CUF in context

Suresh Manandhar*
Human Communication Research Centre
University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
U.K.

1 Introduction

CUF (see [DD93]) is one amongst the current breed of typed feature logic based formalisms which are principally targeted at providing a computational system for implementing HPSG grammars. We provide a theory neutral comparison of CUF against the formalisms TFS [Zaj92][Zaj91] and ALE [Car93] drawing out the similarities and differences. Our comparison will be concentrated mainly on the expressivity of the *type system*, *definite clauses* and *control scheme* which provide additional control over the SLD resolution engine.

2 Type System

The HPSG grammar formalism is based around the idea of employing a (possibly extended) typed feature language for the representation of grammatical knowledge. This is a departure from the rule based approach of other grammar formalisms such as GPSG in which context free rules play a major part in grammar specification. This means that in HPSG possibly recursive type definitions are employed to specify grammatical principles. However the potential recursive nature of type definitions that express grammatical principles is a major source of inefficiency within current typed feature formalisms. This is due to the fact that consistency checking of recursive type definitions is in general *undecidable*.

The fundamental nature of the above problem is the principal reason why different formalisms employ radically different approaches for dealing with it. In order to differentiate clearly between the different formalisms we shall fix some terminology in the following.

Type definitions provide a mechanism for storing definitions of, for instance, lexical items or definitions of concepts. For example, we may want to store the definition that *nouns* have [CAT n] specified. A *type definition* such as the following can be employed:

(1) $nouns = cat : n$

*This work was carried out while the author was supported under the ESPRIT project LRE-061-061. I am specially thankful to Chris Brew for many helpful discussions and comments. Thanks also to Jochen Dörre for helping me understand CUF and to Marc Moens for reading earlier drafts.

The symbol *nouns* is usually referred to as a *type symbol*.

Thus instead of writing [CAT n] in other parts of the grammar specification the grammar writer simply re-uses the symbol *nouns* instead.

One dimension along which typed feature formalisms can be classified is according to the expressivity of the type definition mechanism.

We say that a type system is *general* if there are *no* restrictions imposed on the right hand side of a type definition *i.e.* if at least arbitrary conjunctions of type symbols, feature selection and variables are permitted. On the other hand, we say that the type system is *restricted* if this is not the case. The particular design choice taken by a given formalism has a fundamental impact both on the way HPSG grammars have to be specified and on the design of the rest of the formalism.

The type system of TFS is general since it permits conjunctions of typed feature terms containing variables in its type definitions. Furthermore TFS permits multiple definitions for the same type symbol which are interpreted disjunctively. On the other hand, both ALE and CUF provide restricted type systems. In ALE and CUF, type definitions are restricted to conjunctions of typed feature terms of the form $f : s$ where f is a feature symbol and s is a type symbol in ALE or a boolean expression over types in CUF. Moreover, in ALE types are placed into a hierarchy using ordering statements $s \leq t$ between types. In CUF these statements are generalized to arbitrary boolean expressions which are stated as axioms ($\leq$ can be thought of as implication). However the lack of variables in in the type system of both ALE and CUF makes them quite different from a language like TFS. In particular, this makes it impossible to define, for instance, the HPSG *subcategorisation principle* or the *head feature principle* ([PS87] *pp.* 148) as a type, since they require the use of variables.

2.1 Feature Typing

While type definitions provide a definitional mechanism for type symbols, feature typing is another mechanism that is employed by typed feature formalisms to restrict the possible interpretation of feature symbols and hence provide typing on feature symbols. Usually, feature symbols are interpreted as unary partial functions $f^{\mathcal{I}} : \top^{\mathcal{I}} \rightharpoonup \top^{\mathcal{I}}$. However with feature typing statements such as:

(2) $\quad f : s \longrightarrow t$

the feature symbol f is interpreted as a total function $f^{\mathcal{I}} : s^{\mathcal{I}} \longrightarrow t^{\mathcal{I}}$.

Within systems such as TFS, ALE and CUF, feature typing information is automatically extracted from a given set of type definitions. Thus given the type definitions:

(3)
$$\begin{aligned} agreement = \; & num : number \sqcap \\ & per : person \\ number = \; & sg \sqcup pl \\ person = \; & 1 \sqcup 2 \sqcup 3 \end{aligned}$$

the feature typing specifications:

(4)
$$\begin{aligned} & num : agreement \longrightarrow number \\ & per : agreement \longrightarrow person \end{aligned}$$

are automatically generated.

