A CLP Schema to Integrate Specialized Solvers and its Application to Natural Language Processing

Bernhard Pfahringer, Johannes Matiasek

Austrian Research Institute for Artificial Intelligence
Schottengasse 3, A-1010 Vienna, Austria
Tel: +43-1-533 61 12
bernhard@ai.univie.ac.at, john@ai.univie.ac.at

December 28, 1992

Abstract

The problem of combining different constraint solvers has been mentioned among others by [25], [13], [17], [20] without giving satisfactory solutions. We propose a general framework for implementing specialized reasoners/constraint solvers in a logic programming environment using semantic unification. It allows for a modular and declarative definition of the interactions of such reasoners. This is achieved by by using *attributed variables* [15] as a data-structure relating a variable to the *set of all* constraints for this variable. *Conditional rewrite rules* specify simplification and possible interactions of these constraints. A few examples will demonstrate constraints relating to single variables and interactions thereof. We will demonstrate, how this framework leads to a very natural and concise formulation of principles, grammar and lexicon in a HPSG like formalism. Furthermore the necessity of extending the framework to handle constraints relating two or more variables will be discussed.

1 Introduction

In the past few years several mechanisms allowing user-defined extensions to syntactic unification over Herbrand-terms have been proposed and implemented.[1] One attempt to integrate such extensions is socalled *semantic unification*. We will briefly sketch three different methods described in the literature and discuss their drawbacks.

- Kornfeld's E-unification: [16] proposes the following way of extending unification. Any failed unification triggers an attempt to *prove* the equality of the terms using some defined equality theory. But [8] shows that this approach is unsound, incomplete, and unnecessarily inefficient.

- *Metaterms* [13] and *Metastructures* [21] are both aimed at overcoming some of the problems with E-Unification. Since both approaches are very similar, we will restrain our discussion to metaterms. Metaterms are introduced as an additional data-type into the respective Prolog system in addition to logical variables, constants, and terms. Built-in unification now has to treat two additional cases correctly: attempts to unify two metaterms and attempts to unify a metaterm with an ordinary term. This is done by calling two special user-defined predicates, `metametaunify/2` and `metatermunify/2`, which are supposed to check for unifyability and to update the data-structures (metaterms) accordingly. Metaterms come in two flavors, *dereferable* and *non-dereferable*. As a convention, dereferable metaterms must have a variable as their first argument which will be considered the *value* of the metaterm when bound to a non-metaterm. Unification will dereference such chains of metaterms before trying to unify the respective terms. This specification is cognitively too complex and involved when compared to the typical usage of a metaterm: restricting the domain of possible values for a given logical variable.

- Attributed Variables [15] provide a much simpler solution to the problem than metaterms. An implementation in a modified version of SicstusProlog [5] is described in [14]. Instead of metaterms *attributed variables* (abbreviated as avars from now on) are introduced as an additional data-type, which allow logical variables to be directly qualified by arbitrary user-defined attributes. Once again unification has to be extended to defer the two special cases (unify two *avars* and unify an avar with a term) to user-defined predicates. Unifying an ordinary variable and an avar succeeds with the ordinary variable getting bound to the avar. This schema for unification is identical to the one proposed by [17] in their constraint logic programming shell (CLPS). Instead of avars they introduce the similar notion of *solver-vars*.

[1] We assume basic knowledge of constraint logic programming throughout this paper

So for implementing single solvers in Prolog (or at least supplying a convenient interface) plain avars are an adequate language construct. Let us have a look at two very simple solvers handling finite domains [11] and a very simple form of `dif/2` [6] to understand problems of this approach in section 2. Section 3 will introduce our framework to remedy problems. Section 4 will apply this framework to natural language processing and finally section 5 will discuss results and give an outlook.

