Conditional Constraints in a CLP-based HPSG Implementation

Johannes Matiasek

Austrian Research Institute for Artificial Intelligence*
Schottengasse 3, A-1010 Vienna, Austria
john@ai.univie.ac.at

Abstract. We present a CLP-based method for the direct implementation of HPSG. We interpret unification of typed feature structures under the restrictions of principled constraints as constraint solving in the CLP paradigm. Principles of grammar, viewed as licensing conditions applying to every node of a feature structure, take the form of conditional constraints. The constraint solving mechanism, implemented in a variant of SICStus Prolog allowing for unification extensions, takes care of delaying the enforcement of conditional constraints on insufficiently instantiated feature structures and thus provides a general means to specify delay conditions. Finally, conditionals allow for reformulation of disjunctions into a conjunction of implications, thereby reducing the search space.

Zusammenfassung. In diesem Beitrag wird eine Methode zur direkten Implementierung von HPSG vorgestellt. Unifikation typisierter Merkmalsstrukturen unter den Einschränkung der Grammatik wird als Constraint solving im CLP Paradigma interpretiert. Die Prinzipien der Grammatik werden als konditional formulierte Lizensierungsbedingungen gesehen, die für jeden Knoten einer Merkmalsstruktur anzuwenden sind. Die Implementation (in einer Variante von SICStus Prolog, die Erweiterungsmöglichkeiten der syntaktischen Unifikation vorsieht) verzögert die Anwendung von Implikationen auf unzureichend instantiierte Strukturen. So wird eine generelle Methode für die Formulierung von Verzögerungsbedingungen erreicht. Abschließend wird gezeigt, wie disjunktive Bedingungen in eine Konjunktion solcher Implikationen umformuliert werden können, wodurch eine Reduktion des Suchraumes erreicht werden kann.

1 Introduction

The development of linguistic theories during the past decade exhibits a trend to replace the view of language as a set of rule-generated strings by a view that defines language by giving general principles, constraining the set of linguistic structures. This trend, initiated by the Principles and Parameters Approach to syntax ([3]), is also reflected in Head driven Phrase Structure Grammar (HPSG, [9], [10]). HPSG is a unification based grammar formalism using feature structures to model linguistic entities. Being a lexically based grammar formalism these feature structures originate mainly from the lexicon. How these feature

* Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian *Ministry of Science and Research*

structures can be combined to form linguistically valid structures is constrained by the principles of grammar. These constraints may involve relational dependencies between parts of feature structures.

A feature formalism to be used for implementing HPSG has to provide a means for handling conditional constraints (as most principles are stated conditionally) as well as for handling relational dependencies. Two requirements are crucial for these mechanisms to be useful for grammar *specification* and not simply for a (processing direction dependent) process model:

1. To admit only feature structures compliant with all principles of grammar, these mechanisms must be part of the feature unification formalism itself.
2. To be operational independently of the instantiation order of a feature structure, they must be able to cope with structures not fully instantiated.

We will show how these requirements can be met by interpreting unification of typed feature structures as constraint solving in the CLP paradigm.

The outline of the paper is as follows: first we will give a short overview of HPSG, focussing on the way principles are stated and on the relational dependencies other than structure sharing that might be induced by them. Then we will shortly sketch the CLP facilities of the Prolog system underlying our implementation. The main part of the paper shows how conditional constraints are interpreted by our system, and how this interpretation leads to the implementation of a delay mechanism. Finally we will discuss the possibilities to eliminate disjunctions which are offered by our system.

2 HPSG—A Constraint based Theory

HPSG is a unification based grammar formalism modeling linguistic objects by means of typed feature structures. It differs from other unification based theories (such as LFG, cf. [7]) in that it does not use feature structures as an additional device augmenting traditional phrase structure rules. Instead, HPSG uses (universal and language-specific) *Principles* to constrain the feature structures admissible for a particular grammar. Furthermore, HPSG formally restricts the space of admissible feature structures representing linguistic objects.

