A Unification Method for Disjunctive Feature Descriptions

Robert T. Kasper
USC/Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292

and

Electrical Engineering and Computer Science Department
University of Michigan

Abstract

Although disjunction has been used in several unification-based grammar formalisms, existing methods of unification have been unsatisfactory for descriptions containing large quantities of disjunction, because they require exponential time. This paper describes a method of unification by successive approximation, resulting in better average performance.

1 Introduction

Disjunction has been used in several unification-based grammar formalisms to represent alternative structures in descriptions of constituents. Disjunction is an essential component of grammatical descriptions in Kay's Functional Unification Grammar [6], and it has been proposed by Karttunen as a linguistically motivated extension to PATR-II [2].

In previous work two methods have been used to handle disjunctive descriptions in parsing and other computational applications.

The first method requires expanding descriptions to *disjunctive normal form* (DNF) so that the entire description can be interpreted as a set of structures, each of which contains no disjunction. This method is exemplified by Definite Clause Grammar [8], which eliminates disjunctive terms by expanding each rule containing disjunction into alternative rules. It is also the method used by Kay [7] in parsing FUG. This method works reasonably well for small grammars, but it is clearly unsatisfactory for descriptions containing more than a small number of disjunctions, because the DNF expansion requires an amount of space which is exponential in the number of disjunctions.

The second method, developed by Karttunen [2], uses constraints on disjuncts which must be checked whenever a disjunct is modified. Karttunen's method is only applicable to value disjunctions (i.e. those disjunctions used to specify the value of a single feature), and it becomes complicated and inefficient when disjuncts contain non-local dependencies (i.e. values specified by path expressions denoting another feature).

In previous research [4,5] we have shown how descriptions of feature structures can be represented by a certain type of logical formula, and that the consistency problem for disjunctive descriptions is NP-complete. This result indicates, according to the widely accepted mathematical assumption that $P \neq NP$, that any complete unification algorithm for disjunctive descriptions will require exponential time in the worst case. However, this result does not preclude algorithms with better average performance, such as the method described in the remainder of this paper. This method overcomes the shortcomings of previously existing methods, and has the following desirable properties:

1. It applies to descriptions containing general disjunction and non-local path expressions;
2. It delays expansion to DNF;
3. It can take advantage of fast unification algorithms for non-disjunctive directed graph structures.

2 Data Structures

The most common unification methods for non-disjunctive feature structures use a directed graph (DG) representation, in which arcs are labeled by names of features, and nodes correspond to values of features. For an introduction to these methods, the reader is referred to Shieber's survey [11]. In the remainder of this section we will define a data structure for disjunctive descriptions, using DG structures as a basic component.

In the following exposition, we will carefully observe the distinction between feature structures and their descriptions, as explained in [4]. Feature structures will be represented by DGs, and descriptions of feature structures will be represented by logical formulas of the type described in [4]. The

NIL	denoting *no information*;
TOP	denoting *inconsistent information*;
a	where $a \in A$, to describe atomic values;
$l : \phi$	where $l \in L$ and $\phi \in$ FDL, to describe structures in which the feature labeled by l has a value described by ϕ;
$[\![< p_1 >, \ldots, < p_n >]\!]$	where each $p_i \in L^*$, to describe an equivalence class of paths sharing a common value in a feature structure;
$\phi \wedge \psi$	where $\phi, \psi \in$ FDL;
$\phi \vee \psi$	where $\phi, \psi \in$ FDL.

Figure 1: Syntax of FDL Formulas.

syntax for formulas of this *feature description logic* (hereafter called FDL) is given in Figure 1.[1]

Note, in particular, that disjunction is used in descriptions of feature structures, but not in the structures themselves. As we have shown (see [9]) that there is a unique minimal satisfying DG structure for any nondisjunctive FDL formula, we can represent the parts of a formula which do not contain any disjunction by DGs. DGs are a more compact way of representing the same information that is contained in a FDL formula, provided the formula contains no disjunction.

