1. Introduction

The Budapest Unification Grammar (BUG) system described in this paper is a system for generating natural language parsers from feature-structure based grammatical descriptions (grammars). In the current version, source grammars are limited to the context-free phrase structure grammar format. BUG compiles source grammars into automata, which it can then use for parsing input strings.

BUG was developed at the Research Institute for Linguistics (Budapest) and at the Theoretical Linguistics Program, Budapest University (ELTE) with the support of OTKA (National Funds for Research) of the Hungarian Academy of Sciences. It was written in C and is portable across Unix*, DOS and VMS.

BUG differs from other unification-based grammar-writing tools in two major respects as well as in a number of minor ways. One major difference is that BUG uses feature geometries. The feature geometry is a (recursive) definition of well-formed feature structures, which must be specified in the source grammar. The other major difference is that BUG uses a built-in performance restriction, called the string completion limit (SCL). Using the string completion limit, we can limit the generative power of a context-free grammar to regular languages. The paper focuses on these two innovations as well as a third feature of BUG, which is the separation of the structural description (SD, conditions of application) from the structural change (SC, effect of application) in source rules.

2. Feature Geometries

2.1. What Are Feature Geometries?

The term feature geometry is taken from generative phonology, where it was introduced by Clements (1985). A feature geometry determines what feature structures are allowed by specifying what (complex or atomic) values each path in a feature structure can have. In this way, a feature geometry expresses certain kinds of feature co-occurrence restrictions (FCRs, Gazdar et al., 1985), namely, those FCRs that are local in the sense that they can be formulated in terms of path continuation restrictions. For example, we can incorporate the FCR

\[ \text{TENSE = PAST} \Rightarrow \text{[FINITE]} \]

in a geometry by making TENSE a sub-feature of only FINITE (and PAST a possible value of TENSE). On the other hand, we cannot encode a global FCR like

\[ \text{SUBJ DEF = +} \Rightarrow \text{[INDIR_OBJ NUMBER : PLURAL]} \]

Also, we cannot encode a global FCR such as

\[ \text{TENSE = PAST} \Rightarrow \text{[AGREEMENT]} \]

unless we make TENSE a sub-feature of AGREEMENT alone. This is important because allowing arbitrary or global constraints on well-formed feature structures leads to undecidable systems if coupled with structure sharing (Blackburn and Spaan, 1991).

Our feature geometries, just like the ones used in phonology, specify whether or not the continuations of a given path are pairwise incompatible. For example, the attributes FINITE and NON_FINITE can be made incompatible continuations of the attribute VERB_FORM. As a result, in any actual feature structure at most one edge can lead from a node that a path ending in VERB_FORM leads to. What this mechanism allows us to express are also local FCRs, e.g.,

\[ \neg (\text{VERB_FORM FINITE} \land \text{VERB_FORM NON_FINITE}) \]

in this case.
2.2. How Are Feature Geometries Used?

The main advantage of using feature geometries is that it makes the unification operation and the unifiability test more efficient. Traditional unification only fails if atomic values clash, whereas geometry-based unification will fail if incompatible continuations of a path are to be unified. As a matter of course, this means that an extra check is performed each time new continuations are created during unification. However, if the feature geometry is reasonably structured (i.e., not flat), then the cost of this extra checking is significantly less than the gain from early unification failure. In the typical case, the growth of the comparative advantage of early unification failure over traditional unification (i.e., the proportion of all possibilities of failure to the number of leaves) should grow faster than its comparative disadvantage, i.e., the number of checks.

If feature geometries are used as intended, then the major distinctions between linguistic objects are made by attributes closer to the root of a feature structure, and minor features are in deeply subordinate positions. For example, the information that something is a verb will be superordinate to the information that it has a second person form. As a consequence, the most frequent reason for the failure of unification (which is a conflict between major class features) will be detected earliest. Typically, the opposite is true in traditional unification, i.e., only conflicts between terminal nodes of feature structures are detected. In such systems, major category clashes are found early enough only if the feature structures are very flat, which is undesirable for other reasons.

Moreover, the use of feature geometries assists the grammar-writer to develop her/his grammar in two ways. First, requiring the grammar-writer to specify a feature geometry and write rules accordingly forces her/him to take the semantics of features and feature structures more seriously than is typically the case. Second, since feature geometries define the set of possible feature structures, they also determine which paths can share values. The checking of structure sharing is not necessary during run-time unification, because it can be successfully dealt with at compile-time, thus providing additional error checking on the grammar. These two by-products of using feature geometries should lead to better grammar-writing.