2 Finite Domains, Dif

Finite domains are finite sets defining the set of legal values for a given variable [11], [12]. Treatment of domain-variables is superficially simplified here, allowing only for attaching domains to variables and for unification of such variables. Using the modified SicstusProlog proposed in [14], this can be specified by the following clauses:

```
domain(X,L) :- attach_attribute(X,dom(X,L)).

combine_attributes(dom(X1,L1),dom(X2,L2)) :-
   detach_attribute(X1),
   X1 = X2,
   intersection(L1,L2,L3),
   L3 \== [],
   ( L3 = [SingleValue] ->
     detach_attribute(X2),
     X2 = SingleValue
   ; update_attribute(X2,dom(X2,L3))
   ).

verify_attribute(dom(X,L),Value) :-
   memberchk(Value,L),
   detach_attribute(X),
   X=Value.
```

`Attach_attribute/2`,`detach_attribute/2`, and `update_attribute/2` are system-supplied predicates. `combine_attributes/2` and `verify_attribute/2` are the names of the predicates the user is supposed to define for handling the above-mentioned special cases of unification. `dom(Var,Values)` is the single attribute used here.

There are a few problems with this schema though. The first observation is that the user predicates have to take care of all low-level updates in data-structures. A more serious flaw can be noticed after introducing the next sim-

plified solver. This solver allows to specify that two given variables should never unify or that a given variable should be distinct from a given constant.

```
dif(X,Y) :-
   attach_attribute(X,dif(X,[Y])),
   attach_attribute(Y,dif(Y,[X])).

combine_attributes(dif(X1,Dif1),dif(X2,Dif2)) :-
   all_different(Dif1,X2),
   all_different(Dif2,X1),
   append(Dif1,Dif2,Dif3),
   detach_attribute(X1),
   X1=X2,
   update_attribute(X2,dif(X2,Dif3)).

verify_attribute(dif(X,Dif),Value) :-
   all_different(Dif,Value),
   detach_attribute(X),
   X=Value.

all_different([],_).
all_different([V|Vs],X) :- X \== V, all_different(Vs,X).
```

This specification allows for constraints like `dif(X,a)` or `dif(X,Y)`, but it is not capable of handling correctly combinations of constraints coming from separate solvers. Goals like the following simple fail:

```
?- dif(X,Y), domain(X,[1,2,3]).

no
```

The reason for this undesirable behavior is the system's inability of unifying the two attributes `dif/2` and `dom/2`. Every variable is restricted to have at most *one* attribute! So, if attributes are simply to be collected or if interactions of attributes have to be taken care of, additional attributes must be invented representing combinations of base attributes and of course additional clauses for `combine_attributes/2` and `verify_attribute/2` must be specified, too, determining behavior of the new attributes. So the above example could introduce an attribute `dom_dif/3`:

```
combine_attributes(dom(X1,L1),dif(X2,Dif2)) :-
   detach_attribute(X1),
   X1 = X2,
   update_attribute(X2,dom_dif(X2,L1,Dif2)).
```

```
combine_attributes(dom(X1,L1),dom_dif(X2,L2,Dif2)) :-
   ....

combine_attributes(dif(X1,L1),dom_dif(X2,L2,Dif2)) :-
   ....

combine_attributes(dom_dif(X1,L1,Dif1),dom_dif(X2,L2,Dif2)) :-
   ....

verify_attribute(dom_dif(X,L,Dif),Value) :-
   all_different(Dif,Value),
   memberchk(Value,L),
   detach_attribute(X),
   X=Value.
```

This solution is not satisfying for a number of problems. First, the number of additional clauses needed is large. In fact, it grows exponentially with the number of different attributes to be combined for a single variable. Secondly, due to the calling conventions of `combine_attributes/2` and `verify_attribute/2`, these predicates cannot be called recursively to achieve composition. Cut-and-paste of appropriate code fragments is the only solution. Specifying possible interactions of attributes, which could presumably cut down the search space early on, is even more complicated, as it requires detailed knowledge of the protocol of the respective base attributes. The work-around proposed in both [13] and [25], namely introducing one monolithic attribute taking care of everything, is not a modular design and it is questionable if a specification of the correct behavior of this single monolithic attribute can be derived automatically from the definitions of the base attributes, at least when using such complex low-level specifications as given above. So, is there a better, more declarative way of specifying attributes and possible interactions?