Let us define an *unconditional conjunct* to be a conjunct of a formula which contains no occurrences of disjunction. After path expansion any formula can be put into the form:

$$uconj \wedge disj_1 \wedge \ldots \wedge disj_m$$

where *uconj* contains no occurrences of disjunction, and each $disj_i$, for $1 \leq i \leq m$, is a disjunction of two or more alternatives. The *uconj* part of the formula is formed by using the commutative law to bring all unconditional conjuncts of the formula together at the front. Of course, there may be no unconditional conjuncts in a formula, in which case *uconj* would be the formula NIL.

Each disjunct may be any type of formula, so disjuncts can also be put into a similar form, with all unconditional conjuncts grouped together before all disjunctive components. Thus the disjunctions of a formula can be put into the form

$$(uconj_1 \wedge disj_{11} \wedge \ldots \wedge disj_{1x}) \vee \ldots \vee (uconj_n \wedge disj_{n1} \wedge \ldots \wedge disj_{ny}).$$

The embedding of conjuncts within disjuncts is preserved, but the order of conjuncts may be changed.

The unconditional conjuncts of a formula contain information that is more definite than the information contained in disjunctions. Thus a formula can be regarded as having a definite part, containing only unconditional conjuncts, and an indefinite part, containing a set of disjunctions. The definite part contains no disjunction, and therefore it may be represented by a DG structure. To encode these parts of a formula, let us define a *feature-description* as a type of data structure, having two components:

[1]Let A and L be sets of symbols which are used to denote atomic values and feature labels, respectively.

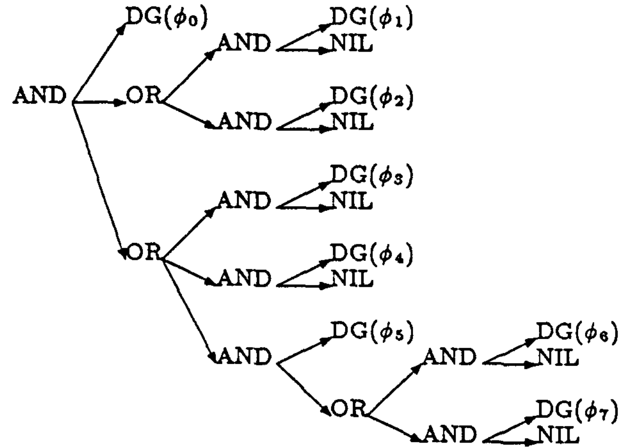

Figure 2: AND/OR graph representation of a feature description.

definite: a DG structure;

indefinite: a SET of *disjunctions*, where each *disjunction* is a SET of *feature-descriptions*.

It is possible to convert any FDL formula into a feature-description structure by a simple automatic procedure, as described in [5]. This conversion does not add or subtract any information from a formula, nor increase its size in any significant way. It simply identifies components of the formula which may be converted into a more efficient representation as DG structures.

A feature-description is conceptually equivalent to a special kind of AND/OR graph, in which the terminal nodes are represented by DG structures. For example, an AND/OR graph equivalent to the formula,

$$\phi_0 \wedge (\phi_1 \vee \phi_2) \wedge (\phi_3 \vee \phi_4 \vee (\phi_5 \wedge (\phi_6 \vee \phi_7)))$$

is shown in Figure 2. In the AND/OR graph representation, each AND-node represents a feature-description. The first outgoing arc from an AND-node represents the definite component of a feature-description, and the remaining outgoing arcs represent the indefinite component. Each OR-node represents a disjunction.

```
Function UNIFY-DESC (f, g) Returns feature-description:
     where f and g are feature-descriptions.

1. Unify definite components.
Let new-def = UNIFY-DGS (f.definite, g.definite).
If new-def = TOP, then return (failure).
Let desc = a feature-description with:
     desc.definite = new-def,
     desc.indefinite = f.indefinite ∪ g.indefinite.
If desc.indefinite = ∅,
     Then return (desc);

Else begin;

2. Check compatibility of indefinite components with new-def.
     Let new-desc = CHECK-INDEF (desc, new-def).
     If new-desc = failure, then return (failure);

3. Complete exhaustive consistency checking, if necessary.
     Else if new-desc.indefinite = ∅
          OR if complete checking is not required,
          Then return (new-desc);
     Else begin;
          Let n = 1.
          Repeat while n < cardinality of new-desc.indefinite:
               new-desc := NWISE-CONSISTENCY (new-desc, n).
               n := n + 1.
          return (new-desc).
          end.
     end.
```