3. The String Completion Limit

3.1. What Is the SCL?

The string completion limit, which is a small integer parameter of BUG's compiler, expresses a performance limitation that BUG incorporates into the automaton it produces. Imposing constraints on the complexity of derivation trees has a long tradition in linguistics. Most proposals of this sort, such as Yngve's (1961), which limits the depth of possible derivation trees, or limitations on the direction of their branching (e.g., Yngve, 1960) are either too weak or too strong on their own. However, there is a suggestion that we find broad enough in its coverage, and yet conceptually simple. This is Kornai's (1984) hypothesis, in terms of which any string that can be the beginning of a grammatical string can be completed with $k$ or less terminal symbols, where $k$ (i.e., the SCL) is a small integer. For example, consider:

(1) This is the dog that chased the cat that ate the rat that stole the cheese.

In this string, each portion up to a numbered position can be completed with at most one word, as the following table illustrates (position numbers are on the left, completions in the middle, and the minimum completion length $K$ on the right):

| Position | Completion | $K$ |
|----------|------------|-----|
| 1, 5, 9, 13 | John. | 1 |
| 2, 6, 10, 14 | cheese. | 1 |
| 3, 7, 11, 15 | ... | 0 |
| 4, 8, 12, 16 | stinks. | 1 |

On the other hand, the following string, although its portions up to each number are grammatical, will be excluded if the SCL is smaller than 5:

(2) The cheese that rat that ate the cat that ate the rat that stole the cheese.

The corresponding table is:

| Position | Completion | $K$ |
|----------|------------|-----|
| 1 | cheese stinks. | 2 |
| 2 | ... | 1 |
| 3 | ... rots stinks. | 2 |
| 4 | ... rat ate rots. | 3 |
| 5 | ... ate rots. | 2 |
| 6 | ... stinks ate rots. | 3 |
| 7 | ... cat chased ate stinks. | 4 |
| 8 | ... chased ate stinks. | 3 |
| 9 | ... stinks ate stole rots. | 4 |
| 10 | ... dog chased ate stole stinks. | 5 |
| 11 | ... chased ate stole stinks. | 4 |
| 12 | ... ate stole stinks. | 3 |
| 13 | ... stole stinks. | 2 |
| 14 | ... stinks. | 1 |

(This seems to show that the SCL in terms of words must be 3 or 4.)
As (2) shows, the SCL imposes a limit on the depth of center-embedding; but, as can be seen from (1), it does not constrain the depth of right-branching structures. Left branching, however, is limited, though the effect of this limitation is less pronounced than in the case of center-embedding. The example with the highest $K$ that we could find in English can be accommodated if $k$ is 3:

(3) After$_1$ a$_2$ very$_3$

(3') 1: ... walking, sleep! $K = 2$
2: ... walk, sleep! $K = 2$
3: ... long walk, sleep! $K = 3$

Although the current implementation of BUG uses the context-free source grammar format, in which so-called cross-serial dependencies cannot be expressed, it is worth noting that the SCL also puts an upper bound on the length of these:

(4) John$_1$, Eve$_2$, Carlos$_3$, and$_4$ Peter$_8$ married respectively$_9$, Sally$_7$, Paul$_4$, Susan$_9$ and$_10$ Inez.

(4') 1: ... sleeps. $K = 1$
2,3: ... and Peter sleep. $K = 3$
4: ... Peter sleep. $K = 2$
5: ... sleep. $K = 1$
6: ... Sally, Paul, Susan and Inez. $K = 5$
7: ... Paul, Susan and Inez. $K = 2$
8: ... Susan and Inez. $K = 3$
9: ... and Inez. $K = 2$
10: ... Inez. $K = 1$

The SCL has two additional consequences (and maybe more). First, it excludes certain lexical categories, such as modifiers of adjective modifiers (if $k < 4$). If, say, shlumma were a word of that category, then we would need at least 4 words to complete After a shlumma... (cf. (3) above). Second, an upper limit is placed on the number of obligatory daughters of non-terminal nodes.