3 Conditional Rewrite Rules

We propose the following solution. The restriction to exactly one attribute per variable must be lifted. Every variable may be qualified by a *set of attributes*. Consistency checking and simplification (of combinations) of attributes to a kind of normalform must be specified by *conditional rewrite rules*. Single attributes are checked for consistency with rules of the following syntax:

```
Attr => fail :- Guard | Body.
```

The (possibly empty) guard and body can be built of arbitrary goals with the sole restriction that these goals must not directly or indirectly add any attribute

to the variable under consideration. Secondly, rules are needed for detecting the case of an attribute being so constrained that only a single value can fulfil the given constraint:

```
Attr => value(V) :- Guard | Body.
```

If such a case is detected, the variable can be unified immediately to this sole possible value, iff all other attributes of the variable also license this value. The essential point of our proposal is the following. In general rules can reduce pairs of attributes to a single attribute (thus subsuming the above two examples as special cases):

```
Attr1,Attr2 => Attr3 :- Guard | Body.
```

The procedural semantics of this is that two attributes `Attr1` and `Attr2` get rewritten to `Attr3` by means of this specific rule, if proving `Guard` and `Body` succeeds. Success of only the `Guard` and consequent failure of the `Body` indicates an inconsistent set of constraints, thereby failing the original unification attempt. As above, `Guard` and `Body` must not add attributes to the variable under consideration (a restriction that must be partially lifted when dealing with constraints relating two or more variables). Allowing pairs of attributes to be rewritten to a single attribute allows of course for an arbitrary number of attributes to be rewritten to a single attribute and it is this feature that constitutes the power of this schema! This should be compared to languages like ALF [9], where functional logic programs are proved by means of conditional rewrite rules, but which rewrite only single goals, thereby achieving power equivalent to Prolog programs annotated with *wait*-declarations like described in [19]. Additionally we still need a way of specifying compatibility of single attributes with non-attribute values (the `verify_attribute/2` pendant). A special predicate `verify/2`, for which the user must define clauses for each attribute, takes care of this. So the above two simple solvers look like the following in this new framework:

```
dom([]) => fail.
dom([X]) => value(X).
dom(L1),dom(L2) => dom(L3) :- intersection(L1,L2,L3).

dif(X,Difs) => fail :- strict_member(X,Difs) |.
dif(X,Difs1),dif(X,Difs2) => dif(X,Difs3) :-
   append_nodup(Difs1,Difs2,Difs3).

verify( dom(L), Value) :- memberchk(Value,L).
verify( dif(X,Difs), Value) :- check_difs( Difs, Value).
```

Now this is considerably simpler than using plain avars. The system ensures that variables are unified, that their respective sets of constraints are consistent and

that these sets get simplified (achieved by the user-specified rewrite rules) and when unifying vars with values, ensuring that such values are valid with respect to the respective constraints by proving the `verify/2` predicate.

So let us see, what the following query yields, assuming we have defined appropriate interface predicates `domain/2` and `dif/2`:

```
?- domain(X,[1,2,3]), dif(X,3).

attributes(X,[dom([1,2,3]),difs(X,[3])])
```

This is of course a correct solution, but not as far simplified as possible, therefore probably leading to redundant search when encountered during proofs. Fortunately, this can be remedied easily in our framework. We only have to add *one* more rewrite rule:

```
dom(L),difs(_,D) => dom(L1) :- remove_difs(L,D,L1).
```

Now the same query yields:

```
?- domain(X,[1,2,3]), dif(X,3).

attributes(X, [dom([1,2])])
```