Figure 3: Unification algorithm for feature-descriptions.

3 The Algorithm: Unification by Successive Approximation

In this section we will give a complete algorithm for unifying two feature-descriptions, where one or both may contain disjunction. This algorithm is designed so that it can be used as a relatively efficient approximation method, with an optional step to perform complete consistency checking when necessary.

Given two feature-descriptions, the strategy of the unification algorithm is to unify the definite components of the descriptions first, and examine the compatibility of indefinite components later. Disjuncts are eliminated from the description when they are inconsistent with definite information. This strategy avoids exploring disjuncts more than once when they are inconsistent with definite information.

The exact algorithm is described in Figure 3. It has three major steps.

In the first step, the definite components of the two descriptions are unified together, producing a DG structure, new-def, which represents the definite information of the result. This step can be performed by existing unification algorithms for DGs.

In the second step, the indefinite components of both descriptions are checked for compatibility with new-def, using the function CHECK-INDEF, which is defined in Figure 4. CHECK-INDEF uses the function CHECK-DISJ, defined in Figure 5, to check the compatibility of each disjunction with the DG structure given by the parameter *cond*.

The compatibility of two DGs can be checked by almost the same procedure as unification, but the two structures being checked are not actually merged as they are in unification.

In the third major step, if any disjunctions remain, and it is necessary to do so, disjuncts of different disjunctions are considered in groups, to check whether they are compatible together. This step is performed by the function NWISE-CONSISTENCY, defined in Figure 6.

When the parameter n to NWISE-CONSISTENCY has the value 1, then one disjunct is checked for compatibility with all other disjunctions of the description in a pairwise manner. The pairwise manner of checking compatibility can be generalized to groups of any size by increasing the value of the parameter n.

While this third step of the algorithm is necessary in order to insure consistency of disjunctive descriptions, it is not necessary to use it every time a description is built during a parse. In practice, we find that the performance of the algorithm can be tuned by using this step only at strategic points during a parse, since it is the most inefficient step of the al-

```
Function CHECK-INDEF (desc, cond) Returns feature-description:
    where desc is a feature-description,
    and cond is a DG.

Let indef = desc.indefinite (a set of disjunctions).
Let new-def = desc.definite (a DG).
Let unchecked-parts = true.

While unchecked-parts, begin;

    unchecked-parts := false.
    Let new-indef = ∅.

    For each disjunction in indef:
        Let compatible-disjuncts = CHECK-DISJ (disjunction, cond).
        If cardinality of compatible-disjuncts is:
            0 : Return (failure);
            1 : Let disjunct = single element of compatible-disjuncts.
                new-def := UNIFY-DGS (new-def, disjunct.definite).
                new-indef := new-indef ∪ disjunct.indefinite.
                unchecked-parts := true;
            otherwise : new-indef := new-indef ∪ {compatible-disjuncts}.

    Prepare to check remaining disjunctions for compatibility with new-def.
    cond := new-def.
    indef := new-indef.
    end (while loop).

Let new-desc = make feature-description with:
    new-desc.definite = new-def,
    new.desc.indefinite = new-indef.

Return (new-desc).
```

