said to be *mandatory*.

A *graph descriptor* consists of a set of path descriptors separated by commas. The graph consists of the set of described paths. If two node descriptors in a graph descriptor specify the same name, they refer to the same node.

A set of paths with identical initial segments may be specified by writing the initial segment just once and including the divergent tails within nested brackets, so that

A=B=C=(X=Y, W=(V=U, Q=R))

is a shorthand form for:

A=B=C=X=Y,
A=B=C=W=V=U,
A=B=C=W=Q=R

The *sub-graph governed by a path* is the set of all terminal sequences of paths whose initial sequence matches the given path. The last node in the given path is called the *root* of the sub-graph governed by the path. Thus in the above example the set of paths *X=Y, W=V=U, W=Q=R* is the sub-graph governed by the path *A=B=C*, and *C* is the root of this sub-graph.

A *macro* is simply a symbol which has been specified as a shorthand for some other sequence of symbols. Macros are expanded by simple textual substitution, so that if *NP* were a macro for the sequence of symbols *cat=n, bar=two* then *head=(NP)* expands to *head=(cat=n!, bar=two!)*. The parentheses are important—*head=NP* expands to *head=cat=n!, bar=two!*, which is very different from *head=(cat=n!, bar=two!)*.

The major differences between GSL and the languages used by Kay and Shieber are that GSL distinguishes between optional and mandatory nodes, and that names (which function as the constraints for turning trees into graphs) can be attached to non-terminal nodes. GSL also differs from FUG in that it does not provide a facility for disjunctive graphs—disjunction is catered for by requiring the grammar and lexicon to contain explicit alternatives, rather than by permitting graphs themselves to contain options. Most of the other differences are cosmetic—the GSL path *agr=num=sing!* is equivalent to the PATR path *[agr: [num: sing]]* and the FUG descriptor *agr=num=sing*. The GSL path *agr=num=sing* is roughly equivalent to the PATR path *[agr: [num: [sing: <Alpha>]]]* and the FUG descriptor *agr=num=sing=ANY*. The fact that the second set of paths are only "roughly" equivalent is a consequence of the new definition of unification given in the next section.

2.2 GSL: unification

The major operation that we are going to perform on graphs specified in GSL is unification. We define this, as usual, in terms of the common extension of sets of graphs. We start by defining the common extension of a pair of graphs. Two graphs G1 and G2 *unify* to produce a *common extension* E under the following conditions:

(i) Suppose V is the value of initial nodes in each of G1 and G2. Then the sub-graphs of G1 and G2 which are governed by the path consisting of just the node V must have a common extension, say E_V. If they do have such a common extension, then the common extension E of G1 and G2 themselves must include all the paths obtained by adding V to the front of members of E_V. If they do not then G1 and G2 do not unify, and hence have no common extension.

Furthermore, if any initial node in either graph with V as its value has a name, that name must be associated with a sub-graph which has a common extension with each of G1 and G2. All the paths which appear in any of these extensions must also be included in E. Again if the sub-graph associated with any such name fails to have a common extension with either G1 or G2 then G1 and G2 themselves do not unify.

(ii) Suppose V appears as the value of one or more initial nodes in G1 but of none in G2. Then if V is a mandatory terminal node of any path in G1 of which it is the initial node then G1 and G2 do not have a common extension (since V is mandatory in G1, but does not appear as an initial node of any path in G2). Otherwise the common extension of G1 and G2, if it exists, must include all the paths in G1 for which V is an initial node. The same condition applies if V is the value of one or more initial nodes in G2 but of none in G1.

(iii) The common extension of G1 and G2 contains no paths not explicitly required by conditions (i) and (ii).

The common extension of a set of graphs {G1, G2, ..., Gn} where n > 2 is simply the common extension of G1 with the common extension of the set {G2, ..., Gn}.

This definition of the common extension of

a set of graphs is rather non-constructive, and is neutral with respect to computational mechanisms. We need to show that we can in fact compute common extensions, and to consider the complexity of the algorithm for doing so, but before that we ought to try to show that we can use GSL to give concise descriptions of syntactic rules. If we can't do that, there is no point in worrying about the efficiency of algorithms for comparing graphs described in GSL at all.

3 SYNTACTIC DESCRIPTIONS USING GSL

We will illustrate the use of GSL with elements of a categorial grammar for a fragment of English. GSL is not specifically designed for categorial grammar, but the complexity of the category structures of any non-trivial categorial grammar means that such grammars provide a good testbed for notations for describing categories. Although categorial grammars have recently received considerable attention (Pareschi & Steedman (1987), Klein & van Benthem (1987), Oehrle, Bach & Wheeler (1987)), computational treatments have been hindered by the need to develop and manipulate large category descriptions. The expressive power of GSL is therefore well illustrated by the ease with which we can develop the category descriptions required for a non-trivial categorial grammar.

