F-PATR: FUNCTIONAL CONSTRAINTS FOR UNIFICATION-BASED GRAMMARS

Kent Wittenburg
Bellcore
445 South St., MRE 2A-347
Morristown, NJ 07962-1910, USA
Internet: kentw@bellcore.com

Abstract

Motivation for including relational constraints other than equality within grammatical formalisms has come from discontinuous constituency and partially free word order for natural languages as well as from the need to define combinatory operations at the most basic level for languages with a two-dimensional syntax (e.g., mathematical notation, chemical equations, and various diagramming languages). This paper presents F-PATR, a generalization of the PATR-II unification-based formalism, which incorporates relational constraints expressed as user-defined functions. An operational semantics is given for unification that is an adaptation and extension of the approach taken by Ait-Kaci and Nasr (1989). It is designed particularly for unification-based formalisms implemented in functional programming environments such as Lisp. The application of unification in a chart parser for relational set languages is discussed briefly.

1. INTRODUCTION

For the most part, unification-based grammar formalisms (e.g., Kaplan and Bresnan 1982; Pereira and Warren 1980; Shieber 1984) have adopted string rewriting conventions from context-free grammar rules, assuming string concatenation as the basic combining operator external to the unification process itself. Kay's Functional Unification Grammar (Kay 1979), while not borrowing the conventions of CFG rewriting rules, still assumed concatenation of strings as the underlying combining operation. However, recent work in HPSG (e.g., Pollard and Sag 1987, Reape 1990, Carpenter et al. 1991) and elsewhere has sought to incorporate constraints for combining operations into the unification-based representation directly. Part of the motivation for doing so is to accommodate partially free word order and discontinuous constituency without the complication of passing along intermediate "threading" information within derivations. Such exensions to unification grammars require the use of nonequational constraints, i.e., constraints on values other than simple conditions of equality and the logical connectives built with them. Reape (1990) has proposed, for example, the relations *permutation* and *sequence union* to constrain word sequences in his HPSG fragment for German.

A different motivation for extending the constraint language for combination within unification grammars comes from languages with a two-dimensional syntax (e.g., mathematical notation, chemical equations, and various diagramming languages). Approaching such domains from a linguistic perspective requires that grammars be capable of dealing with a richer source of data types than just strings and also with specifying a richer set of combinatory operations than simple string concatenation. The approach taken by Helm and Marriott (1986, 1990) and Wittenburg, Weitzman, and Talley (1991) [hereafter WWT] is to augment declarative, unification-based grammars with relational constraints. Combinatory operations can then be defined out of the sets of relational constraints present in rule bodies. The approach in WWT includes a set-valued attribute called *cover* in feature structures. Relations such as *above*, *below*, *north-east-of*, and *connected-to* are examples that may be incorporated into cover constraints used in grammars for two-dimensional languages. These constraints apply to *sets* of the basic input vocabulary, whose members may themselves be complex objects. The use of sets in these grammars takes the place of strings, or sequences of words, as used in grammars for natural languages.

This paper presents a generalization of the PATR-II unification-based grammar formalism to incorporate relational constraints. The extension has been primarily motivated by the demands of combinatory operations in the syntax for two-dimensional languages although such constraints can be used to express more complex combinatory relations on strings as well as for other purposes (see, for example, work in CLG (Damas and Varile 1989; Balari et al. 1990)).

The approach described here arose as a result of extending a Lisp-based implementation of PATR-II used with a chart parser. A natural path was provided by

Ait-Kaci and Nasr (1989), who proposed integrating logic and functional programming by allowing constraints to be specified with applicative expressions. This work has subsequently become one of the three cornerstones of the programming language Life (Ait-Kaci 1990). The key idea is to allow interpreted functional expressions to appear as *bona fide* arguments in logical statements. Unification operations then must allow for delaying the evaluation of functional expressions until such time as argument variables become grounded, a process that leads to what Ait-Kaci and Nasr call residuation.