The formulation of the above examples in our framework compares very favorably to the one given in [13] with respect to the amount and complexity of user code. Certainly our framework is more adequate from a specificational point of view. But there remains a second question to be answered. Can comparable efficiency be achieved? We have done a proof-of-concept implementation. Avars are used as a low-level device to implement an interpreter handling our generalized rewrite rules. As a testbed for simple domain-variables we chose the KARDIO system [3], where diagnosis considerably benefits from pruning of the search space due to domain variable interactions. First results were not encouraging, runtimes were 2.5 times larger than for the low-level implementation. But using partial evaluation on the interpreter and the rewrite rules, we were able to reduce the overhead to a mere 40 percent. One has to keep in mind that in this example we do not benefit from complex interaction between different types of attributes as there is only one type, namely domains. In the next section we will show a more complex and therefore more interesting application of this schema in the context of Natural Language Processing where benefits of interaction become clear.

4 Application to Natural Language Processing

Recent grammatical formalisms for natural language leave little to do for traditional phrase structure grammar based parsers/generators. Instead they model

language by (possibly typed) feature structures and rely on unification as the operation to combine these feature structures. Phrase structure rules are replaced by the lexicon and by universal and language specific principles [2]. Subcategorization requirements (i.e. which arguments a word may take) are specified in the lexicon. The principles constrain the feature structures themselves, thus restricting the possible combinations and shapes of feature structures to the grammatical ones.

This is especially true for HPSG[23, 24], where a system of typed feature structures is used as a basis for the description of language. A language (more precisely the sign tokens of that language) can be described by the conjunction of the universal and language specific principles conjoined with the disjunction of the lexical signs and grammar rules of that language [23, p. 44].

Although formalisms such as HPSG provide very elegant and descriptive means to describe language, problems may arise, if one wants to implement such a formalism directly. Either some procedural guidance has to be provided to constrain the search space or—what we believe is more adequate, since no extragrammatical devices are needed—some delay mechanisms preventing too early commitment to particular choices or infinite recursion have to be established.

We will show that the CLP-framework described above provides a basis for an efficient, direct implementation of an HPSG grammar.[2]

4.1 Implementing typed feature structures as constraints on variables

First we will describe the kind of feature structures HPSG deals with (following the conventions of [24]). Feature structures are *typed*, i.e. every node is labelled with the type it belongs to. These types form a lattice, part of which is shown below.

```
object ..> sign.
           sign ..> word.
           sign ..> phrase.
                    phrase ..> headed_phrase.
```

A further requirement is that feature structures have to be *well-typed*, that means every (non-atomic) type determines which attibutes are admissible for it, and which types the values of these attributes must belong to, e.g.

```
sign          ==> [phon:   phon,
                   synsem: synsem].
```

[2]The implementation described here is used within the natural language consulting system VIE-$\mathcal{DU}$ (cf. [4]) being developed at the Austrian Research Institute for Artificial Intelligence, for further details of the grammar itself see [10].

```
word          ==> [].
phrase        ==> [dtrs:   const_struc].
headed_phrase ==> [dtrs:   headed_struc].
```

Subtypes inherit the slots of their supertypes. The specifications have to be consistent with the type lattice of course.

Feature structures are implemented by the attribute `fs(Type,Dag)` where `Dag` is either a well-typed list of feature-value pairs, the values being constrained variables, or the atom `uninstantiated`. So instantiation of feature structures can be done in a *lazy* fashion, which can save considerable amounts of space and time in cases where unification fails due to incompatible types.

Unification of two feature structures is performed by the following rewrite rules:

```
fs(T1,uninstantiated),fs(T2,uninstantiated)=>fs(T3,uninstantiated):-
   glb(T1,T2,T3).
fs(T1,uninstantiated),fs(T2,Dag2) => fs(T3,Dag3) :-
   glb(T1,T2,T3),
   instantiate(T1,Dag1),
   unify_dags(Dag1,Dag2,Dag3).
fs(T1,Dag1),fs(T2,Dag2) => fs(T3,Dag3) :-
   glb(T1,T2,T3),
   unify_dags(Dag1,Dag2,Dag3).
```

For `unify_dags/3` there is not very much left to do: only the two input lists have to be merged unifying the values of equal features. Since these values are constrained variables, recursion is handled automagically by the rewrite rules above.