We start with the basic categorial rules:

```
(major/X, minor/Y, subcat/SUB, slash/SLASH)
⟶
(HEAD=(major/X, minor/Y, subcat/SUB,
                    slash/SLASH),
 RSLASH=(major/X1, minor/Y1, subcat/SUB1,
                    slash/SLASH),
 slash=null!),
(major/X1, minor/Y1, subcat/SUB1,
 slash/SLASH)

(major/X, minor/Y, subcat/SUB, slash/SLASH)
⟶
(major/X1, minor/Y1, subcat/SUB1,
 slash=null!)
(HEAD=(major/X, minor/Y, subcat/SUB,
                    slash/SLASH),
 LSLASH=(major/X1, minor/Y1, subcat/SUB1,
                    slash/SLASH),
 slash/SLASH)
```

The first of these is an extended version of the normal categorial rule for combining something which requires an argument to its right with an argument of the appropriate type, namely:

A ⟶ A/B B

We have been forced to complicate this rule, as have others trying to produce categorial grammars for non-trivial fragments, in order to take into account intrinsic syntactic functions such as case and number agreement, and to deal with the fine details of sub-categorisation rules. In our extended version of the basic rule, the *A* of the basic version is replaced by *(major/X, minor/Y, subcat/SUB, slash/SLASH)* and the *B* of the basic version by *(major/X1, minor/Y1, subcat/SUB1, slash/SLASH)*. The *major* features of a category are simply its main category (*noun, verb, preposition, conj*) and its bar level (*zero, one, two*). The *minor* features are the intrinsic syntactic features such as *agr* and *aux*. *subcat* specifies what arguments (*lslash* and *rslash*) are required and what the head (*head*) of the local tree described by the rule is like. *slash*, as usual in unification grammars, carries information about unbounded dependencies. The category *A/B* of the basic rule is replaced by:

```
(HEAD=(major/X, minor/Y,subcat/SUB,
                              slash/SLASH),
 RSLASH=(major/X1, minor/Y1, subcat/SUB1,
                              slash/SLASH),
 slash=null!)
```

This describes a structure which will join with a *(major/X, minor/Y, subcat/SUB, slash/SLASH)* to its right to make a *(major/X1, minor/Y1, subcat/SUB1, slash/SLASH)*.

We have made very little use of the extra facilities provided by GSL in specifying this rule, beyond the convenience of the abbreviations *HEAD* for *subcat=head* and *RSLASH* for *subcat=rslash*. Apart from that, we have used names for specifying constraints, but that could easily have been done in any of the standard formalisms; and we have used the exclamation mark to constrain the value of *slash* on the first element of the right hand side to be *null*. The second of the basic rules is sufficiently similar that it requires no further discussion.

To show how the extra power of GSL can help us construct concise descriptions, we will consider two specific examples. The first is the definition

of the lexical entry for an auxiliary. This requires the following three macro definitions:

```
VP ⇝ (V, I, minor/X=vform=agr/AGR,
          RSLASH=null!,
          HEAD=(S, minor/X),
          LSLASH=minor=agr/AGR)
VERB ⇝ (V, O, minor/X, LSLASH=null!,
          HEAD=(VP, minor/X))
AUX ⇝ (VERB, minor=aux=yes!,
          RSLASH=(VP, LSLASH/SUBJ),
          HEAD=LSLASH/SUBJ)
```

The definition of *AUX* says that it is a special type of *VERB*, namely one that will combine with a *VP* to its right. The *head* of the *AUX* inherits any constraints on the subject of its own *rslash*. The definition of *VERB* says that it is something which does not require anything to its left, and that it will participate in local trees dominated by objects of type *VP*, with the constraint that the *VERB* has the same *minor* features as the *VP*. The definition of *VP* is fairly similar, but it does make use of the facility for placing names in non-terminal positions to enforce two constraints—one between the entire set of *minor* features of the *VP* and the *minor* features of its *head*, and another between the *agr* features of the *VP* and the *agr* features of its subject.

Although this set of abbreviations appears only to call upon the facility for including names for non-terminal nodes once, we can see that if we were to expand the macros inside the definition of *AUX* there would be two other places where this was done (the definition below still has some macros unexpanded to help keep it readable):

```
AUX ⇝ (V, O,
        minor/X=aux=yes!,
        LSLASH=null!,
        H=(V, I,
            minor/X=vform=agr/AGR,
            RSLASH=null!,
        H=(S,minor/X),
        LSLASH/SUBJ=minor=agr/AGR),
        RSLASH=(VP, LSLASH/SUBJ))
```

It is worth noting that nowhere in either the expanded definition or in the three abbreviations is the major category of the subject specified. This information may be inherited from the main verb of the *VP* argument of the auxiliary, but otherwise its major category is unconstrained, in order to permit sentences like *Eating people is going out of fashion* and *For me to eat you would be the height of impropriety.* It is assumed that the lexical entries for verbs will sub-categorise for *NP, VP* or *S* subjects as required, just as they sub-categorise for complements.