For the most part, the adaptation of Ait-Kaci and Nasr's methods to an extension of PATR-II proved to be straitforward. However, there are two points on which the operational semantics of F-PATR unification as defined here differs from theirs. The first, a variation on dereferencing applicative values, was motivated by the demands of caching intermediate results imposed by chart parsing. The second, atomic disjunction, allows for more expressiveness in the grammar and also, again, was motivated by the parsing algorithm we assumed. We will return to these points in Section 6.

2. FUNCTIONAL CONSTRAINTS

From the graph perspective, the basic vocabulary of PATR-II (Shieber 1984) consists of a set of arc labels and a set of terminal (leaf) node labels, the latter including a variable (or null) value. The graphs can have reentrancies at the leaf levels or higher up, which express identity (or unification) of structure.

Following Ait-Kaci and Nasr (1989), we incorporate applicative expressions (function specification followed by zero or more argument specifications), into our constraint language. Two uses of applicative expressions in the Ait-Kaci/Nasr language Le Fun concern us here. The first allows variables to equate to an (eventual) evaluation of some applicative expression whose arguments may contain variables. For example,

X = (union Y Z)

(Our convention will be to write applicative expressions using Lisp s-expression syntax, i.e, function name followed by zero or more arguments all enclosed in parentheses.) The second allows Le Fun clauses to be formed from arbitrary ground-decidable predicates, i.e., applicative expressions whose arguments also may start out as variables. For example, given the user-defined boolean function *sw-of* (south-west of), the following would be an acceptable statement: *(sw-of X Y)*.

The analogous PATR-II extension to the first of these allows leaf nodes to be labeled with an applicative expression. Any "unbound" arguments in these applicative expressions will point to variable nodes elsewhere in the graph. Equations such as the following example will then be allowed in the language.

<mother cover> = (union <daught1 cover>
<daught2 cover>)

In F-PATR, we restrict the types of nodes represented by paths to those that may appear as leaf values, i.e., atomic, a disjunction of atoms, null (variable), or another applicative value. This restriction is significant: it does not allow for arguments in functional constraints to be of the complex attribute-value type.

The second use of applicative expressions, as predicates, allows the inclusion of functional expressions into feature specifications as independent conditions on successful unification. So here the evaluation of the expression is not associated with a leaf node's value. The statement below is an example of a such a constraint on the value of a node that might be included in graph. This predicate *sw-of* will be taken to be a condition on successful unification.

(sw-of <daught1 cover> <daught2 cover>)

The two statements above taken together would then correspond to the graph shown in Figure 1, a first approximation for a rule for forming exponent expressions in a grammar of mathematical notation. The unlabeled arcs linking arguments in applicative expressions to the variable nodes are a notational convenience, indicating a forwarding pointer. The arguments to these expressions are in fact the nodes themselves.

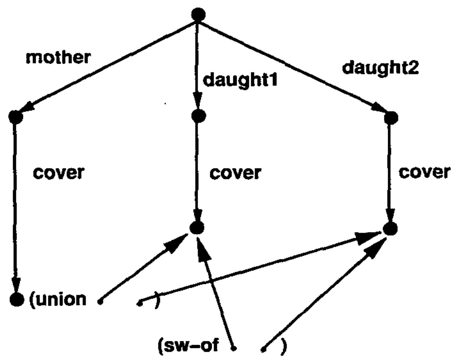

Figure 1 An F-PATR Graph

Our proposal for F-PATR feature structures begins with a vocabulary of the following types suitable for interpretive, functionally oriented programming languages such as Lisp.