Figure 4: Algorithm to check compatibility of indefinite parts of feature-descriptions with respect to a condition DG.

```
Function CHECK-DISJ (disj, cond) Returns disjunction:
    where disj is a disjunction of feature-descriptions,
    and cond is a DG.

Let new-disj = ∅ (a set of feature-descriptions).

For each disjunct in disj:
    If DGS-COMPATIBLE? (cond, disjunct.definite),
        Then if disjunct.indefinite = ∅,
            Then new-disj := new-disj ∪ {disjunct};
        Else begin;
            Let new-disjunct = CHECK-INDEF (disjunct, cond).
            If new-disjunct ≠ failure, then begin;
                new-disj := new-disj ∪ {new-disjunct}.
                end.
            end.

Return (new-disj).
```

Figure 5: Algorithm to check compatibility of disjunctions with respect to a condition DG.

```
Function NWISE-CONSISTENCY (desc, n) Returns feature-description:
    where desc is a feature-description.

If number of disjunctions in desc.indefinite ≤ n,
    Then Return (desc).

Let def = desc.definite.
Let indef = desc.indefinite.
Let new-indef = ∅.

While disjunctions remain in indef:
    Let disjunction = remove one disjunction from indef.
    Let new-disj = ∅.

    For each disjunct in disjunction:
        Let disjunct-def = UNIFY-DGS (def, disjunct.definite).
        Let disjunct-indef = disjunct.indefinite ∪ indef ∪ new-indef.
        Let hyp-desc = make feature-description with:
            hyp-desc.definite = disjunct-def,
            hyp-desc.indefinite = disjunct-indef.
        If n = 1,
            Then let new-desc = CHECK-INDEF (hyp-desc, disjunct-def).
        Else let new-desc = NWISE-CONSISTENCY (hyp-desc, n-1).
        If new-desc ≠ failure,
            Then new-disj := new-disj ∪ {new-desc}.

    If cardinality of new-disj is:
        0: Return (failure);
        1: Let new-desc = single element of new-disj.
            def := new-desc.definite.
            indef := new-desc.indefinite.
            new-indef := ∅;
        otherwise: (keep this disjunction in result)
            new-indef := new-indef ∪ {new-disj}.

Let result-desc = make feature-description with:
    result-desc.definite = def,
    result-desc.indefinite = new-indef.
Return (result-desc).
```

Figure 6: Algorithm to check compatibility of disjunctions of a description by checking groups of n disjunctions.

gorithm. In our application, using the Earley chart parsing method, it has proved best to use NWISE-CONSISTENCY only when building descriptions for complete edges, but not when building descriptions for active edges.

Note that two feature-descriptions do not become permanently linked when they are unified, unlike unification for DG stuctures. The result of unifying two descriptions is a new description, which is satisfied by the intersection of the sets of structures that satisfy the two given descriptions. The new description contains all the information that is contained in either of the given descriptions, subtracting any disjuncts which are no longer compatible.

4 An example

In order to illustrate the effect of each step of the algorithm, let us consider an example of unifying the description of a known constituent with the description of a portion of a grammar. This exemplifies the predominant type of structure building operation needed in a parsing program for Functional Unification Grammar. The example given here is deliberately simple, in order to illustrate how the algorithm works with a minimum amount of detail. It is not intended as an example of a linguistically motivated grammar.

Let us trace what happens when the two descriptions of Figure 7 are given as inputs to the function UNIFY-DESC. Figure 8 shows the feature-description which results after step 1 of the algorithm. The definite components of the two descriptions have been unified, and their indefinite components have been conjoined together.

In step 2 of the algorithm each of the disjuncts of DESC.INDEFINITE is checked for compatibility with DESC.DEFINITE, using the function CHECK-INDEF. In this case, all disjuncts are compatible with the definite information, except for one; the disjunct of the third disjunction which contains the feature *Number* : *Sing*. This disjunct is eliminated, and the only remaining disjunct in the disjunction (i.e., the disjunct containing *Number* : *Pl*) is unified with DESC.DEFINITE. The result after this step is shown in Figure 9. The four disjuncts that remain are numbered for convenience.