One possibility to get an uninstantiated feature structure instantiated is the unification with an already instantiated feature structure as can be seen above, the other one is to refer to a substructure within an uninstantiated feature structure. For this purpose, i.e. to express path equations, appropriate operators have been defined. For example `X::synsem:loc:cat:head===noun` enforces a subtyping of the syntactic head of `X` to type `noun`, in a similar way structure sharing of substructures can be enforced by using simple Prolog variables as coreference tags in path equations. These syntactic constructs are useful to specify lexical entries and to state the grammatical principles, to which we will turn below.

4.2 Delayed Application of Grammatical Principles

Grammatical principles in HPSG are formulated as conditional feature structures and virtually apply to all nodes of a feature structure. As an example we show how the well-known *Head Feature Principle* of HPSG is represented in our system:

```
head_feature_principle(X) :-
      X::=headed_phrase
  ===>
      X::synsem:loc:cat:head===H,
      X::dtrs:head_dtr:synsem:loc:cat:head===H.
```

The (procedural) interpretation of these principles is, that

- if a node satisfies the antecedent of the conditional then the consequent has to be enforced,
- if a node and the antecedent of the conditional fail to unify, then the principle simply does not apply.

The case not made explicit above is the one which prevents the principles to be integrated straightforwardely, i.e. the case when a node is compatible with the antecedent of a principle but does not satisfy it. In precisely that case principle application has to be delayed.

The idea behind implementing this kind of principle application blocking is to annotate the variables that are "responsible" for the unabiliy to decide on the antecedent with the principle that has to be applied. Therefore we arrive at a representation of feature structures by the attribute `fs(Type,Dag,Goals)` (replacing `fs/2`), where `Goals` is a list of goals which have to be invoked in case either `Type` or `Dag` get restricted due to unification with another feature structure. The rewrite rules above have to be augmented by calls to `start_goals`, which calls all goals in the list, e.g.

```
fs(T1,Dag1,Goals1),fs(T2,Dag2,Goals2) => fs(T3,Dag3,[]) :-
  ...
  start_goals(Goals1), start_goals(Goals2).
```

Note the empty `Goals` in the resulting attribute description. Principles that cannot be applied due to insufficient specification of the resulting feature structure reinsert themselves into that list.

This insertion of delayed goals is triggered by the special treatment of the conditions in the antecedent of the principles, which are restricted to be path equations. If during descending the path of the path equation a substructure is uninstantiated or the type found at the path target is a supertype of the type specified in the condition, the goal is delayed by attaching it to the goals list of that variable.

Implementing application of grammatical principles that way no additional procedural devices have to be introduced to prevent too early commitment to a particular choice or infinite loops due to blind instantiation. The declarative specification of the principles as they are suffices.

Comparing our approach to the implementation of HPSG described in [1], we note the following advantage of our approach: In contrary to their problems of gaining reasonable efficiency our approach benefits from what could be called indexing. Every variable is related to exactly those constraints that are relevant for this variable and to no other constraint whatsoever. So rewriting can be done just at the right point in time, namely when the variable is augmented by an additional constraint or instantiated during unification, and rewriting need only consider the relevant, *small* subset of all constraints. CLG(2) just augments predicates with two arguments for the list of constraints at clause entry and exit, and "applies a rewriting process to the whole list from time to time".