2.1 Well-typing Constraints

Feature structures of HPSG are *strongly typed*; every feature structure must be associated with a type. Every type restricts its associated feature structure by allowing only certaion features and restricting the type of their values. These appropriateness and value restrictions are inherited along the type hierarchy.

Since HPSG requires feature structures to be well-typed, there is a need to compute well-typed feature structures from arbitrary ones (excluding those which are not typeable). This *Type Inference* procedure is defined, relying on the unique introduction of features and their monotonic inheritance along the type hierarchy. *Well-typed Unification* is definable by interleaving unification and type inference. It exhibits desirable monotonicity properties, which are important in connection with lazy instantiating implementations. A formal characterization of these properties can be found in [2] and [1].

The advantages of using typed instead of plain feature structures are obvious. Unification failure can often be detected early by inspecting only the types, without creating expensive copies during structural unification. Furthermore, types in many cases allow a more natural and efficient encoding of linguistic information than this would be the case with untyped structures.

2.2 Principles of Grammar

The second source of constraints on feature structures are the principles of grammar. [9] allowed general implicative and negative constraints in the form of conditional feature structures, interpreted via the relative pseudo-complement operation in a Heyting algebra. In [10] principles are given only verbally.

Recent work on formalizing the basis of HPSG (e.g., [2]) models them as constraints attached to types. Thus principles are stated as conditionals whose antecedent is restricted to be a type (thereby not allowing for the stating of the Subcategorization Principle (cf. below) in its original form). This restriction is not necessarily a disadvantage, since the required subtyping of the antecedent can also be achieved by introducing a type into the hierarchy, the type definition of which reflects this subtyping.

Nevertheless, our implementation provides for an explicit narrowing of the antecedent in the principle itself, since this device also plays an important role in specifying delay conditions, as will be seen below.

The distinctions made above affect only how the applicability of a principle is specified. Another important point relevant for the implementation of these conditional constraints is the form which the constraints expressed by a principle may take. Besides constraints enforcing simple structure sharing there are also complex relational dependencies.

An example of a principle involving relational dependencies (here stated functionally) is the *Subcategorization Principle* (Fig. 1). It states that

> "in a headed phrase (i.e. a phrasal sign whose DTRS value is of sort *head-struc*) the SUBCAT value of the head daughter is the concatenation of the phrases SUBCAT list with the list of the SYNSEM values of the complement daughters."

$$\left[\mathrm{DTRS}\ \textit{head-struc}\right]_{\textit{phrase}} \implies \begin{bmatrix} \mathrm{SYNSEM} & \left[\mathrm{LOC}|\mathrm{CAT}|\mathrm{SUBCAT}\ \boxed{1}\right] \\ \mathrm{DTRS} & \begin{bmatrix} \mathrm{H\text{-}DTR}|\mathrm{SYNSEM}|\mathrm{LOC}|\mathrm{CAT}|\mathrm{SUBCAT}\ \textit{append}(\boxed{1}, \textit{synsem-values}(\boxed{2})) \\ \mathrm{C\text{-}DTRS}\quad \boxed{2} \end{bmatrix} \end{bmatrix}_{\textit{phrase}}$$

Fig. 1. Subcategorization Principle

This principle exhibits complex functional constraints (list-append and a function collecting the SYNSEM values of a list of feature structures) which extend the usual feature structure unification formalisms, but which must also be dealt with when implementing HPSG.

3 A CLP-based Approach to Feature Unification

As has become apparent from the discussion above, the constraints used in HPSG extend the capabilities of the standard feature unification formalisms. In order to integrate such complex constraints in the feature unification framework we interpret unification of typed feature structures under the restrictions of principled constraints as constraint solving in the CLP paradigm ([6]).

In CLP the notion of unification is replaced by the more general notion of constraint solving. Constraint solvers may be embedded into a logic programming language either by writing a meta-interpreter or by making use of a system which allows for the implementation of unification extensions.