In step 3, NWISE-CONSISTENCY is used with 1 as the value of the parameter n. A new description is hypothesized by unifying disjunct (1) with the definite component of the description (i.e., NEW-DESC.DEFINITE). Then disjuncts (3) and (4) are checked for compatibility with this hypothesized structure: (3) is not compatible, because the values of the *Transitivity* features do not unify. Disjunct (4) is also incompatible, because it has *Goal* : *Person* : 3, and the hy-

GRAMMAR:

$$\text{DEFINITE} = \begin{bmatrix} Rank : Clause \\ Subj : Case : Nom \end{bmatrix}$$

INDEFINITE =

$$\left(\begin{bmatrix} Voice : Passive \\ Transitivity : Trans \\ [\![< Subj >, < Goal >]\!] \end{bmatrix} \vee \begin{bmatrix} Voice : Active \\ [\![< Subj >, < Actor >]\!] \end{bmatrix} \right)$$
$$\left(\begin{bmatrix} Transitivity : Intrans \\ Actor : Person : 3 \end{bmatrix} \vee \begin{bmatrix} Transitivity : Trans \\ Goal : Person : 3 \end{bmatrix} \right)$$
$$\left(\begin{bmatrix} Number : Sing \\ Subj : Number : Sing \end{bmatrix} \vee \begin{bmatrix} Number : Pl \\ Subj : Number : Pl \end{bmatrix} \right)$$

SUBJECT CONSTITUENT:

$$\text{DEFINITE} = Subj : \begin{bmatrix} Lex : y'all \\ Person : 2 \\ Number : Pl \end{bmatrix}$$

INDEFINITE $= NIL$

Figure 7: Two descriptions to be unified.

pothesized description has $[\![< Subj >, < Goal >]\!]$, along with $Subj : Person : 2$. Therefore, since there is no compatible disjunct among (3) and (4), the hypothesis that (1) is compatible with the rest of the description has been shown to be invalid, and (1) can be eliminated. It follows that disjunct (2) should be unified with the definite part of the description. Now disjuncts (3) and (4) are checked for compatibility with the definite component of the new description: (3) is no longer compatible, but (4) is compatible. Therefore, (3) is eliminated, and (4) is unified with the definite information. No disjunctions remain in the result, as shown in Figure 10.

5 Complexity of the Algorithm

Referring to Figure 3, note that the function UNIFY-DESC may terminate after any of the three major steps. After each step it may detect inconsistency between the two descriptions and terminate, returning failure, or it may terminate because no disjunctions remain in the description. Therefore, it is useful to examine the complexity of each of the three steps independently.

Let n represent the total number of symbols in the combined description $f \wedge g$, and d represent the total number of disjuncts (in both top-level and embedded disjunctions) contained in $f \wedge g$.

Step 1. This step performs the unification of two DG structures. Ait-Kaci [1] has shown how this operation can be performed in almost linear time by the UNION/FIND algorithm. Its time complexity has an upper bound of $O(n \log n)$. Since an unknown amount of a description may be contained in the definite component, this step of the algorithm also requires $O(n \log n)$ time.

Step 2. For this step we examine the complexity of the function CHECK-INDEF. There are two nested loops in CHECK-INDEF, each of which may be executed at most once for each disjunct in the description. The inner loop checks the compatibility of two DG structures, which requires no more time than unification. Thus, in the worst case, CHECK-INDEF requires $O(d^2 n \log n)$ time.

Step 3. NWISE-CONSISTENCY requires at most $O(2^{d/2})$ time. In this step, NWISE-CONSISTENCY is called at most $(d/2) - 1$ times. Therefore, the overall complexity of step 3 is $O(2^{d/2})$.

Discussion. While the worst case complexity of the entire algorithm is $O(2^d)$, an exponential, it is significant that it often terminates before step 3, even when a large number of disjunctions are present in one of the descriptions. Thus, in many practical cases the actual cost of the algorithm is bounded by a polynomial that is at most $d^2 n \log n$. Since d must be less than n, this complexity function is almost cubic. Even when step 3 must be used, the number of remaining disjunctions is often much fewer than $d/2$, so the exponent is usually a small number. The algorithm performs well in most cases, because the three steps are ordered in increasing complexity, and the number of disjunctions can only decrease during unification.