Atom	Symbol or number
Fun-exp	Function, i.e, symbol pointing to a function, or lambda expression interpretable as a function, of type Atom × Atom × ... Atom → Atom or else Atom × Atom × ... Atom → List-of-atoms (where List-of-atoms will be interpreted as a logical disjunction of atomic values)
S-expression	Any complete evaluatable expression without internal references to F-PATR nodes

The following then is a BNF grammar for F-PATR equations representing feature structures:

Feat-struct ::= Statement$^+$
Statement ::= **Atom** | Equation | Appl
Equation ::= Path = Path | Path = Val | Path = Appl
Path ::= < **Atom**$^+$ >
Val ::= **Atom** | { **Atom Atom**$^+$ }
Appl ::= (**Fun-exp** Arg*)
Arg ::= Path | Appl | **S-expression**

We will assume the existence of a familiar equivalent notation for these feature equations, in which graph reentrancies (or path equivalences) are expressed by a matrix with integers used for shared reference. Predicates will follow the core attribute-value matrix. For example,

[a: 1[]
b: 2(foo <1>)]
(fie <1> <2>)

is equivalent to

<b> = (foo <a>)
(fie <a> <b>).

In addition to functional values and constraints, we augment the original PATR-II notation with atomic disjunction (interpreted as exclusive OR) as a possible value of leaf nodes. Such values are written with curly braces surrounding two or more atoms. Atomic disjunction is one of the most basic extensions to the PATR-II unification language and is in common use. If atomic values are considered to be singleton sets, unification of atomic disjunctions with other disjunctions or atoms can be operationally treated as set intersection. In F-PATR, atomic disjunctions may appear not only independently but also as arguments and values of applicative expressions.

3. DATA TYPES

In Ait-Kaci and Nasr (1989), functional expressions in feature structures are evaluated as soon as their arguments become bound. Otherwise, data structures will become *residuated*, a state representing incompletion with respect to determining constraints on unification. Ait-Kaci and Nasr's algorithms thus delay the resolution of functionally-specified values or predicates until all variables are bound, but then resolve them as early as possible once bindings occur. Here we follow this same general approach for predicates only, but not for applicative values, which are checked for readiness to evaluate only when dereferenced. Further, we expand the routines to deal with atomic disjunction.

We assume the following data types for nodes in a feature structure graph:

:Arc-list	a set of attribute labels and associated values, the latter of which may be of any type
:null	the uninstantiated "variable" type
:atomic	a singleton set of one symbol or number
:disjunct	a set of 2 or more atomic values
:appl	an applicative expression
:res-var	a residuated variable, i.e., a :null type that appears as an argument in at least one predicate
:res-disjunct	a residuated disjunction, i.e., a :disjunct type that appears as an argument in at least one predicate

The node types that may acquire residuations include :null, :disjunct, and :appl (a type for which we do not distinguish residuated from nonresiduated subtypes). There are two kinds of residuations: predicates not ready for evaluation and delayed unifications associated with the :appl type. Predicate residuations arise when a predicate contains any arguments of type :null or :appl, or else when a predicate has more than one argument of type :disjunct. During unification, any such arguments mutate to a residuated type (if they are unresidutated to start with), and the predicate is pushed onto their residuation list.

The second kind of residuation arises when unification is called for between a node of type :appl that is not ready for evaluation and any other non-:null type. The unification call itself must be delayed until such time as the function is ready for evaluation, and so a form that will provoke the unification is pushed onto the residuation list of the :appl node.

4. DEREFERENCING

The notion of dereferencing a data structure representing a feature value (or node) is common to most unification implementations. A field in the data structure indicates whether the value is to be found locally or else by following pointers to other data structures that may have been introduced through prior unification. Introducing residuations into the data structures adds the wrinkle that, during dereferencing, applicative expressions will be evaluated if they are ready. In F-Patr, dereferencing an :appl type node is in fact the only point at which to evaluate an applicative expression. This is a change from Le Fun--there arguments in applicative expressions may acquire applicative expressions as residuations that can be evaluated as argument terms become grounded during unification. This design change will be motivated in Section 6.

For each node type, the dereference function follows pointers in the usual way until no pointers remain. In addition, if the resulting node is of :appl type, we check to see if all its arguments are atomic or else lisp s-expressions, an indication that the function is ready to be evaluated. If the function evaluates to a non-nil atom or a disjunctive list of atoms, then any residuations (delayed unifications) on the node are also called. Note then that dereferencing can itself fail as a result of provoking unifications that fail, which the top-level unification routines need to take account of.