DMCAI CLP[2] [4] is a Prolog system, whose unification mechanism is extended in such a way that the user may introduce interpreted terms and specify their meaning with regard to unification through Prolog predicates. The basic mechanism to achieve this behavior is the use of *attributed variables* [5], which can be qualified by arbitrary user-defined attributes. Their behavior wrt. to unification differs from ordinary Prolog variables in two cases: when an attributed variable is unified with a non-variable term or with another attributed variable the unification extensions come into play. For either case a hook-predicate has to specify how the attributes interact and how they have to be interpreted with respect to the semantics of the application domain. Unification succeeds only if this combination—or verification—of the attributes involved is successful:

`combine_attributes(C1,C2)` is called when two attributed variables are unified, computes the resulting attribute according to the underlying theory, and updates the attribute of the unified variable destructively. However, these changes are backtrackable in case of failure later on.

`verify_attribute(C,T)` is called when an attributed variable is to be unified with a non-variable term.

These constraint solving clauses provide a means to integrate constraint solvers into logic programming languages in a declarative and efficient way. Declarativeness is achieved because the equality theory can be stated simply by predicates. Efficiency is preserved because the interaction between syntactic and extended unification is coded in the Prolog kernel, dispensing with the need to have a meta-interpreter managing this interaction.

The representation of typed feature structures in our system makes use of the CLP facilities provided by the enhanced Prolog system described above. A feature structure rooted in `X` is implemented by the attribute `fs(X,Type,Dag)` of the variable `X`, where `Dag` is either

- a list of feature-value pairs, the features being appropriate for `Type` and their values (attributed variables themselves) appropriately type-restricted, or
- an empty list in case `Type` is atomic, or
- the atom `uninstantiated` if the structure rooted in `X` is uninstantiated.

[2] DMCAI CLP, a version of SICStus Prolog providing extensible unification, is available from `ftp.ai.univie.ac.at`. SICStus Prolog 2.1#10 will provide these extensions directly.

Instantiation of feature structures is done via feature logical expressions[3]. As an example, the Prolog goals to instantiate a feature structure representing the constraint induced by the Head Feature Principle of HPSG are given in Fig. 2.

```
X::synsem:loc:cat:head===H,
X::dtrs:head_dtr:synsem:loc:cat:head===H.
```

Fig. 2. Instantiation of Feature Structures

Well-typing is preserved during instantiation, but in a lazy fashion, leaving all nodes uninstantiated that are not on the path denoted by the instantiation expression. This lazy instantiation scheme offers several advantages:

- Recursive types **must** be instantiated lazily to avoid infinite loops. If a lazy instantiation scheme is adopted as the general strategy, such recursive types do not require any special treatment.
- Efficiency gains may result when postponing "default"-initializations of feature structures according to the type scheme. In many cases type information alone is sufficient to eliminate branches in the search space.

Well-typed unification of two feature structures amounts to rewriting the attributes of their root variables and unifying the root-variables. This operation is implemented via the constraint solving clause `combine-attributes`[4] shown in Fig. 3, taking the type hierarchy and feature appropriateness into account (for a more detailed description on how the type hierarchy and feature appropriateness is specified and well-typing is interleaved with instantiation and unification cf. [8]). Recursion is handled automatically by the same constraint solving mechanism when the variables serving as values of a feature are unified.

```
combine_attributes( fs(X1,T1,Dag1), fs(X2,T2,Dag2)) :-
    merge_fs(T1, Dag1, T2, Dag2, T3, Dag3),      % unify well-typed
    detach_attribute(X1),                        % remove attribute
    X1=X2,                                       % unify Prolog vars
    update_attribute(X1, fs(X1, T3, Dag3)).      % rewrite attribute
```

Fig. 3. Feature Constraint Rewriting (preliminary)

The mechanisms presented so far enable the representation of feature structures by Prolog variables carrying them as an attribute. Although these structures are not required to be fully instantiated, they are guaranteed to conform to the well-typing requirements of the grammar. Thus a first step towards complete independence of a particular process model is made. However, the second kind of constraints originating from the grammar, i.e. the principles, have not been incorporated so far. We will turn to that issue below.

[3] The operators `::=`, `::`, `:`, `===` are defined for typing of a node, path restriction, path concatenation and value restriction (type *or* coreference) respectively.