The second example of the use of GSL features comes from a group of rules which describe alternative sub-categorisation frames—rules which say, for instance, that a typical ditransitive verb has a case frame requiring two NP's rather than an NP and a PP. The rule below generates the "aux-inverted" case frame for *AUX*'s:

```
(V, O, minor=vform/VFORM=agr/AGR,
 RSLASH=(NP,minor=(SUBJ, agr/AGR),
                slash=null!),
 HEAD=(major=cat=partial!, RSLASH/A2,
 HEAD=(S, minor=(vform/VFORM,
                mood=interrogative!))))
⟶
(AUX,
 minor=(vform/VFORM=finite=tensed!),
                RSLASH/A2)
```

This rule again specifies names for non-terminal nodes, with VFORM twice being used as a name for a non-terminal node. The effect of this is to constrain the relevant item to be tensed and to share the same value for *agr* as its "inverted" subject. The rule also contains a number of mandatory features. The path *minor=vform=finite=tensed!*, for instance, restricts the rule to cases of tensed auxiliaries.

We cannot use examples to "prove" that GSL makes it possible to write more concise specifications than we could write in FUG or PATR. This is particularly clear when the examples are culled from a grammar whose overall structure imposes constraints which can only be motivated by considering the grammar as a whole (which we do not have space for), rather than by looking at the examples in isolation. The best we can hope for is that the examples do seem to describe the constructions they are aimed at fairly concisely; and perhaps that it is not all that obvious how you would describe them in PATR or FUG.

4 COMPUTATIONAL COMPLEXITY

We end by briefly considering the complexity of the task of seeing whether two graphs with named non-terminal nodes have a common extension. It is well-known that disjunctive unification is NP-complete (Kasper 1987). What is the status of unification of structures with constraints on sub-graphs?

The definition of unification given in Section 2 looks very non-deterministic—full of phrases like "Suppose V is the value of initial nodes in each of G1 and G2" and "Suppose V appears as the value of one or more initial nodes in G1 but of none in G2". We can make it much more constrained by imposing a normal form on graphs. The first thing we need for this is an arbitrary ordering on features, which we can easily find since features are just alphanumeric strings, and these can be ordered lexicographically. If we were working with trees rather than DAGS, and we had such an ordering, we could impose a normal form by ordering the sub-trees of a node by the lexicographic ordering of their own root nodes, so that the normal form of the tree

(A (X (Z Y)) (P (S R)))

would be:

(A (P (R S)) (X (Y Z)))

Unification of trees in this kind of normal form is of complexity o(M × N), where M is the maximum branching factor for the tree and N is the maximum depth. It is clear that we can impose a very similar normal form on DAGs without constraints on non-terminal nodes. For DAGs which do have constraints on non-terminal nodes, we have to split the representation of the graph into two pieces. We represent the basic structure of the graph in terms of sets of nodes and their successors; but where a node has a name, we include the name rather than the node itself. For each such named node, we store the sub-graph rooted at the node separately as the value of the name (this sub-graph itself, of course, may contain named nodes, in which case we just do the same again). We now effectively have a set of DAGs each of which has no constraints on internal nodes. We can therefore put each of these into normal form as before. The theoretical time for unification is again o(M × N), though N is now the length of the longest path through the graph you would get if you replaced names by the sub-graphs they name. The practical time is such as to make it perfectly sensible to use it as the basis of a computational system. Quoting times for analysing specific texts is a fairly meaningless way of comparing parsers, let alone unification algorithms, since there are so many unspecified parameters—size of the grammar, degree of ambiguity in the lexicon, speed of the basic machine, ... All I can say is that left-corner chart parsing with categorial rules specified via GSL descriptions of categories is markedly quicker than naive top-down left-right parsing of grammars of comparable coverage written as DCGs.

References

Gazdar G. (1987) The new grammar formalisms—a tutorial survey *IJCAI-87*

Kasper R. (1987) A unification method for disjunctive feature descriptions *ACL Proceedings, 25th Annual Meeting* 235-242

Kay M. (1985) Parsing in functional unification grammar in *Natural Language Parsing* eds. D.R. Dowty, L. Karttunen & A.M. Zwicky, Cambridge University Press, Cambridge, 251-278

Klein E. & van Benthem J. (eds) *Categories, Polymorphism, and Unification* (1987) Centre for Cognitive Science, University of Edinburgh and Institute for Language, Logic, and Information, University of Amsterdam Edinburgh and Amsterdam

Oehrle D., Bach E. & Wheeler D. (1987) *Categorial grammars and natural language structures* Reidel, Dordrecht

Pareschi R. & Steedman M.J. (1987) A lazy way to chart-parse with categorial grammars *ACL Proceedings, 25th Annual Meeting* 81-88

Shieber S.M. (1984) The design of a computer language for linguistic information *COLING-84* 362-366