5. UNIFICATION

The types associated with successful unifications of dereferenced leaf node types are shown in Table 1. Some cells contain more than one type since residuations and disjunctions may or may not be reduced in the result term. Note that an :appl type unified with any other type always yields another :appl type. This is a bit misleading, however, since the table does not take into account the effects of dereferencing, which, as we have just described, can provoke a chain of delayed unifications involving any types.

During unification, the evaluation of functions used in predicates and :appl nodes each may produce disjunctive values, but in different ways. Predicates can be evaluated when there is at most one disjunctive argument node, in which case we map the predicate over each of the disjunctions in the disjunctive argument, and collect successful results. If there is more than one successful result, then the result is a disjunction. Alternatively, for functions appearing in :appl nodes only, the function itself may produce a disjunctive value as directed by the internal definition of the function. But note that functions used in F-PATR graphs do not themselves take disjunctive arguments directly, as indicated in the discussion of data types above.

Table 1: Unification for leaf types

	:null	**:atom**	**:disju**	**:appl**	**:r-var**	**:r-dis**
:null	:null	:atom	:disju	:appl	:r-var	:r-dis
:atom	:atom	:atom	:atom	:appl	:atom	:atom :disju :r-dis
:disju	:disju	:atom	:atom :disju	:appl	:atom :disju :r-dis	:atom :disju :r-dis
:appl	:appl	:appl	:appl	:appl	:appl	:appl
:r-var	:r-var	:atom	:atom :disju :r-dis	:appl	:r-var	:atom :disju :r-dis
:r-dis	:r-dis	:atom :disju :r-dis	:atom :disju :r-dis	:appl	:atom :disju :r-dis	:atom :disju :r-dis

There are a number of pairings in Table 1 that are capable of producing either residuated disjunctions, disjunctions, or atoms. These all involve a residuated predicate appearing in at least one of the leaf node arguments. If the initial intersection of the node's contents (independently from residuations) yields a value that still does not provoke evaluation of the predicate, then the result is a residuated disjunction. If the predicate is evaluated, then the unification process may yield an atomic value or a disjunctive value, as explained in the previous paragraph.

Space precludes us from further discussion of the unification algorithms here. With reference to Ait-Kaci and Nasr (1989) and Table 1, however, the details should emerge. See also the examples in the Appendix, which are taken from program output.

6. APPLICATION TO PARSING

The two significant design changes that we have introduced were motivated by our application of F-PATR to parsing of relational set grammars for graphical languages, which is discussed in detail in WWT. Initial experiments adopted the Ait-Kaci/Nasr approach of evaluating the functions of :appl nodes as soon as possible, which meant residuating the argument nodes of these functions. However, this approach led to difficulties in our chart parsing algorithm, which needed to cache the feature structures of active edges before any of the destructive effects of unification involving what we call *expander* functions took place. The root of the issue is that with the Ait-Kaci/Nasr approach, the control of function evaluation is within unification rather than with some external algorithm. In our approach, it was most natural to use external control to implement chart parsing. This point may be clarified by considering an example, for which we need to summarize F-PATR relational set grammars. (See also Wittenburg (1992a 1992b).)

The feature structures for grammatical constituents include the primary attributes *cover*, *syntax*, and *semantics*. The attribute *cover* takes as value a reference to a subset of input objects. This scheme is analogous to HPSG feature structures, where the string-valued *phonology* attribute is replaced by the set-valued *cover* attribute. Rules have the form

```
[mother: [cover: []
          syntax: []
          semantics: []]
 daught1: [cover: []
          syntax: []
          semantics: []]
   . . .
 daughtn: [cover: []
           syntax: []
           semantics: []]]
```

with the condition that for the daughter elements of a rule $D_1...D_n$, there must exist at least one expander relation between covers of each daughter D_i, $2 \leq i \leq n$, and a cover of daughter D_j where $j < i$.