[4] `detach_attribute/1` and `update_attribute/2` are builtin predicates in DMCAI CLP for manipulating attributed variables

4 The Interpretation of Conditional Constraints

As already noted above, our implementation does not rely solely on types to define the applicability of the principles of grammar. Although types play the most important role in stating the conditions under which such a constraint has to be applied, additional devices to narrow these conditions are provided. On the one hand this prevents an overload of the type hierarchy by allowing implicit subtyping. A more important aspect of allowing additional restrictions in the antecedent is their use for specifying delay conditions. Nevertheless, from a formal point of view these extensions can be regarded as syntactic sugar, since only type restrictions (possibly on the type of embedded structures) may be stated in the antecedent. So the formal results (e.g., the Solution Theorem) of [2] and [1] still carry over to our implementation.

Due to the more general format of the constraints in our implementation, type information alone is no longer sufficient to determine which constraints have to be applied. Therefore we adopt a *licensing* view of the principles: Every node of a feature structure has to be licensed by all principles of grammar.
A node is **licensed** by a principle if

- either the feature structure F rooted in that node *satisfies* the applicability conditions and the constraints expressed by the principle are unified into F.
- or the feature structure F rooted in that node is *incompatible* with the applicability conditions of the principle.

The interesting case arises when a feature structure does not satisfy the applicability conditions of the principle but is compatible with them. To preserve monotonicity, the application (or the abandoning) of the constraint has to be delayed until further instantiation or unification steps have restricted the (sub)structure rooted at that node and made a decision possible.

This licensing view also affects the processing strategy of these constraints in a way that squares well with the lazy incremental instantiation of feature structures. Looking up the constraints associated with a node's type any time the type of the node is restricted during unification involves either having to rewrite with all inherited constraints or doing some bookkeeping on which constraints have already been applied. In contrast, in our implementation the set of constraints is applied to a node at the time of its instantiation, i.e. only *once*. Applicable constraints are applied immediately, irrelevant constraints are dropped, and possibly relevant constraints are delayed. Inheritance of constraints along the type hierarchy is achieved automatically by this method.

4.1 Principles as Conditional Constraints

Conditional constraints are written as Prolog clauses where the body of the clause consists of *preconditions* and *constraints*, separated by the operator `===>`. The preconditions may contain only conjunctively connected path expressions; the constraint part may contain arbitrary Prolog code. The procedural interpretation of such conditional constraints reflects their use as rules of inference. The consequent is executed when the antecedent of the conditional is satisfied by the feature structure X. It is dropped if the preconditions are incompatible with X, and delayed otherwise. Failure of a conditional constraint occurs only if feature structure X satisfies the preconditions but the constraint part fails, or (trivially) if X is not a feature structure. Inapplicable or delayed conditional constraints succeed always (at the time they are applied). Of course, delayed constraints may fail later on, if inappropriate instantiations are attempted. As an example, we show the actual code for the Subcategorization Principle (Fig.1) in Fig.4.

```
subcat_principle(X) :-
       X::=phrase, X::dtrs===head_struc
  ===> X::synsem:loc:cat:subcat===Subcat,
       X::dtrs:head_dtr:synsem:loc:cat:Subcat===HeadDtrSubcat,
       X::dtrs:comp_dtrs===CompDtrs,
       synsem_values(CompDtrs,CompDtrsSynsemValues),
       fs_append(HeadDtrSubcat,CompDtrsSynsemValues,Subcat).
```

Fig. 4. Subcategorization Principle

The set of principles that a grammar requires to license each node is defined by the grammar writer via `apply_principles/1` which is called with every node during instantiation.[5] Thus every node of a feature structure may safely be assumed to be appropriately constrained. Not only (incrementally defined) well-typedness of nodes (i.e. of the feature structures rooted therein) is guaranteed automatically, but also their well-formedness wrt. the principles of grammar, disallowing instantiation of substructures not compliant with them.