The ALE type system is restricted in such a way that every feature symbol has to be declared at a unique most general type. However the same feature can be redeclared on more specific type symbols. Thus the following is a valid ALE type definition (modulo ALE specific syntax):

(5)
$$\begin{aligned}
&agreement = \quad num : number \sqcap \\
&\qquad\qquad\qquad\; per : person \\
&sg \leq number \\
&pl \leq number \\
&1 \leq person \\
&2 \leq person \\
&first_sg \leq agreement \\
&first_sg = \quad num : sg \sqcap \\
&\qquad\qquad\quad\; per : 1
\end{aligned}$$

This generates the following feature typing specifications which are interpreted conjunctively:

(6)
$$\begin{aligned}
&num : agreement \longrightarrow number \\
&per : agreement \longrightarrow person \\
&num : first_sg \longrightarrow sg \\
&per : first_sg \longrightarrow 1
\end{aligned}$$

In fact the simplicity of the ALE type discipline means that type definitions can simply be treated as infix notation for feature typing specifications.

However the above simple strategy does not quite work within CUF and TFS since both formalisms do not impose any restrictions on where a feature may be declared. Thus the same feature symbol may occur within two different type definitions. For instance, in addition to the specification of the *agreement* type in (3) the following type definition may be specified.

(7)
$$\begin{aligned}
&quantity = \quad weight : integer \sqcap \\
&\qquad\qquad\quad\; per : measure \\
&measure = lb \sqcup kg
\end{aligned}$$

CUF combines both type definitions and feature typing definitions into a compact syntax which is called *feature declarations*. Similarly ALE's *appropriateness declarations* achieve the same effect. However, we have chosen to make explicit the distinction between type definitions which provide definitions for type symbols and feature typing which provide typing information for feature symbols. This we believe provides a clearer understanding of the semantics of typed feature formalisms.

According to the specification in (7) the following feature typing can be assumed.

(8) $per : quantity \longrightarrow measure$

If we further assume that the type symbols *quantity* and *agreement* are disjoint then (assuming a conjunctive interpretation of feature typing specifications) leads to an inconsistent definition of both *agreement* and *quantity* types.

A fix to the above problem can be obtained as follows.

Let $f^{\;\phi}$ denote the function obtained by restricting the function f to the members of the set ϕ. Then a restricted interpretation of feature typing can be obtained by assuming feature typing specifications of the form:

(9) $\quad f : s \longrightarrow t$

which is interpreted as:

(10) $\quad f^{\;s^{\mathcal{I}}} : s^{\mathcal{I}} \longrightarrow t^{\mathcal{I}}$

Thus the behaviour of the function now is dependent on the *domain* type symbol. Now we can interpret the feature *per* in examples (3) and (7) by the following feature typing definitions:

(11) $\quad per : agreement \longrightarrow person$

(12) $\quad per : quantity \longrightarrow measure$

which are interpreted conjunctively. The definition in (11) states that the range of the function $per^{\mathcal{I}}$ when its domain is restricted to the set $agreement^{\mathcal{I}}$ is $person^{\mathcal{I}}$. On the other hand, the definition in (12) states that the range of the function $per^{\mathcal{I}}$ when its domain is restricted to $quantity^{\mathcal{I}}$ is $measure^{\mathcal{I}}$.

2.2 Propositional Constraints

A propositional theorem prover is employed in the Prolog-III language [Col87] to provide consistency checking of propositional formulas involving just constant symbols. This is an efficient alternative to the depth-first search strategy of Prolog's SLDNF resolution engine which otherwise generates a large search space. Since grammatical descriptions often employ propositional formulas, e.g. see (13), a propositional theorem prover is an important requirement for efficient execution of HPSG grammars.

(13)
$$\begin{aligned} & \ldots \sqcap case : (case \sqcap \neg gen) \\ case = \; & nom \sqcup acc \sqcup gen \sqcup dat \end{aligned}$$

Following the incorporation of such a propositional theorem prover in the STUF formalism [DS91], the current CUF formalism also employs a propositional theorem prover.

In addition CUF permits inheritance specifications and disjointness specifications between type symbols which meshes cleanly with both the operational and declarative semantics of propositional type symbols.

Thus in CUF one is allowed to write type definitions of the following forms.