4.3 Feature Structures and Disjunction

Handling of disjunctions appears to be a crucial point in efficiently processing natural language and has therefore attracted a lot of attention in the last few years [26]. Disjunctions result from two different sources, namely the grammar and the lexicon. For disjunctive grammar rules usually enumeration of the different choices via backtracking leads to acceptable runtimes, as the number of such disjunctions is typically rather small. To handle disjunctions coming from the lexicon the same way, usually leads to combinatorial explosion of search space and consequently also of runtime, as serious lexicons tend to contain an abundance of such disjunctions. One can distinguish two types of disjunction, namely *simple* disjunctions of atomic values and *hard* disjunctions of arbitrary feature structures. Simple ones can be interpreted efficiently with the above described `domain/2` attribute. So we can specify, e.g., that the only possible values for case of German proper nouns having no -s suffix are nominative, dative, or accusative:

```
X::synsem:loc:cat:head:case===Case, domain(Case,[nom,dat,acc]).
```

There is just one additional rewrite rule to be specified telling the system to fail in case a domain attribute and a feature structure attribute have to be unified:

```
dom(L),fs(T,D) => fail.
```

This rule is not even necessary, but improves control as it allows for early detection of failure. Alternatively such atomic disjunctions could be expressed by additional types in the hierarchy, but we think that our solution is more natural, keeping the hierarchy free of arbitrary sets of names, viewing it as a terminology for building structures. Unfortunately such simple disjunctions are rather rare. A slightly more general version allowing for arbitrary ground feature structures, which must be free of path equations, can be specified in analogy to the work described in [22] generalizing finite domains to ground n-ary relations. More interesting are of course disjunctions involving arbitrary feature structures including path equations. We are just experimenting with a schema of postponing these as long

as there is some other deterministic computation to be done. This behavior can be achieved by (ab)using the above described blocking mechanism. Additionally we are investigating the merits of eager satisfiability testing on such disjunctions for early detection of non-local inconsistencies, but still without going to full expansion to disjunctive normalforms.

5 Discussion and Further Research

We have outlined a framework for a reasoning architecture integrating specialized solvers in a logic programming environment via rewriting sets of attributes of variables. Conditional rewrite rules allow for a declarative and modular specification of such an integration. Furthermore they open the possibility of using well-known algorithms and results from research in rewrite rule systems such as proving properties like termination [7], completeness, etc. for solvers and combinations thereof. Right now we are investigating more interesting/complex constraint theories exhibiting interactions of two or more variables. These seem to fit into our framework, too, albeit less elegantly. We have already done a generalization of domain variables to ground relational tables capturing relational dependencies of two or more variables. The tricky point is to ensure having updates behave as *atomic actions*. One approach we currently investigate is splitting the body of the rewrite rules into two separate parts, where the latter part is allowed to *add* attributes thereby partially lifting the above made restrictions. Additionally we are working on the compilation and optimization of the rewrite rules. The ultimate goal is to reduce the overhead to less than ten percent when the constraint stores contain just a few attributes (especially for the case of single attributes) and to reasonably bound the search for larger constraint stores using ideas from production systems [18]. Regarding application in natural language processing the most interesting phenomena is of course handling of disjunctions. Once we have completed a reasonably sized lexicon, empirical comparisons will be performed to other approaches like those discussed in [26].

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research. The second author was supported by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*, Grant No. P7986-PHY.

We are indebted to Igor Mozetic for providing the KARDIO model, to Christian Holzbaur for providing the modified SicstusProlog, and especially to Robert Trappl for creating a very special working environment.

References

[1] Balari S., Varile G.B., Damas L., Moreira N.: CLG(n): Constraint Logic Grammars, in Karlgren H.(ed.), *Proceedings of the 13th International Conference on Computational Linguistics*, University of Helsinki, Finland, pp.7-12, 1990.

[2] Berwick R.: Principles of Principle-Based Parsing, in Berwick R., Abney S., Tenny C.: *Principle-Based Parsing*, Kluwer, Dordrecht, 1991.

[3] Bratko I., Mozetic I., Lavrac N.: *Kardio - A Study in Deep and Qualitative Knowledge for Expert Systems*, MIT Press, Cambridge, MA, 1989.

