[From: *Current issues in computational linguistics: in honour of Don Walker*, ed. Antonio Zampolli, Nicoletta Calzolari, Martha Palmer (Linguistica Computazionale, vol. 9-10); Pisa, Dordrecht, [1994]

UD, yet another unification device*

R. Johnson, IDSIA, Lugano
M. Rosner, IDSIA, Lugano
e-mail: mike@idsia.uu.ch

Abstract

This article[1] describes some of the features of a sophisticated language and environment designed for experimentation with unification-oriented linguistic descriptions. The system, called UD, has to date been used successfully as a development and prototyping tool in a research project on the application of situation schemata to the representation of real text, and in extensive experimentation in machine translation.

While the UD language bears close resemblances to all the well-known unification grammar formalisms, it offers a wider range of features than any single alternative, plus powerful facilities for notational abstraction which allow users to simulate different theoretical approaches in a natural way.

After a brief discussion of the motivation for implementing yet another unification device, the main body of the article is devoted to a description of the most important novel features of UD.

1 Introduction

The development of UD arose out of the need to have available a full set of prototyping and development tools for a number of different research projects in computational linguistics, all involving extensive text coverage in several languages: principally a demanding machine translation exercise and a substantial investigation into some practical applications of situation semantics (Rupp, Johnson and Rosner, 1992).

The interaction between users and implementers has figured largely in the development of the system, and a major reason for the richness of its language and environment has been the pressure to accommodate the needs of a group of linguists working on three or four languages simultaneously and importing ideas from a variety of different theoretical backgrounds.

*We thank Suissetra and the University of Geneva for supporting the work reported in this article, and the ACL for granting reproduction rights. We are grateful to all our former colleagues in ISSCO, and to all UD users for their help and encouragement. Special thanks are due to C.J. Rupp for being a willing and constructive guinea-pig, as well as for allowing us to plunder his work for German examples.

[1] This article is a slightly updated version of the authors' "A rich environment for experimentation with unification grammars" that appeared in the Proceedings of ACLE-89, Manchester. At the time of publication, the novelty of the system lay in the fact that it provided a number of experimental features, as described here, in an implementation that was not only freely available but also *efficient,* even by today's standards.

Historically UD evolved out of a near relative of PATR-II (see Shieber, 1984) and its origins are still apparent, not least in the notation. In the course of development, however, UD has been enriched with ideas from many other sources, most notably from LFG (Bresnan, 1982) and HPSG (Sag and Pollard, 1987).

Among the language features mentioned in the article are

- a wide range of data types, including lists, trees and user-restricted types, in addition to the normal feature structures;
- comprehensive treatment of disjunction;
- dynamic binding of pathname segments.

A particular article of faith which has been very influential in our work has been the conviction that well-designed programming languages (including ones used primarily by linguists), should not only supply a set of primitives which are appropriate for the application domain but should also contain *within themselves* sufficient apparatus to enable the user to create new abstractions which can be tuned to a particular view of the data.

We have therefore paid particular attention to a construct which in UD we call a *relational abstraction*, a generalisation of PATR-II templates which can take arguments and which allow multiple, recursive definition. In many respects relational abstractions resemble Prolog procedures, but with a declarative semantics implemented in terms of a typical feature-structure unifier.

1.1 Structure of the article

Section 2 gives a concise summary of the semantics of the basic UD unifier. This serves as a basis for an informal discussion, in Section 3, of our implementation of relational abstractions in terms of 'lazy' unification. The final section contains a few remarks on the issue of completeness, and a brief survey of some other language features.

2 Basic Unifier Semantics

In addition to the usual atoms and feature structures, the UD unifier also handles lists, trees, typed instances, and positive and negative disjunctions of atoms. This section contains the definition of unification over these constructs and employs certain notational conventions to represent these primitive UD data types, as shown in figure 1.