4.2 The Delay Mechanism for Conditional Constraints

To achieve the delaying behavior of Conditional Constraints as it is described above clauses employing the `===>` operator have to be given a special interpretation at load time (calling the path expressions of the antecedent in the usual way would just instantiate them). This interpretation is achieved via `term_expansion/2` by wrapping the antecedent (in the form of type-restricting path expressions) into a call to `preconditions/2`, which interprets the conditions at run time. The consequent is executed depending on the result of this precondition check. `preconditions/2` interprets path expressions by going down the path in the feature structure and checking the type of the node reached against the type required. The path expression is

satisfied by the feature structure if the path is fully instantiated and the type of the node the path is leading to is at least as specific as the type required;

satisfiable if the path is not fully instantiated but the feature continuing the path is appropriate for the type of the node discontinuing the path, or if the type of the target node is more general than the type required;

incompatible in case of inappropriate features in the path or incompatible type of the target node.

[5] A hierarchical grouping of the principles may be used to reduce the amount of preconditions checking required during node instantiation.

preconditions/2 encodes the result of this subsumption check in its second argument, a list. If the list is empty all preconditions are satisfied. In case of satisfiable conditions, the uninstantiated or too general nodes (i.e. the attributed variables representing them) are collected in this result list.In case of conditions incompatible with the feature structure preconditions/2 fails. Precisely, these conditional clauses are translated as shown in Fig. 5. The actual delaying mech-

```
term_expansion((G :- PreConds ===> Actions), Z) :-
  Z=(G :- ( preconditions(PreConds, L) ->  % check conditions
           ( L == [] -> Actions            % satisfied     -> apply
           ; block_goal(L,G) )             % satisfiable   -> delay
         ; true )).                        % unsatisfiable -> drop
```

Fig. 5. Load time expansion of conditional constraints

anism is implemented via block_goal(VarList,Goal), which stores Goal with each variable in VarList. The idea behind this way of implementing the delay mechanism is to annotate the variables that are "responsible" for the inability to decide on the antecedent of a conditional.[6]

To allow for the annotation of feature structures with delayed goals, the attribute representing them has to be extended to fs(X,Type,Dag,DelayedGoals) and the attribute rewriting clause has to account for thawing of the frozen goals (see Fig. 6). Every time unification affected a node it might be restricted enough to satisfy the applicability conditions of its attached principles, or to contradict them. Therefore all these delayed goals need to be invoked again.

```
combine_attributes( fs(X1,T1,Dag1,P1), fs(X2,T2,Dag2,P2)) :-
    merge_fs(T1, Dag1, T2, Dag2, T3, Dag3),      % unify well-typed
    (X1 == X2 -> true
    ; detach_attribute(X1),                      % remove attribute
      X1=X2,                                     % unify Prolog vars
      update_attribute(X1, fs(X1, T3, Dag3,[])), % rewrite attribute
      start_goals(P1), start_goals(P2)).         % apply constraints
```

Fig. 6. Feature Constraint Rewriting (final version)

The attribute is rewritten with an empty goals list, start_goals/1 successively calls the delayed goals of both input structures. Conditional constraints that are again unable to decide on their applicability, insert themselves at the appropriate place using the mechanism described above for interpreting their preconditions.

[6] This mechanism bears some similarity with the block declaration of SICStus Prolog, which allows to specify delay conditions via argument patterns. But while with block only a distinction between vars and nonvars is possible, our method allows for subsumption checks along arbitrary paths of feature structures.

4.3 Relational Dependencies and Disjunction Elimination

Up to now only constraints involving equality and type restriction have been discussed. But some principles of HPSG involve relational constraints expressible only by recursive definitions. These relational constraints can be integrated straightforwardly into the framework presented here without introducing any further mechanisms. The reasons for the ease of integration are:

- The representation of feature structures by attributed variables makes the interface between feature structures and the programs manipulating them is as simple as possible.
- Only the antecedent of a conditional is syntactically restricted, the constraint itself may be arbitrary Prolog code.
- The delay mechanism is not confined to unary grammar principles. Predicates with more than one argument may also make use of the delay mechanism using the conditional constraint syntax, as long as their arguments (or at least the arguments involved in the preconditions) are feature structures.