[4] Buchberger E., Garner E., Heinz W., Matiasek J., Pfahringer B.: VIE-DU - Dialogue by Unification, in Kaindl H.(ed.), *7. Österreichische Artificial-Intelligence-Tagung*, Springer, Berlin, pp.42-51, 1991.

[5] Carlsson M., Widen J.: *Sicstus Prolog Users Manual*, Swedish Institute of Computer Science, SICS/R-88/88007C, 1990.

[6] Colmerauer A.: Opening the Prolog III Universe, *BYTE*, August 1987.

[7] Dershowitz N.: Termination, *Proceedings Rewriting Techniques and Applications*, Springer, Heidelberg, 1985

[8] Elcock E.W., Hoddinott O.: Comments on Kornfeld's "Equality for Prolog": E-unification as a Mechanism for Augmenting the Prolog Search Strategy, in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Morgan Kaufmann, Los Altos, CA, 1986.

[9] Hanus M.: Improving Control of Logic Programs by Using Functional Logic Languages, in Bruynooghe M. and Wirsing M.(eds.), *Programming Language Implementation and Logic Programming*, Springer, LNCS 631, 1992.

[10] Heinz, W. and J. Matiasek: Argument Structure and Case Assignment in German, in J. Nerbonne, K. Netter and C. Pollard (eds.), *German Grammar in HPSG*, CSLI Lecture Notes, CSLI, Stanford, to appear.

[11] Hentenryck P.van, Dincbas M.: Domains in Logic Programming, in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Morgan Kaufmann, Los Altos, CA, 1986.

[12] Hentenryck P.van: *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[13] Holzbaur C.: Specification of Constraint Based Inference Mechanisms through Extended Unification, Institut fuer Med.Kybernetik u. AI, Universitaet Wien, Dissertation, 1990.

[14] Holzbaur C.: *A Variant of SicstusProlog featuring Extensible Unification*, Institut fuer Med.Kybernetik u. AI, Universitaet Wien, 1992.

[15] Huitouze S.le: A new data structure for implementing extensions to Prolog, in Deransart P., Maluszunski J.(eds.), *Programming Language Implementation and Logic Programming*, Springer, Heidelberg, 136-150, 1990.

[16] Kornfeld W.A.: Equality for Prolog, in *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1983.

[17] Lim P., Stuckey P.: A Constraint Logic Programming Shell, in Deransart P., Maluszunski J.(eds.), *Programming Language Implementation and Logic Programming*, Springer, Heidelberg, 1990

[18] Miranker D.P., Brant D.A., Lofaso B., Gadbois D.: On the Performance of Lazy Matching in Production Systems, in *Proceedings of the 8th National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, 1990.

[19] Naish L.: *Negation and Control in Prolog*, University of Melbourne, 85/12, 1985.

[20] Nelson G., Oppen D.: Simplification by Cooperating Decision Procedures, *TOPLAS*, 1(2), April 1980.

[21] Neumerkel U.: Extensible Unification by Metastructures, *Proc. META90*, 1990.

[22] Pfahringer B.: *CLP(gRel): Explicit Manipulation of (ground) Relational Dependencies in Logic Programming*, OeFAI Technical Report 92-03, Vienna, 1992

[23] Pollard, C. and I. Sag: *Information-Based Syntax and Semantics, Vol. 1: Fundamentals*, CSLI Lecture Notes 13, CSLI, Stanford, 1987.

[24] Pollard, C., and I. Sag, *Head-Driven Phrase Structure Grammar*, To be published by University of Chicago Press and CSLI Publications, in press.

[25] Schroedl S.: *FIDO: Implementation eines Constraint Logic Programming Systems Finite Domains*, Diploma Thesis, University of Saarbruecken, Germany, 1991

[26] Trost H.(ed.): *Coping with Linguistic Ambiguity in Typed Feature Formalisms*, Proceedings of a workshop held at ECAI'92, 1992.