6 Implementation

The algorithm presented in the previous sections has been implemented and tested as part of a general parsing method for Systemic Functional Grammar, which is described in [3]. The algorithm was integrated with the structure building module of the PATR-II system [10], written in the Zetalisp programming language.

While the feature-description corresponding to a grammar may have hundreds of disjunctions, the descriptions that result from parsing a sentence usually have only a small number of disjunctions, if any at all. Most disjunctions in a systemic grammar represent possible alternative values that some particular feature may have (along with the grammatical consequences entailed by choosing particular values for the feature). In the analysis of a particular sentence most features have a unique value, and some features are not present at all. When disjunction remains in the description of a sentence after parsing, it usually represents ambiguity or an underspecified part of the grammar.

With this implementation of the algorithm, sentences of up to 10 words have been parsed correctly, using a grammar which contains over 300 disjunctions. The time required for most sentences is in the range of 10 to 300 seconds, running on lisp machine hardware.

The fact that sentences can be parsed at all with a grammar containing this many disjunctions indicates that the algorithm is performing much better than its theoretical worst case time of $O(2^d)$.[2] The timings, shown in Table 1, obtained from the experimental parser for systemic grammar also indicate that a dramatic increase in the number of disjunctions in the grammar does not result in an exponential increase in parse time. G_{98} is a grammar containing 98 disjunctions,

[2]Consider, $2^{300} \gg 2^{80}$, and 2^{80} is taken to be a rough estimate of the number of particles in the universe.

$$\text{DESC.DEFINITE} = \begin{bmatrix} Rank : Clause \\ Subj : \begin{bmatrix} Case : Nom \\ Lex : y'all \\ Person : 2 \\ Number : Pl \end{bmatrix} \end{bmatrix}$$

$$\text{DESC.INDEFINITE} =$$

$$\left(\begin{bmatrix} Voice : Passive \\ Transitivity : Trans \\ [\langle Subj \rangle, \langle Goal \rangle] \end{bmatrix} \vee \begin{bmatrix} Voice : Active \\ [\langle Subj \rangle, \langle Actor \rangle] \end{bmatrix} \right)$$

$$\left(\begin{bmatrix} Transitivity : Intrans \\ Actor : Person : 3 \end{bmatrix} \vee \begin{bmatrix} Transitivity : Trans \\ Goal : Person : 3 \end{bmatrix} \right)$$

$$\left(\begin{bmatrix} Number : Sing \\ Subj : Number : Sing \end{bmatrix} \vee \begin{bmatrix} Number : Pl \\ Subj : Number : Pl \end{bmatrix} \right)$$

Figure 8: UNIFY-DESC: After step 1 (UNIFY-DGS).

$$\text{NEW-DESC.DEFINITE} = \begin{bmatrix} Rank : Clause \\ Subj : \begin{bmatrix} Case : Nom \\ Lex : y'all \\ Person : 2 \\ Number : Pl \end{bmatrix} \\ Number : Pl \end{bmatrix}$$

$$\text{NEW-DESC.INDEFINITE} =$$

$$\left((1) \begin{bmatrix} Voice : Passive \\ Transitivity : Trans \\ [\langle Subj \rangle, \langle Goal \rangle] \end{bmatrix} \vee (2) \begin{bmatrix} Voice : Active \\ \llbracket \langle Subj \rangle, \langle Actor \rangle \rrbracket \end{bmatrix} \right)$$

$$\left((3) \begin{bmatrix} Transitivity : Intrans \\ Actor : Person : 3 \end{bmatrix} \vee (4) \begin{bmatrix} Transitivity : Trans \\ Goal : Person : 3 \end{bmatrix} \right)$$

Figure 9: UNIFY-DESC: After step 2 (CHECK-INDEF).

$$\text{NEW-DESC.DEFINITE} = \begin{bmatrix} Rank : Clause \\ Subj : \begin{bmatrix} Case : Nom \\ Lex : y'all \\ Person : 2 \\ Number : Pl \end{bmatrix} \\ Number : Pl \\ Voice : Active \\ [\langle Subj \rangle, \langle Actor \rangle] \\ Transitivity : Trans \\ Goal : Person : 3 \end{bmatrix}$$

$$\text{NEW-DESC.INDEFINITE} = NIL$$

Figure 10: UNIFY-DESC: After step 3 (NWISE-CONSISTENCY).