3.2. How Is the SCL Used?

The way in which we can produce the biggest regular subset of a context-free language that respects the SCL can be sketched as follows. First we produce an RTN (recursive transition network) equivalent to the source grammar, call it $A$. (An RTN is like a finite-state automaton, but its input symbols may be RTNs or terminal symbols.) Then we assign a minimum completion length ($K$ in the tables above) to each node (accepting states will have $K = 0$). If $B$ is an RTN accepted by the transition from state $s_1$ to state $s_2$ in $A$, then we try to replace the transition with $B$ itself, so that initial state of $B$ becomes $s_1$ and its accepting states become $s_2$. (This can be done with standard techniques.) Since the $K$-value of $s_2$ may be bigger than 0, assigning $K$ values to some states of $B$ may be impossible (if those values would exceed $k$). We leave out those states (and whatever additional states and transitions depend on them).

In those cases when the above procedure would not terminate (i.e., when $s_2$ is an accepting state in $A$ and $B$ is the same RTN as some other RTN $C$ the acceptance of which takes the machine to $s_2$, we eliminate the transition corresponding to $B$, and collapse $s_1$ with the initial state of $C$ (with the standard technique). So the procedure will terminate in all cases. In the current implementation, we use the actual finite-state network so produced, but (as our reviewer notes) we could as well use the RTN directly, and compute whether the SCL is respected as we go. We have not made experiments with this latter solution, so we cannot compare it with our current solution in terms of space and time requirements.

4. SD Versus SC

One of the most important among BUG's features is the separation of structural descriptions from structural changes in source rules. Although the unificationalists have been asserting that this old-fashioned distinction should be abandoned (arguing that pieces of information coming from different sources have the same status), many voices have been raised to show that the origins of a piece of information may matter (see Zaenen and Karttunen, 1984; Pullum and Zwicky, 1986; Inglia. 1990).

The structural description in a BUG rule specifies the conditions under which the rule can be applied in the parsing process. That is, when parsing, it refers to the right-hand side of the rewrite rule only, and it is never used to update any feature structure. The structural change, on the other hand, describes what action to take when the structural description is satisfied, i.e., how to build a new feature structure (when parsing, this corresponds to the left-hand side of the context-free rule). Thus, structural descriptions are used to check unifiability, whereas the application of structural changes actually builds structure.

In usual unification-based grammars, the conditions of applying a rule are satisfied if some unification succeeds. In BUG, what determines whether a rule should apply is unifiability. Unifiability differs from unification in a crucial respect, which is illustrated by the following example:

$A$: [ ]
$B$: [NUMBER = SINGULAR]
$C$: [NUMBER = PLURAL]

$A$ is unifiable with $B$ and $A$ is unifiable with $C$, even though $B$ is not unifiable with $C$. Therefore, if a structural description requires unifiability of $A$
with both $B$ and $C$, it will be satisfied. However, if we were to formulate this requirement in terms of unification, as is currently done in unification-based grammars, then $A$, $B$ and $C$ will not satisfy this requirement. A similar example from 'real life' is the requirement that the auxiliary verb should agree with each subject of a co-ordination:

(5) *Is/*Are Jean leaving and the others arriving?

In this example, NUMBER of is is not unifiable with that of the others, and NUMBER of are is not unifiable with that of Jean, so traditional unification-based grammars and BUG would yield the same (correct) result. Now, consider:

(6) Will Jean leave and the others arrive?

This sentence is in because will's NUMBER is unifiable with both that of Jean and that of the others, although the unification of all three NUMBER values still leads to failure. So BUG will behave correctly in this case.

5. Generative Capacity

Somewhat misleadingly, we have avoided so far making a distinction between the context-free grammar format and context-free grammars. In actual fact, it is well-known that a unification-based grammar in the context-free format is not context-free unless the number of possible feature structures arising in all its possible derivations is finite. By the same token, the automata compiled by BUG would not recognize a regular language if we did not constrain the possible feature structures that they give rise to. The separation of SDs from SCs allows BUG to avoid this problem. Since SDs are only used in unifiability tests and are never modified at run-time, they can be constrained in such a way that they yield a finite set of equivalence classes of feature structures. Moreover, carrying out SCs only affects the structures being built and cannot interfere with the trajectory through the automaton. Incidentally, this means that unification (but not unifiability tests!) may never fail. For that purpose, we use an associative, idempotent and commutative version of 'default unification' (see Bouma, 1990), which we are not going into here. The automaton produced by BUG is, thus, actually finite-state. We consider this an extremely important benefit, if not the most important one, of separating SDs from SCs in a grammar-writing system.