Throughout the description, the metavariables *U* and *V* stand for objects of arbitrary type. Three other special symbols are used:

Type name	**Notation**
atom	ABC
list	$[U\|V]$
n-ary tree	$V_0(V_1, .., V_n)$
+ve disjunction	$/C_1, .., C_r/$
−ve disjunction	$\neg/C_1, .., C_r/$
feature structure	$\{< A_1, V_1 >, .., < A_r, V_r >\}$
typed instance	$< C, \{< A_1, V_1 >, .., < A_n, V_n >\} >$

Figure 1: Notational Conventions

[1]	$\sqcup$ is commutative:	$U \sqcup V = V \sqcup U$
[2]	$\top$ is the identity:	$V \sqcup \top = V$
[3]	$\sqcup$ is $\bot$-preserving:	$V \sqcup \bot = \bot$

Figure 2: Algebraic Properties

$\sqcup$ stands for the unification operator

$\top$ stands for top, the underdefined element.

$\bot$ stands for bottom, the overdefined element that corresponds to failure.

The semantics of unification proper are summarised in figures 2–5: Clauses [1]–[3] define its algebraic properties; clauses [4]–[6] define unification over constants, lists and trees.

In figure 4, clause [7] treats positive and negative disjunctions with respect to sets of atomic values. In figure 5, clause [8] deals with feature structures and *typed instances*. Intuitively, type assignment is a method of strictly constraining the set of attributes admissible in a feature structure.

Any case not covered by [1]–[8] yields $\bot$. Moreover, all the complex type constructors are strict, yielding $\bot$ if applied to any argument that is itself $\bot$.

The extensions to a conventional feature structure unifier described in this section are little more than cosmetic frills, most of which could be simulated in an orthodox, PATR-style environment, even if with some loss of descriptive clarity.

In the rest of the article, we discuss a further enhancement which dramatically and perhaps controversially extends the expressive power of the language.

3 Extending the Unifier

The major shortcoming of typical PATR-style languages is their lack of facilities for defining new abstractions and expressing linguistic generalisations not foreseen (or even foreseeable) by the language designer. This becomes a serious issue when, as in our

[4] constants: $C_1 \sqcup C_2 = C_1$, if $C_1 = C_2$

[5] lists: $[U_1|U_2] \sqcup [V_1|V_2] = [U_1 \sqcup V_1|U_2 \sqcup V_2]$

[6] trees: $U_0(U_1, .., U_n) \sqcup V_0(V_1, .., V_n) =$
$U_0 \sqcup V_0(U_1 \sqcup V_1, .., U_n \sqcup V_n)$

Figure 3: Unification of constants, lists and trees

[7] $/C_1, \ldots, C_n/ \sqcup C = C$
if $C \in \{C_1, \ldots, C_n\}$

$/A_1, \ldots, A_p/ \sqcup /B_1, \ldots, B_q/ = /C_1, \ldots, C_r/$,
if $C_i \in \{A_1, \ldots, A_p\}$ and $C_i \in \{B_1, \ldots, B_q\}$,
$1 \leq i \leq r$, provided (for $r > 0$)

$\neg/C_1, \ldots, C_n/ \sqcup C = C$,
if $C \neq C_i$, $1 \leq i \leq n$

$\neg/A_1, \ldots, A_p/ \sqcup \neg/B_1, \ldots, B_q/ = \neg/C_1, \ldots, C_r/$,
where $C_i \in \{A_1, \ldots, A_p\}$ or $C_i \in \{B_1, \ldots, B_q\}$,
$1 \leq i \leq r$

$/A_1, \ldots, A_p/ \sqcup \neg/B1, \ldots, Bq/ = \neg/C1, \ldots, Cr/$,
where $C_i \in \{A1, \ldots, A_p\}$ and $C_i \notin \{B_1, \ldots, B_q\}$,
$1 \leq i \leq r$