The expander relations are a subclass of relational constraints among sets of input objects used to define the combinatory possibilities of rules. For parsing, the constraints are expressed as functions from cover-sets to cover-sets and appear as a functional value of *cover* attributes.

```
[mother: [syntax: Exp
          cover: (union-covers <2> <3> <4>)
          semantics: (divide <6> <7>)]
 daught1: [syntax: horizontal-line
          cover: 2[]]
 daught2: [syntax: Exp
           cover: 3(what-is-above <2>)
           semantics: <6>]
 daught3: [syntax: Exp
          cover: 4(what-is-below <2>)
          semantics: <7>]]
    (contains-in-x <2> <4>)
    (contains-in-x <2> <3>)
```

The example above is the rule for vertical infixation for fractions, used in a grammar of mathematical notation.

Let us consider now what the feature structure for an active chart-parsing edge for the fraction rule would look like after the first daughter had been unified in. The *cover* attribute would acquire a set-reference value (we will use a number in binary suggestive of the use of bit vectors to represent subsets).

Active edge feature structure:

```
[mother: [syntax: Exp
          cover: (union-covers 0001 <3> <4>)
          semantics: (divide <6> <7>)]
 daught1: [syntax: horizontal-line
          cover: 0001]
 daught2: [syntax: Exp
          cover: 3(what-is-above 0001)
          semantics: <6>]
 daught3: [syntax: Exp
          cover: 4(what-is-below 0001)
          semantics: <7>]]
 (contains-in-x 0001 <4>)
 (contains-in-x 0001 <3>)
```

At this point the Ait-Kaci/Nasr algorithm for unification would provoke the evaluation of the *what-is-above* and *what-is-below* functions, since their arguments are now "grounded". However, this is not what we want for a chart parser since the features of the active edge graph shown here must be kept independent from each of its future advancements. That is, we want to evaluate these two functions at separate cycles in the parsing algorithm at the points when we are ready to extend this edge with the daughters in question. The more conservative approach to derefencing and evaluation of :appl nodes and also the extension of disjunctions as possible values of expander functions provides an elegant solution.[1] The functions *what-is-above* and *what-is-below* will be evaluated in independent *expand* steps of the WWT algorithm. In either case, the function is capable of returning a disjunction of values. But any such values must also meet the constraints of the predicate *contains-in-x*, the application of which may have the effect of reducing the set of val-

1. Hassan Ait-Kaci (personal communication) has pointed out that a solution to the control problem is available in the Le Fun/LIFE framework. An extra unbound argument could be added to expander functions such as *what-is-above* so that evaluation would not be provoked at undesired times. A binding for this extra variable could later be offered when evaluation was wanted.

ues and perhaps eliminating all of them, leading to a unification failure. All this happens as it should with the approach to unification outlined above.

7. CONCLUDING REMARKS

One of the goals of this paper is to bring the work of Ait-Kaci and Nasr to the attention of the computational linguistics community. Their techniques for marrying declarative and functional programming paradigms are an important avenue to explore in expanding the expressiveness of formalisms for linguisic applications. The design issues encountered in building an implementation of F-PATR should be of interest to implementors of such a paradigm. Of course we do not address here issues in the logic of such feature structures or their declarative semantics. The significant differences of F-Patr from Le Fun include an alternative approach to dereferencing certain data types, a change motivated by an environment in which parsing control is outside the unification process, and also an extension to a simple form of disjunction. In contrast to the research projects that implement unification-based grammar formalisms on top of Prolog, this implementation has built a unification environment on top of Lisp. The job of integrating the declarative and functional paradigms is made considerably easier by relying on Lisp for lambda conversion and function evaluation.

In the by now extensive literature on unification grammar frameworks, the current proposal figures as a somewhat conservative, and yet radically expressive, extension to PATR-II. It is conservative in that the logic of feature structures includes only minimal disjunction and no negation or conditionalization. But the extension leads to unlimited expressive power by bringing in the full power of function evaluation. It appears to be an extension appropriate for the representational problems we encountered, but it also has led to unanticipated uses. For example, in writing the semantics for graphical grammars we have been able to use functions in feature structures as a way of building forms that can simply be evaluated to invoke the appropriate operations for applications. Here again, having more control over when evaluation takes place external to the unification process has proved to be important.