Sentence	G_{98}	G_{440}
Nigel has been speaking English.	22.9	144.3
Nigel has been speaking English to me.	28.6	203.5

Table 1: Timings obtained from a systemic parser.

and G_{440} is a grammar containing 440 disjunctions. The total time used to parse each sentence is given in seconds.

7 Conclusions

The unification method presented here represents a general solution to a seemingly intractable problem. This method has been used successfully in an experimental parser for a grammar containing several hundred disjunctions in its description. Therefore, we expect that it can be used as the basis for language processing systems requiring large grammatical descriptions that contain disjunctive information, and refined as necessary and appropriate for specific applications.

While the range of speed achieved by a straightforward implementation of this algorithm is acceptable for grammar testing, even greater efficiency would be desirable (and necessary for applications demanding fast real-time performance). Therefore, we suggest two types of refinement to this algorithm as topics for future research: using heuristics to determine an opportune ordering of the disjuncts within a description, and using parallel hardware to implement the compatibility tests for different disjunctions.

Acknowledgements

I would like to thank Bill Rounds, my advisor during graduate studies at the University of Michigan, for his helpful criticism of earlier versions of the algorithm which is presented here. I would also like to thank Bill Mann for suggestions during its implementation at USC/ISI, and Stuart Shieber for providing help in the use of the PATR-II system.

This research was sponsored in part by the United States Air Force Office of Scientific Research contracts FQ8671-84-01007 and F49620-87-C-0005, and in part by the United States Defense Advanced Research Projects Agency under contract MDA903-81-C-0335; the opinions expressed here are solely those of the author.

References

[1] Ait-Kaci, H. *A New Model of Computation Based on a Calculus of Type Subsumption.* PhD thesis, University of Pennsylvania, 1984.

[2] Karttunen, L. Features and Values. In *Proceedings of the Tenth International Conference on Computational Linguistics: COLING 84*, Stanford University, Stanford, California, July 2-7, 1984.

[3] Kasper, R. Systemic Grammar and Functional Unification Grammar. In J. Benson and W. Greaves, editors, *Systemic Functional Perspectives on Discourse: Selected Papers from the* 12th *International Systemics Workshop*, Norwood, New Jersey: Ablex (forthcoming).

[4] Kasper, R. and W. Rounds. A Logical Semantics for Feature Structures. In *Proceedings of the* 24th *Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, NY, June 10-13, 1986.

[5] Kasper, R. *Feature Structures: A Logical Theory with Application to Language Analysis.* PhD dissertation, University of Michigan, 1987.

[6] Kay, M. Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley Linguistics Society, Berkeley, California, February 17-19, 1979.

[7] Kay, M. Parsing in Functional Unification Grammar. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Parsing.* Cambridge University Press, Cambridge, England, 1985.

[8] Pereira, F. C. N. and D. H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231-278, 1980.

[9] Rounds, W. C. and R. Kasper. A Complete Logical Calculus for Record Structures Representing Linguistic Information. *Symposium on Logic in Computer Science.* IEEE Computer Society, June 16-18, 1986.

[10] Shieber, S. M. The design of a computer language for linguistic information. In *Proceedings of the Tenth International Conference on Computational Linguistics: COLING 84*, Stanford University, Stanford, California, July 2-7, 1984.

[11] Shieber, S. M. *An Introduction to Unification-based Approaches to Grammar.* Chicago: University of Chicago Press, CSLI Lecture Notes Series, 1986.