Figure 4: Unification of +ve and −ve atomic value disjunctions

[8]

$$\begin{array}{ll}
\{< A_1, U_1 >, \ldots, < A_p, U_p >\} \sqcup & \\
\{< B_1, V_1 >, \ldots, < B_q, V_q >\} & = \\
\{< A_i, U_i > | A_i \notin B_1, \ldots, B_q\} \cup & \\
\{< B_j, U_j > | B_j \notin A_1, \ldots, A_p\} \cup & \\
\{< A_i, U_i \sqcup V_j > | A_i = B_j\} & \\
1 \leq i \leq p,\ 1 \leq j \leq q &
\end{array}$$

$$\begin{array}{ll}
< C, \{< A_1, U_1 >, \ldots, < A_p, U_p >\} > \sqcup & \\
< C, \{< A_1, V1 >, \ldots, < A_p, V_p >\} > & = \\
< C, \{< A_1, U_1 \sqcup V_1 >, \ldots, < A_p, U_p \sqcup V_p >\} > &
\end{array}$$

$$\begin{array}{ll}
< C, \{< A_1, U_1 >, \ldots, < A_p, U_p >\} > \sqcup & \\
\{< B_1, V_1 >, \ldots, < B_q, V_q >\} & = \\
< C, \{< A_i, U_i > | A_i \notin \{B_1, \ldots, B_q\}\} \cup & \\
\{< A_i, U_i \sqcup V_j > | A_i = B_j\} >, & \\
1 \leq i \leq p,\ 1 \leq j \leq q & \\
(\text{for } \{B_1, \ldots, B_q\} \subseteq \{A_1, \ldots, A_p\}) &
\end{array}$$

Figure 5: Unification of feature structures and typed instances

own case, quite large teams of linguists need to develop several large descriptions simultaneously.

To meet this need, UD provides a powerful abstraction mechanism which is notationally similar to a Prolog *procedure*, but having a strictly declarative interpretation. We use the term *relational abstraction*[2] to emphasise the non-procedural nature of the construct.

3.1 Some Examples of Relational Abstraction

The examples in this section are all adapted from a description of a large subset of German written in UD (Rupp, 1990). As well as relational abstractions, two other UD features are introduced here: a built-in list concatenation operator '++' and generalised disjunction, notated by curly brackets (e.g. {X,Y}). These are discussed briefly in Section 4.

The first example illustrates a relation `Merge`, used to collect together the semantics of an arbitrary number of modifiers in some list X into the semantics of their head Y. Its definition in the external syntax of the current UD version is

```
Merge(X,Y) :
   !Merge-all(X, <Y desc cond>, <Y desc ind>)
```

(The invocation operator '!' is an artefact of the LALR(1) compiler used to compile the external notation - one day it will go away. X and Y should, in this context, be variables over feature structures. The `desc`, `cond` and `ind` attributes are intended to be mnemonics for, respectively, 'description', (a list of) 'conditions' and 'indeterminate'.)

`Merge` is defined in terms of a second relation, `Merge-all`, whose definition is

```
Merge-all([Hd|Tl], <Hd desc cond> ++ L, Ind) :
   Ind = <Hd desc ind>
   !Merge-all(Tl,L,Ind)

Merge-all([],[],Ind)
```

`Merge-all` does all the hard work, making sure that all the indeterminates are consistent and recursively combining together the condition lists.

Although these definitions look suspiciously like pieces of Prolog, to which we are clearly indebted for the notation, the important difference, which we already referred to above, is that the interpretation of `Merge` and `Merge-all` is strictly declarative.

The best examples of the practical advantages of this kind of abstraction tend to be in the lexicon, typically used to decouple the great complexity of lexically oriented descriptions from the intuitive definitions often expected from dictionary coders. As illustration, without entering into discussion of the underlying complexity, for which we

[2]Relational abstractions are comparable, for example, in spirit and in syntax, to parametric sorts in the CUF (see Dörre and Eisele, 1991).

unfortunately do not have space here, we give an external form of a lexical entry for some of the senses of the German verb "tŕaumen".

This is a real entry taken from an HPSG-inspired analysis mapping into a quite sophisticated situation semantics representation. All of the necessary information is encoded into the four lines of the entry; the expansions of `Pref`, `Loctype` and `Subcat` are all themselves written in UD. The feature `-prefix` is a flag interpreted by a separate morphological component to mean that "tŕaumen" has no unstressed prefix and can take 'ge-' in its past participle form.

```
traeumen -prefix
 !Pref(none)
 !Loctype([project])
 !Subcat(np(nom), {vp(inf,squi), pp(von,dat)})
```

`Pref` is a syntactic abstraction used in unraveling the syntax of German separable prefixes. `Loctype` is a rudimentary encoding of *Actionsart*.

