A rich environment for experimentation with unification grammars

R. Johnson & M. Rosner

IDSIA, Lugano

ABSTRACT

This paper describes some of the features of a sophisticated language and environment designed for experimentation with unification-oriented linguistic descriptions. The system, which is 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 paper is devoted to a description of the most important novel features of ud.

The paper concludes with a discussion of some questions of implementation and completeness.

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 (Johnson, Rosner and Rupp, forthcoming).

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.

Historically ud evolved out of a near relative of PATR-II (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 paper 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 path-name 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 paper

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 deals with lists, trees, feature structures, 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, and juxtaposed integers are intended to be read as subscripts.

Three other special symbols are used:

+ stands for the unification operator

* stands for top, the underdefined element.

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

The semantics of unification proper are summarised in figures 2 - 4. Clauses [1] - [3] define its algebraic properties; clauses [4] - [6] define unification over constants, lists and trees in a manner analagous to that found in Prolog.

In figure 4, clause [7] treats positive and negative disjunctions with respect to sets of atomic values. 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] to [8] yields #. Moreover, all the complex type constructors are strict, yielding # if applied to any argument that is itself #.

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 a standard PATR environment, even if with some loss of descriptive clarity.

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

Type name	Notation
constant	A B C
list	[U ¦ V]
n-ary tree	V0(V1,..,Vn)
+ve disjunction	/C1,..,Cr/
-ve disjunction	~/C1,..,Cr/
feature structure	{<A1,V1>,..,<Ar,Vr>}
typed instance	<C,{<A1,V1>,..,<An,Vn>}>

figure 1 : Notational Conventions

[1] + is commutative:

U + V = V + U

[2] * is the identity:

V + * = V

[3] + is #-preserving:

V + # = #

figure 2 : Algebraic Properties

[4] unification of constants:

C1 + C2 = C1, if C1 = C2

[5] unification of lists:

[U1¦U2] + [V1¦V2] = [U1+V1¦U2+V2]

[6] unification of trees:

U0(U1,..,Un) + V0(V1,..,Vn) = U0+V0(U1+V1,..,Un+Vn)

figure 3 : Constants, Lists and Trees

[7] disjunction:

```
/C1,..,Cn/ + C        = C, if C in {C1,...,Cn}

/A1,..,Ap/ + /B1,..,Bq/
                      = /C1,..,Cr/, if Ci in {A1,..,Ap}
                                    and Ci in {B1,..,Bq},
                                    1<=i<=r, r > 0

~/C1,..,Cn/ + C       = C, if C ~= Ci, 1<=i<=n

~/A1,..,Ap/ + ~/B1,..,Bq/
                      = ~/C1,..,Cr/, where Ci in {A1,..,Ap}
                                     or Ci in {B1,..,Bq},
                                     1<=i<=r
/A1,..,Ap/ + ~/B1,..,Bq/
                      = ~/C1,..,Cr/, where Ci in {A1,..,Ap}
                                     and Ci not in {B1,..,Bq},
                                     1<=i<=r
```

[8] feature structures:

```
{<A1,U1>,..,<Ap,Up>} + {<B1,V1>,..,<Bq,Vq>}
                      = {<Ai,Ui> | Ai not in {B1,..,Bq}} union
                        {<Bj,Uj> | Bj not in {A1,..,Ap}} union
                        {<Ai,Ui+Vj> | Ai = Bj},
                        1<=i<=p, 1<=j<=q}

<C,{<A1,U1>,..,<Ap,Up>}> + <C,{<A1,V1>,..,<Ap,Vp>}>
                      = <C,{<A1,U1+V1>,..,<Ap,Up+Vp>}>

<C,{<A1,U1>,..,<Ap,Up>}> + {<B1,V1>,..,<Bq,Vq>}
                      = <C,{<Ai,Ui> | Ai not in {B1,..,Bq}}
                           union {<Ai,Ui+Vj> | Ai = Bj}>,
                           if all Bj in {A1,..,Ap},
                           where  1<=i<=p, 1<=j<=q
```

figure 4 : Atomic Value Disjunctions and Feature Structures

3. Extending the Unifier

One of the major shortcomings 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 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 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 by C.J. Rupp. 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 unfortunately do not have space here, we give an external form of a lexical entry for some of the senses of the German verb traeumen.

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 merely a flag interpreted by a separate morphological component to mean that traeumen 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).

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.

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 which holds the set of as yet unresolved constraints K which potentially hold over I. Unification of two objects,

$$\langle I1,K1\rangle + \langle I2,K2\rangle$$

involves the attempt to resolve the pooled set of constraints

$$K1 \cup K2 = K0$$

with respect to the newly unified information structure I0 = I1 + I2, 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 K0 to I0 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 K0 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 {<J1,L1>,...,<Jr,Lr>}, where

the Li are the current constraint sets for the corresponding Ji.

If this result set is empty, the unification fails immediately, i.e. I0 is inconsistent with K0. Otherwise, we allow the process to continue, breadth first, only with those <Ji,Li> pairs such that the cardinality of Li 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 {<I1,K1>...<InKn>} where some solutions are still incomplete - i.e. some of the Ki 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 Ki 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.

4.2.1. Discussion

Designers of unification grammar formalisms typically 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 running now without observable error for almost a year on descriptions of substantial subsets of French and German, and we have only once had to intervene on the depth bound, which defaults to zero (this was when someone tried to use it to run Prolog programs).

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).

4.3. 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 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 oppposed 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 under-specified pathnames to allow simulation of some recent ideas from LFG (Kaplan, Maxwell and Zaenen, 1987).

5. Current state

The system is still under development, with a complete parser and rudimentary synthesiser, plus a full, reversible, morphological component. We are now working on a more satisfactory generation component, as well as tools - such as bi/multi-lingual lexical access and transfer - specifically crafted for use in machine translation research. Substantial fragments of German and French developed in ud are already operational.

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 Suns, and we have begun to work on portability to other lisp/unix combinations.

References

Bresnan J (ed) (1982). The Mental Representation of Grammatical Relations. MIT Press.

Fenstad J-E, P-K Halvorsen, T Langholm and J van Benthem (1987). Situations, Language and Logic. Reidel.

Gunji T (1987). Japanese Phrase Structure Grammar. Reidel.

Johnson R, M Rosner and C J Rupp (forthcoming). 'Situation schemata and linguistic representation'. In M Rosner and R Johnson (eds). Computational Linguistics and Formal Semantics. Cambridge University Press (to appear in 1989).

Kaplan R, J Maxwell and A Zaenen (1987). 'Functional Uncertainty'. In CSLI Monthly, January 1987.

Sag I and C Pollard (1987). Head-Driven Phrase Structure Grammar: an Informal Synopsis. CSLI Report # CSLI-87-79.

Shieber S (1984). 'The design of a computer language for linguistic information'. Proceedings of Coling 84.

Acknowledgements

We thank the Fondazione Dalle Molle, Suissetra and the University of Geneva for supporting the work reported in this paper. 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.