(14) $\quad case = nom \sqcup acc \sqcup dat$

(15) $\quad case = nom \mid acc \mid dat$

(16)
$$\begin{aligned} & nom \leq case \\ & acc \leq case \\ & dat \leq case \end{aligned}$$

(17) $nom \mid acc \mid dat \leq case$

Each of the above definitions have a different interpretation (see [DD93]) for a description of the CUF semantics) and only (15) captures precisely the intended specification. Apart from the above compact syntax, disjointness specifications can be defined in isolation too, as in the following example:

(18) $nom \mid acc \mid dat$

which states that *nom*, *acc* and *dat* are disjoint types. In this particular example, disjointness comes for free if *nom*, *acc* and *dat* were defined as CUF *constant types*. However note that in the above examples *nom, acc* and *dat* could equally have definitions attached to them. Thus *nom* could be additionally defined by:

(19)
$$
\begin{array}{ll}
nom = nom1 \; ; \; nom2. & \\
nom1 = & case : nominative, \\
 & role : sub, \\
 & voice : active. \\
nom2 = & case : nominative, \\
 & role : \neg sub, \\
 & voice : passive.
\end{array}
$$

An interesting feature of the CUF type checking machinery is that it does not create two choice points for the above two definitions of *nom*. Instead CUF relies on a lazy constraint propagation machinery that eliminates disjunctive choices incrementally as new constraints get added. Thus if $voice : active$ is known then the possibility of $role : \neg sub$ can be eliminated without actually having to add $role : sub$ explicitly.

This mechanism cleanly integrates a propositional theorem proving component with the feature typing component in a fairly efficient manner.

2.3 Consequences for the Grammar Writer

While the availability of all the propositional connectives in the CUF type system provides great flexibility for the grammar writer, the lack of variables in the type definitions means that a fair proportion of a realistic grammar cannot be expressed within the type system. For instance, it is difficult to encode HPSG lexical entries in the type system since they often require variable co-references *e.g.* variable coreferences between the values of the SEMANTICS attribute of the verb and the subcategorised noun, variable coreferences between a subcategorised item and a SLASH value.

Thus some *a priori* decisions have to be made regarding which parts of the grammars are to be written within the type system and which parts are to be relegated to the definite clause component. In [ED90] it is suggested that both the lexicon and the HPSG grammatical principles are to be specified via the definite clause mechanism and the rest of the grammar is to be specified within the type system.

3 Definite Clauses

For specifying HPSG grammars yet another seemingly different expressive device is employed - *relational dependencies*. Relational dependencies are employed to state HPSG grammatical principles such as the *subcategorisation principle* stated below.

(20) **Subcategorisation Principle**

$$\begin{bmatrix} \text{SYN}|\text{LOC}|\text{SUBCAT } Y & \\ \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR}|\text{SYN}|\text{LOC}|\text{SUBCAT} & Z \\ \text{COMP-DTRS} & X \end{bmatrix} \end{bmatrix} : -append(X, Y, Z)$$

The *append* relation is known as a *relational dependency*. It can be defined as given below in (21).

(21)
$$\begin{array}{l} append(nil, X, X). \\ append(head : X \sqcap tail : R, Y, head : X \sqcap tail : Z) : - \\ \quad append(R, Y, Z). \end{array}$$

However, if a general type system is available relational dependencies can be translated into a type definition as shown in (22).

(22)
$$\begin{array}{rl} append = & (arg1 : nil \sqcap arg2 : X \sqcap arg3 : X) \\ append = & (arg1 : (head : X \sqcap tail : R) \sqcap \\ & arg2 : Y \sqcap \\ & arg3 : (head : X \sqcap tail : Z) \\ & tmp : (append \sqcap \\ & arg1 : R \sqcap \\ & arg2 : Y \sqcap \\ & arg3 : Z)). \end{array}$$

Thus, given a general type system relational dependencies can be treated simply as syntactic sugar. This means that in a formalism such as TFS, relational dependencies do not provide additional expressivity other than that already provided by the type system.

However, neither ALE nor CUF provide a general type system and hence needs to provide an additional expressive device for specifying HPSG relational dependencies. We shall call this additional mechanism the *definite clause* mechanism. The choice of a different terminology here is mainly an indication of the different operational realisation of the definite clause component from that of the type system.

Both ALE and CUF provide a mechanism to specify relational dependencies *via* their definite clause component which is treated as being separate from the type system. The idea is to provide a Prolog-like language in which first-order terms are replaced by typed feature terms which have to respect the type definitions that are specified. From a foundational perspective, such a system can be viewed as an instantiation of the CLP(Σ) scheme where Σ is to be identified with the equational theory of typed feature terms [HS88] [JL87].

Although from the above discussion both ALE and CUF appear similar with respect to their definite clause component there is a fundamental difference between the CUF definite clause component and the ALE definite clause component which makes it impossible to execute realistic HPSG grammars in ALE. This has to do with the operational semantics of definite clauses which we examine next.

4 Controlling resolution

The depth first nature of the Prolog SLD-resolution engine is the result of employing a *selection function* that selects the *leftmost* goal for resolution. This choice of the selection function is just an instance of the declarative formulation of the SLD-resolution scheme which does not commit to any particular selection function. However the selection function plays a crucial role in determining the operational behaviour of systems based on SLD-resolution which includes both Prolog and typed feature formalisms.