`Subcat` contains all the information necessary for mapping instances of verbs with VP or PP complements to a situation schema (Fenstad, Halvorsen, Langholm and van Benthem, 1987; Rupp, Johnson and Rosner, 1992). Here, for completeness but without further discussion, are the relevant fragments of the definition of `Subcat`.

```
Subcat(np(nom), pp(P,C)) :
  !Normal
  !Obl(Pobj,P,C,X)
  !Arg(X,2)
  <* subcat> = [Pobj|T]
  !Assign(T,_)

Subcat(np(nom), vp(F,squi)) :
  !ControlVerb
  !Vcomp(VP,F,NP,Sit)
  !Arg(Sit,2)
  <* subcat> = [VP|T]
  !Assign(T,X)
  F = inf/bse
  !Control(X,NP)

Assign([X], X) :
  <* voice> = active
  !Subj(X)
  !Arg(X,1)

Assign({[Y], []}, Z) :
  <* voice> = passive
  <* vform> = psp
  !Takes(none)
  !Obl(Y,von,dat,Z)
  !Arg(Z,1)
```

4 Implementation of the Extensions

In this section we describe briefly the algorithm used to implement a declarative semantics for relational abstractions, concluding with some remarks on further interesting extensions which can be implemented naturally once the basic algorithm is in place. For the moment, we have only an informal characterisation, but a more formal treatment is in preparation (Johnson and Rupp, forthcoming).

4.1 The solution algorithm

The main problem which arises when we introduce relational abstractions into the language is that some unifications which would ultimately converge may not converge locally (i.e. at some given intermediate stage in a derivation) if insufficient information is available at the time when the unification is attempted (of course some pathological cases may not converge at all—we return to this question below).

We cope with this by defining an argument to the unifier as a *pair* $< I, K >$, consisting of an *information structure* I belonging to one of the types listed in section 2, plus an *agenda* K which holds the set of as yet unresolved constraints potentially holding over I. Unification of two objects,

$$< I_1, K_1 > \sqcup < I_2, K_2 >$$

involves the attempt to resolve the pooled set of constraints $K_1 \cup K_2 = K_0$ with respect to the newly unified information structure $I_0 = I_1 \sqcup I_2$, if it exists.

The question of deciding whether or not some given constraint set will converge locally is solved by a very simple heuristic. First we observe that application of the constraint pool K_0 to I_0 is likely to be non-deterministic, leading to a *set* of possible solutions. Growth of this solution set can be contained locally in a simple way, by constraining each potentially troublesome (i.e. recursively definined) member of K_0 to apply only once for each of its possible expansions, and freezing possible continuations in a new constraint set.

After one iteration of this process we are then left with a set of pairs $\{< J_1, L_1 >, \ldots, < J_r, L_r >\}$, where the L_i are the current constraint sets for the corresponding J_i.

If this result set is empty, the unification fails immediately, i.e. I_0 is inconsistent with K_0. Otherwise, we allow the process to continue, breadth first, *only* with those $< J_i, L_i >$ pairs such that the cardinality of L_i is strictly less than at the previous iteration. The other members are left unchanged in the final result, where they are interpreted as *provisional* solutions pending arrival of further information, for example at the next step in a derivation.

4.2 Decidability

It is evident that, when all steps in a derivation have been completed, the process described above will in general yield a set of information/constraint pairs

$$\{< I_1, K_1 >, \ldots, < I_n, K_n >\}$$

where some solutions are still incomplete—i.e., some of the K_i are not empty. In very many circumstances it may well be legitimate to take no further action—for example where the output from a linguistic processor will be passed to some other device for further treatment, or where one solution is adequate and at least one of the K_i is empty. Generally, however, the result set will have to be processed further.