There are limitations, however, to the expressive power of F-PATR as it stands. It cannot directly support some of the constraints envisioned in current HPSG literature, for example, because of F-PATR's restrictions on arguments to functional constraints. In HPSG, relations constrain not just atomic values but also general feature structures incuding lists and sets. Such an extension to F-PATR is not planned by the author but it may be of interest. From the logic grammar point of view, the work reported on here may be relevant as a source of ideas for efficiency. Constraints expressed as relations in frameworks such as Zajac (1992) could instead be expressed in F-PATR as compiled functions, leading perhaps to improved runtime speeds.

The MCC/Bellcore implementation of F-PATR includes both destructive and nondestructive versions of unification. The destructive version is, as expected, more straightforward to implement but more expensive computationally given that over copying and early copying are profligate (see Wroblewski 1987). The algorithms for nondestructive unification have been influenced by Tomabechi (1991), but applicative expressions and residuations change the landscape significantly. There tends to be extensive circularity in the data structures: residuated argument nodes point to predicates that in turn point back to their arguments; residuations in applicative-valued nodes point to unification forms that in turn point back to the applicative nodes. There is a need for future work to address issues of space and time efficiency for extensions represented by F-PATR just as there has been such a need for other PATR-II extensions.

A line of research that the author is pursuing currently (Wittenburg 1992b) is to design a more specialized grammar formalism that finesses some of the complexity of residuation and unification through a version of "pseudo-unification" (Tomita 1990). In contrast to residuation, which manages function evaluation at runtime, the idea is to manage the order of evaluation for functional constraints at compile time. In grammar formalisms and parsers under investigation, it is possible for a compiler to order constraints within rule data structures such that evaluation readiness is a deterministic matter, circumventing the need for runtime checks and extra data structures required for delaying evaluation dynamically.

ACKNOWLEDGEMENTS

This research was carried out at MCC under the sponsorship of Bellcore and MCC. Louis Weitzman and Jim Talley worked closely with the author in exploring the use of F-PATR grammars in visual language applications. Roger Nasr was very helpful in consultations on the Le Fun unification algorithms. Thanks to Phil Cannata and Jim Hollan for their support of the project and to the anonymous ACL reviewers for their helpful comments on the manuscript.

REFERENCES

Ait-Kaci, H. (1991) An Overview of LIFE. In J.W. Schmidt and A.A. Stogny (eds.), Next Generation Information System Technology, Proceedings of the 1st International East/West Data Base Workshop, Lecture Notes in Computer Science 504, Springer Verlag, pp. 42-58.

Ait-Kaci, H., and R. Nasr (1989) Integrating Logic and Functional Programming. Lisp and Symbolic Computation 2:51-89.

Balari, S., L. Damas, and G. B. Varile (1989) CLG: Constraint Logic Grammars, Proceedings of the 13th International Conference on Computational Linguistics, Helsinki, vol. 3, pp. 7-12.

Carpenter, B., C. Pollard, and A. Franz (1991) The Specification and Implementation of Constraint-Based Unification Grammars. In Proceedings IWPT 91, Second International Workshop on Parsing Technologies, pp. 143-153.

Damas, L. and G. Varile (1989) CLG: A Grammar Formalism based on Constraint Resolution. In E.M. Morgado and J.P. Martins (eds.), EPIA '89, Lecture Notes in Artificial Intelligence 390, Springer Verlag.

Helm, R., and K. Marriott (1986) Declarative Graphics. In Proceedings of the Third International Conference on Logic Programming, Lecture Notes in Computer Science 225, pp. 513-527. Springer-Verlag.

Helm, R., and K. Marriott (1990) Declarative Specification of Visual Languages. In 1990 IEEE Workshop on Visual Languages (Skokie, Illinois), pp. 98-103.

Kaplan, R., and J. Bresnan (1982) Lexical-Functional Grammar: A Formal System for Grammatical Representation. In J. Bresnan (ed.), The Mental Representation of Grammatical Relations, MIT Press, pp. 173-281.