References

Blackburn, Patrick and Edith Spaan. 1991. 'Some complexity results for Attribute Value Structures'. To appear in: Proceedings of the Eighth Amsterdam Colloquium.

Bouma, Gosse. 1990. 'Defaults in unification grammar'. In: Proceedings of the 28th Annual Meeting of the ACL, ACL, Pittsburgh.

Clements, George N. 1985. 'The geometry of phonological features'. Phonology Yearbook 2, 223–250.

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum and Ivan Sag. 1985. Generalized Phrase Structure Grammar. Harvard University Press, Cambridge MA.

Ingria, Robert J.P. 1990. 'The limits of unification'. In: 28th Annual Meeting of ACL: Proceedings of the Conference. ACL, Morristown, NJ. Pp. 194–204.

Kornai, András. 1984. 'Natural Languages and the Chomsky Hierarchy'. In: Proceedings of the ACL Second European Chapter Conference. ACL, Geneva. Pp. 1–7.

Pullum, Geoffrey K., and Arnold M. Zwicky. 1986. 'Phonological resolution of syntactic feature conflict'. Language 62, 751–773.

Zaenen, Annie and Lauri Karttunen. 1984. 'Morphological non-distinctness and co-ordination'. In: ESCOL 84, pp. 309–320.

Yngve, Victor H. 1961. 'The depth hypothesis'. Language 61, 283-305.

Yngve, Victor H. 1960. 'A model and an hypothesis for language structure'. Proceedings of the American Philosophical Society 104, 444–466.
Appendix: Example BUG source files and run

; Geometry for simple categorial grammar
; Major features: category and semantics,
; both of them may be present
; at the same time
< >= {cat sem}
; Category is simple or complex
; (but not both):
<cat> = [simple complex]
; Simple category is np, s or n:
<cat simple> = [np s n]
; A complex category consists of an input,
; a result, and a slash:
<cat complex> = {inp res slash}
; The input must be a simple category here:
<cat complex inp> = <cat simple>
; The result may be any category:
<cat complex res> = <cat>
; The slash is either forward or backward:
<cat complex slash> = [forw back]
; Semantics is analogous to category:
<sem> = [sim cam]
; (no constraint on simple values)
<sem com> = {fun arg}
<sem com fun> = <sem>
<sem com arg> = <sem>
; End of geometry

; Start category:
; Name of start category:
Sentence
; SD:
; it has to be of category s:
<Sentence cat simple s>
; SC:
; only the semantics is kept:
<sem> = <Sentence sem>
; End of start category

; Rules:
; The name of forward application rule:
"Forward application"
; Production schema:
RES --> FUN ARG
; SD:
; FUN must be a complex category
; with forward slash:
<FUN cat complex slash forw>
; ARG must have a simple category:
<ARG cat simple>
; FUN's input must be ARG's category:
<FUN cat complex inp> == <ARG cat simple>
; SC:
; RES's category is FUN's result:
<cat> = <FUN cat complex res>
; RES's semantics is as expected:
<sem com fun> = <FUN sem>
<sem com arg> = <ARG sem>
; Backward application is very similar:
"Backward application"
RES --> ARG FUN
<FUN cat complex slash back>
<ARG cat simple>
<FUN cat complex inp> == <ARG cat simple>
<cat> = <FUN cat complex res>
<sem com fun> = <FUN sem>
<sem com arg> = <ARG sem>
; End of rules

; Sample lexical items:
; '//' indicates the beginning of a lexicon:

"Joe" ; np 'JOE'
<cat simple np>
<sem sim JOE>
"hit" ; (s
p)/np 'HIT'
<cat complex>
<inp np>
<res complex>
<lap up>
<res simple s>
<slash back>
<slash oru>
<sem cam art sim JDE>
<fun cam fun sim HIT>
<sem cam arg sim THE>
<art sim fun sim BALL>

# Example run:
Y, bug -i cat cat
doing cat.gs --> cat.go.
doing cat.ls --> cat.lo.
Joe
hit
the
ball
Loading lexicon cat.lo.
==> Joe hit the ball.
sem com arg sim JOE
fun com fun sim HIT
arg com fun sim THE
arg sim BALL.