A crucial requirement when integrating relational dependencies into a lazy instantiating feature formalism is their ability to cope with insufficiently instantiated arguments. Consider, e.g., the standard definition for `append/3`.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

When called with uninstantiated first and third argument, `append/3` serves as a generator for arbitrary long lists and may trap a program in an infinite loop. Such a behavior is unacceptable in a framework that aims at independence of a particular processing sequence and therefore provides no means to avoid such traps by relying on procedural considerations. As a consequence, the execution of relations has also to be delayed until their arguments are sufficiently instantiated.

The conditionals used for defining the grammar principles provide means to specify the delay conditions also for recursive relations, as in the following example (Fig. 7), defining a version of *append* for feature structure lists[7]. The

```
fs_append(X,Y,Z) :- fs_empty_append(X,Y,Z), fs_nonempty_append(X,Y,Z).
fs_empty_append(X,Y,Z) :- X::=elist  ===> Y = Z.
fs_nonempty_append(X,Y,Z) :-
        X::=nelist
   ===> X::first===F, Z::first===F, X::rest===XRest, Z::rest===ZRest,
        fs_append(XRest,Y,ZRest).
```

Fig. 7. Append for feature structure lists

two specialized clauses of `fs_append/3` are applied in parallel and wait until it can be determined which of the two constraints has to be enforced. Thus the advantages of the conditional definition are twofold. The disjunctive relation *append* can now be written as the conjunction of two specialized cases applying conditionally, and infinite loops due to uninstantiated variables are prevented.

[7] The type *list* subsumes *elist*(the empty list) and *nelist*(the type of nonempty lists, with the appropriate attributes FIRST and REST)

5 Conclusion

The methods for implementing HPSG presented in this paper exploit the benefits of lazy incremental instantiation without having to rely on extra-grammatical control mechanisms to ensure wellformedness of feature structures. Feature structure unification has been implemented as constraint solving under the restriction of well-typedness constraints and the principles of grammar. Conditional constraints have been introduced as the natural device for stating the principles of HPSG, and a delay mechanism has been developed that allows for the application of conditional constraints to insufficiently instantiated structures.

Despite their possibly incomplete instantiation, feature structures may always be regarded as well-formed since their well-typedness and their compliance with the grammar principles is guaranteed in the sense that they are not unifiable with an ill-formed feature structure.

Conditional constraints and the associated delay mechanisms are not confined to unary principles but may be employed by arbitrary relations over feature structures. Thus relational dependencies may be specified independently of the processing model without risking infinite loops. Moreover, disjunctive constraints may be replaced by a conjunction of specialized conditional constraints, waiting until the choice becomes deterministic.

References

1. B. Carpenter. *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science 32. Cambridge University Press, New York, 1992.
2. B. Carpenter, C. Pollard, and A. Franz. The Specification and Implementation of Constraint-Based Unification Grammars. In *Proceedings of the 2nd International Workshop on Parsing Technology*, pp.143–153, Cancun, Mexico, 1991.
3. N. Chomsky. *Lectures on Government and Binding.* Foris, Dordrecht, 1981.
4. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, LNCS 631, pp.260–268. Springer, 1992.
5. S. le Huitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Maluszynski, editors, *Programming Language Implementation and Logic Programming*, pp.136–150. Springer, Heidelberg, 1990.
6. J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proceedings of the 14^{th} ACM POPL Conference*, Munich, 1987.
7. R. Kaplan and J. Bresnan. Lexical-Functional Grammar: A Formal System for Grammatical Representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Mass., 1982.
8. J. Matiasek and W. Heinz. A CLP Based Approach to HPSG. Technical Report TR-93-26, Austrian Research Institute for Artificial Intelligence, 1993.
9. C. Pollard and I. Sag. *Information-Based Syntax and Semantics, Vol. 1: Fundamentals.* CSLI Lecture Notes 13. CSLI, Stanford, CA, 1987.
10. C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago and CSLI Publications, Stanford, 1994.