Kay, M. (1979) Functional Grammar. In Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society.

Pereira, F.C.N., and D. Warren (1980) Definite Clause Grammars for Language Analysis--A Survey of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence 13:231-278.

Pollard, C., and I. Sag (1987) Information-based Syntax and Semantics: Volume 1. Center for the Study of Language and Information.

Reape, M. (1990) Getting Things in Order. In Proceedings of the Symposium on Discontinuous Constituency, Institute for Language Technology and Artificial Intelligence, Tilburg University, The Netherlands, pp. 125-137.

Tomabechi, H. (1991) Quasi-Destructive Graph Unification. In Proceedings IWPT 91, Second International Workshop on Parsing Technologies, pp. 164-171.

Tomita, M. (1990) The Generalized LR Parser/Compiler V8-4: A Software Package for Practical NL Projects. In: COLING-90, Volume 1, 59-63.

Wittenburg, K. (1992a) Earley-style Parsing for Relational Gramars. In Proceedings of IEEE Workshop on Visual Languages, Sept. 15-18, 1992, Seattle, Washington, pp. 192-199.

Wittenburg, K. (1992b) The Relational Language System, Bellcore Technical Memorandum TM-ARH-022353.

Wittenburg, K., L. Weitzman, and J. Talley (1991) Unification-Based Grammars and Tabular Parsing for Graphical Languages. Journal of Visual Languages and Computing 2:347-370.

Wroblewski, D. (1987) Nondestructive Graph Unification. In Proceedings of AAAI 87, pp. 582-587.

Zajac, R. (1992) Inheritance and Constraint-Based Grammar Formalisms. Computational Linguistics 18:159-182.

APPENDIX: Unification Examples

Example 1 :appl with :atom

```
 [ obj: [ length: 25]]
 U [ obj: [ length: (+ 1[] 2[])]
      obj2: [ length: <2>]
      obj1: [ length: <1>]]
 = [ obj: [ length: 1(+ 2[] 3[])]
      obj2: [ length: <3>]
      obj1: [ length: <2>]]
      (Unify <1> 25)
 U [ obj1: [ length: 20]]
 = [ obj: [ length: 1(+ 20 2[])]
      obj2: [ length: <2>]
      obj1: [ length: 20]]
      (Unify <1> 25)
 U [ obj2: [ length: 5]]
 = [ obj: [ length: 1(+ 20 5)]
      obj2: [ length: 5]
      obj1: [ length: 20]]
      (Unify <1> 25)
```

Example 2 :res-var with :atom

```
 [ obj: [ length: 1[]]
   obj1: [ length: 2[]]]
    (EQ <1> <2>)
 U [ obj: [ length: 55]]
 = [ obj: [ length: 55]
      obj1: [ length: 1[]]]
      (EQ 55 <1>)
 U [ obj1: [ length: 55]]
 = [ obj: [ length: 55]
      obj1: [ length: 55]]
```

Example 3 :res-var with :disjunct

```
 [ obj: [ length: 1[]]
   obj1: [ length: 2[]]]
    (EQ <1> <2>)
 U [ obj: [ length: {55 36}]]
 = [ obj: [ length: 1{55 36}]
      obj1: [ length: 2[]]]
      (EQ <1> <2>)
 U [ obj1: [ length: 55]]
 = [ obj: [ length: 55]
      obj1: [ length: 55]]
```

Example 4 :appl with :res-var

```
 [ obj1: [ length: (+ 5 1[])]
   obj2: [ length: <1>]]
```

```
U [ obj1: [ length: 1[]]
    obj2: [ length: []]]
    (>= <1> 54)
= [ obj1: [ length: 1(+ 5 2[])]
    obj2: [ length: <2>]]
    (>= <1> 54)
U [ obj2: [ length: 50]]
= [ obj1: [ length: 1(+ 5 50)]
    obj2: [ length: 50]]
    (>= <1> 54)
U [ obj1: [ length: []]]
= [ obj1: [ length: 55]
    obj2: [ length: 50]]
```