Due to the highly declarative nature of HPSG grammars a Prolog-like resolution is strategy is inappropriate for executing HPSG grammars since grammar specifications do not encode the processing order of different type symbols.

In this respect ALE differs from both CUF and TFS. ALE employs a Prolog-like resolution strategy and goes all the way in providing both *cuts* and *negation-by-failure*. Although this provides an interesting and fairly powerful programming language in its own right, executing declaratively specified HPSG grammars is not feasible within ALE. However ALE provides a DCG backbone that enables specification of typed unification grammars within which it may very well be possible to reformulate all HPSG grammatical rules and principles.

CUF differs from Prolog in that it employs a selection function that is based on *deterministic closure*. A literal is called *deterministic* with respect to a program P, if it unifies at most with one head of the definite clauses in P. The selection function employed by CUF then selects deterministic goals whenever this is possible falling back to a Prolog-like strategy when every goal to be reduced is non-deterministic. This simple strategy results in a drastically reduced search space since those non-deterministic branches that can be pruned with the aid of deterministic information are eliminated.

TFS also employs a delaying scheme that delays goals until the variable (upon which the goal is delayed) is sufficiently instantiated (see [Zaj92]) resorting to a Prolog-like strategy when all goals are delayed.

4.1 Delay Statements

Finer control of the resolution strategy is achieved in CUF by employing *delay statements* defined by the user. Delay statements in CUF state the features which need to be instantiated in order that the predicate can be expanded otherwise the resolution engine delays this predicate.

However when all predicates are blocked a Prolog like strategy is employed to select the next goal. This *fall back* position is essential in order to guarantee the completeness of the resolution engine.

5 Summary

In summary, TFS provides a rapid prototyping environment for HPSG grammars in a high-level declarative language that is as close as one could possibly get to the original HPSG definitions.

CUF on the other hand also provides an excellent prototyping environment for HPSG grammars. Of the two, CUF has the potential of providing better performance. However, CUF requires some *ad hoc* choice from the grammar writer on whether a given definition is to be stated *via* the type system or the definite clause component. This choice is motivated by whether the given specification contains variable co-references or not. Thus (for realistic grammars) lexical entries have to specified

Figure 1: ALE, TFS and CUF at a glance

as definite clauses losing some of the advantages of hierarchical lexical organisation. Grammatical principles too need to be specified as definite clauses but as [ED90] has shown this fits fairly cleanly within the CUF architecture.

ALE on the other hand is oriented towards efficiency and would be a good vehicle for implementing large grammars. Furthermore, ALE is the only system (amongst the three) that provides excellent support for context-free grammars.

In effect TFS, CUF and ALE represent 3 different design philosophies for implementing typed feature formalisms (*see* fig. 1).

The philosophy behind TFS is to stay as close as possible to the high-level needs of HPSG grammars while at the same time achieving reasonable efficiency. The philosophy behind CUF seems to be to achieve maximum efficiency by sacrificing as little at possible in its ability to specify HPSG grammars in a high-level language. Finally the philosophy behind ALE seems to be provide a minimal typed extension to DCGs in order to enable specification of HPSG-like grammars thereby achieving maximum efficiency.

References

[Car93] Robert Carpenter. ALE:Attribute Logic Engine Users Guide, Version β. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[Col87] A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), August 1987.

[DD93] Jochen Dörre and Michael Dorna. CUF: A Formalism for Linguistic Knowledge Representation. Dyana deliverable, 1993. This Volume.

[DS91] Jochen Dörre and Roland Seiffert. Sorted feature terms and relational dependencies. IWBS Report 153, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, Germany, February 1991.

[ED90] Andreas Eisele and Jochen Dörre. A comprehensive unification formalism. Dyana internal memo, Institut für maschinelle Sprachverarbeitung, University of Stuttgart, September 1990.

[HS88] Markus Höhfeld and Gert Smolka. Definite Relations over Constraint Languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, Germany, October 1988. Accepted for the Journal of Logic Programming.

[JL87] J. Jaffar and J-L Lassez. Constraint Logic Programming. In *14th ACM Symposium on Principles of Programming Languages (POPL-87)*, pages 111–119. Association for Computing Machinery, Munich, 1987.

[PS87] Carl Pollard and Ivan Sag. *Information based syntax and semantics, VOL I.* CSLI Lecture Notes. CSLI, 1987.

[Zaj91] Rémi Zajac. Notes on the Typed Feature System. Draft: January 1991 Project Polygloss, University of Stuttgart, Germany, 1991.

[Zaj92] Rémi Zajac. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182, 1992.