The obvious move, of relaxing the requirement on immediate local convergence and allowing the iteration to proceed without bound, is of course not guaranteed to converge at all in pathological cases. Even so, if there exist some finite number of complete solutions our depth first strategy is guaranteed to find them eventually. If even this expedient fails, or is unacceptable for some reason, the user is allowed to change the environment dynamically so as to set an arbitrary depth bound on the number of final divergent iterations. In these latter cases, the result is presented in the form of a feature structure annotated with details of any constraints which are still unresolved.

4.3 Discussion

Designers of unification grammar formalisms have tended to avoid including constructs with the power of relational abstraction, presumably through concern about issues of completeness and decidability. We feel that this is an unfortunate decision in view of the tremendous increase in expressiveness which these constructs can give. (Incidentally, they can be introduced, as in UD, without compromising declarativeness and monotonicity, which are arguably, from a practical point of view, more important considerations.) On a more pragmatic note, UD has been run without observable error on evolving descriptions of substantial subsets of French and German, and it has been rarely necessary to intervene on the depth bound, which defaults to zero.

In practice, users seem to need the extra power very sparingly, perhaps in one or two abstractions in their entire description, but then it seems to be crucially important to the clarity and elegance of the whole descriptive structure (list appending operations, as in HPSG, for example, may be a typical case).

4.4 Other extensions

Once we have a mechanism for 'lazy' unification, it becomes natural to use the same apparatus to implement a variety of features which improve the habitability and expressiveness of the system as a whole. Most obviously we can exploit the same framework of local convergence or suspension to support efficient hand-coded versions of some basic primitives like list concatenation and non-deterministic extraction of elements from arbitrary list positions. This has been done to advantage in our case, for example, to facilitate importation of useful ideas from, *inter alia* HPSG and JPSG (Gunji, 1987). We have also implemented a fully generalised disjunction (as opposed to the atomic value disjunction described in section 2) using the same lazy strategy to avoid exploding alternatives unnecessarily. Similarly, it was quite simple to add a treatment of underspecified pathnames to allow simulation of some recent ideas from LFG (Kaplan, Maxwell and Zaenen, 1987).

4.5 Current state of the system

The system has now been in a stable state for some years, and supports substantial fragments of German French and Italian. A derivative, ELU, specialised for machine translation applications, has been built at ISSCO, Geneva (see Estival, 1990).

There is also a rich user environment, of which space limitations preclude discussion here, including tracing and debugging tools and a variety of interactive parameterisations for modifying run-time behaviour and performance. The whole package runs on any Unix platform which supports Allegro Common Lisp.

References

[1] Bresnan J., ed. *The Mental Representation of Grammatical Relations,* Cambridge, Ma.:MIT Press, 1982.

[2] Dörre, J. and A. Eisele. "A comprehensive unification-based grammar formalism", DYANA deliverable R3.1.B, Centre for Cognitive Science, University of Edinburgh, Scotland, January 1991.

[3] Estival, D. "ELU user manual", Technical Report, ISSCO, University of Geneva, 1990.

[4] Fenstad J-E., P-K. Halvorsen, T. Langholm and J. van Benthem, *Situations, Language and Logic,* Reidel, 1987.

[5] Gunji T., *Japanese Phrase Structure Grammar,* Reidel, 1987.

[6] Johnson, R. and C. J. Rupp. "Evaluating complex constraints in linguistic formalisms", In Trost, H., editor, *Feature Formalisms and Linguistic Ambiguity.* Ellis Horwoood, Chichester, 1993. To appear.

[7] Kaplan R., J. Maxwell and A. Zaenen, "Functional Uncertainty", in CSLI Monthly, January 1987.

[8] Sag I. and C. Pollard, "Head-Driven Phrase Structure Grammar: an Informal Synopsis", CSLI Report no.CSLI-87-79, 1987.

[9] Rupp, C. J. *Semantic Representation in a Unification Environment*, PhD thesis, University of Manchester, 1990.

[10] Rupp, C.J., R. Johnson and M. Rosner, "Situation schemata and linguistic representation", in M. Rosner and R. Johnson (eds.), *Computational Linguistics and Formal Semantics,* Cambridge:Cambridge University Press, 1992.

[11] Shieber S., "The design of a computer language for linguistic information", Proceedings of Coling 84, Stanford, 1984.