Example 5 :res-var with :res-var

```
[ obj: [ length: 1[]]]
  (<= <1> 60)
U [ obj: [ length: 1[]]]
    (>= <1> 54)
= [ obj: [ length: 1[]]]
    (>= <1> 54)
    (<= <1> 60)
U [ obj: [ length: 55]]
= [ obj: [ length: 55]]
```

Example 6 :res-disjunct with :atom and :disjunct

```
[ obj: [ length: 1{55 43 44}]
  obj1: [ length: 2[]]]
  (EQ <1> <2>)
U [ obj: [ length: {43 55}]]
= [ obj: [ length: 1{55 43}]
    obj1: [ length: 2[]]]
    (EQ <1> <2>)
U [ obj: [ length: 55]]
= [ obj: [ length: 55]
    obj1: [ length: 1[]]]
    (EQ 55 <1>)
U [ obj1: [ length: 55]]
= [ obj: [ length: 55]
    obj1: [ length: 55]]
```

Example 7 :appl with :res-disjunct

```
[ obj1: [ length: (+ 5 1[])]
  obj2: [ length: <1>]]
U [ obj1: [ length: 1{55 43 42}]
    obj2: [ length: 2[]]]
    (> <1> <2>)
= [ obj1: [ length: 1(+ 5 2[])]
    obj2: [ length: <2>]]
    (Unify <1> {55 43 42})
    (> <1> <2>)
U [ obj2: [ length: 50]]
= [ obj1: [ length: 1(+ 5 50)]
    obj2: [ length: 50]]
    (> <1> 50)
    (Unify <1> {55 43 42})
U [ obj1: [ length: []]]
= [ obj1: [ length: 55]
    obj2: [ length: 50]]
```

Example 8 :res-var with :res-disjunct

```
[ obj: [ length: 1{55 43 42}]
  obj1: [ length: 2[]]]
  (EQ <1> <2>)
U [ obj: [ length: 1[]]
    obj1: [ length: 2[]]]
    ((LAMBDA (X Y) (EQ (+ X Y) 110)) <1> <2>)
= [ obj: [ length: 1{55 43 42}]
    obj1: [ length: 2[]]]
    ((LAMBDA (X Y) (EQ (+ X Y) 110)) <1> <2>)
    (EQ <1> <2>)
U [ obj1: [ length: 55]]
= [ obj: [ length: 55]
    obj1: [ length: 55]]
```

Example 9 :appl with :appl

```
[ obj: [ length: (+ 5 1[])]
  obj2: [ length: <1>]]
U [ obj: [ length:
              ((LAMBDA (X) (- (+ X 10) 5)) 1[])]
    obj2: [ length: <1>]]
= [ obj: [ length: 1(+ 5 2[])]
    obj2: [ length: <2>]]
    (Unify <1>
          ((LAMBDA (X) (- (+ X 10) 5)) <2>))
U [ obj2: [ length: 55]]
= [ obj: [ length: 1(+ 5 55)]
    obj2: [ length: 55]]
    (Unify <1> ((LAMBDA (X) (- (+ X 10) 5)) 55))
U [ obj: [ length: []]]
= [ obj: [ length: 60]
    obj2: [ length: 55]]
```

Example 10 :res-disjunct with res-disjunct

```
[ obj1: [ length: 1{55 43 42}]
  obj2: [ length: 2[]]]
  (EQ <1> <2>)
U [ obj1: [ length: 1{45 55 43}]
    obj2: [ length: 2[]]]
    ((LAMBDA (X Y) (EQ (+ X Y) 110)) <1> <2>)
= [ obj1: [ length: 1{55 43}]
    obj2: [ length: 2[]]]
    (EQ <1> <2>)
    ((LAMBDA (X Y) (EQ (+ X Y) 110)) <1> <2>)
U [ obj2: [ length: 55]]
= [ obj1: [ length: 55]
    obj2: [ length: 55]]
```