ARBEITSPAPIERE DES SONDERFORSCHUNGSBEREICHS 340
"Sprachtheoretische Grundlagen der Computerlinguistik"
Bericht Nr. 150, Oktober 2000

SOME ASPECTS OF NATURAL LANGUAGE PROCESSING AND CONSTRAINT PROGRAMMING

FRANK MORAWIETZ

(Editor)

Universität Stuttgart • Universität Tübingen
IBM Beutschland

ISSN 0947-6954/00

Contents

Preface

This collection of articles grew out of a class on *Constraint Programming* taught at the *Seminar für Sprachwissenschaft* of the Universität Tübingen in the Winter 1998/99 by Tom Cornell and Frank Morawietz. The course was based on textbooks from computer science, mainly the book by Frühwirth and Abdennadher (1997), in other parts on the one written by Marriott and Stuckey (1998) and furthermore a large number of individual articles given in the bibliography for this preface.

During the course, it became apparent that despite of the success of systems like Oz (Smolka; 1995), there is less connection than commonly assumed between the constraint-based programming schemes in computer science and constraint-based theories about natural language. This may be due to the fact that computer scientists are interested in solving the combinatorial problem of finding simultaneous variable bindings on tightly constrained domains, whereas linguists are usually interested in the well-formedness of entire structures where variable bindings only serve as the means of identifying particular nodes in that structure.

At the beginning of the course, we covered the basic techniques of constraint satisfaction, constraint propagation, constraint logic programming and concurrent constraint programming both theoretically and practically. In the remainder, we tried to apply the constraint programming techniques to issues in natural language processing (NLP).

It also became apparent that on the practical side Constraint Handling Rules (CHR) are flexible and powerful enough to test and develop prototypes for all sorts of applications fast and conveniently. Out of these attempts to test, to learn and to apply these new techniques grew the following papers which center around the usage of constraint programming and its techniques in NLP. The papers simply appear in alphabetical order, no significance is attached to this.

I wish to thank Tom Cornell, without whom the course wouldn't have happened.

Tübingen, October 16, 2000
Frank Morawietz

BIBLIOGRAPHY

Abdennadher, S., Fruehwirth, T. and Meuss, H. (1996). On confluence of constraint handling rules, *Lecture Notes in Computer Science* **1118**: 1–15.

Aït-Kaci, H., Podelski, A. and Smolka, G. (1992). A feature-based constraint system for logic programming with entailment, *Research Report RR-92-17*, Deutsches Forschungszentrum für Künstliche Intelligenz.

Andersen, H. R. (1996). An introduction to binary decision diagrams, Course Notes, Technical University Denmark.

Baader, F. and Schulz, K. U. (1995). On the combination of symbolic constraints, solution domains, and constraint solvers, *Proceedings of the International Conference on Constraint Programming, CP-95*, Vol. 976 of *Lecture Notes in Computer Science*, Springer–Verlag.

Blache, P. (1992). Using active constraints to parse GPSGs, *Proceedings of COLING-92*, pp. 81–86.

Bryant, R. (1986). Graph-based algorithms for boolean function manipulation, *IEEE Transactions on Computers* **8**(C-35): 677–691.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary decision diagrams, *Technical Report CS-92-160*, Carnegie Mellon University, School of Computer Science.

Bryant, R. E. (1995). Binary decision diagrams and beyond: Enabling technologies for formal verification, *International Conference on Computer Aided Design*, IEEE Computer Society Press, Los Alamitos, Ca., USA, pp. 236–245.

Carlson, B. (1991). An approximation theory for constraint logic programs, Licentiate Thesis. Dep. of Computer Science, Uppsala University.

Carpenter, B. (1992). *The Logic of Typed Feature Structures: With Applications to Unification Grammars, Logic Programs and Constraint Resolution*, Cambridge University Press, Cambridge.

Comon, H. (1994). Completion of rewrite systems with membership constraints, *Journal of Symbolic Computation*.

Cornell, T. L. (1994). On determining the consistency of partial descriptions of trees, *32nd Annual Meeting of the ACL: Proceedings of the Conference*, Association for Computational Linguistics, Las Cruces, New Mexico, pp. 163–170.

Cornell, T. L. (1996). On determining the consistency of partial descriptions of trees. Longer version of Cornell (1994). Adds two appendices.

Dechter and van Beek (1997). Local and global relational consistency, *Theoretical Computer Science, all*, Vol. 173.

Dörre, J. and Dorna, M. (1993). CUF - a formalism for linguistic knowledge representation, *in* J. Dörre (ed.), *Computational aspects of constraint based* *linguistic descriptions I*, DYANA-2 Deliverable R1.2.A, Universität Stuttgart, pp. 1–22.

Fitting, M. (1990). *First-Order Logic and Automated Theorem Proving*, Texts and Monographs in Computer Science, Springer-Verlag, New York.

Frühwirt, T. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1–3): 95–138. Special Issue on Constraint Logic Programming.

Frühwirth, T. (1995). Constraint Handling Rules, *in* A. Podelski (ed.), *Constraint Programming: Basics and Trends*, LNCS 910, Springer. (Châtillon-sur-Seine Spring School, France, May 1994).

Frühwirth, T. and Abdennadher, S. (1997). *Constraint-Programmierung*, Springer-Lehrbuch, Springer.

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E. and Wallace, M. (1993). Constraint logic programming: An informal introduction, *Technical Report ECRC-93-05*, ECRC. Appears in *Logic Programming Summer School*, Zürich, Springer Lecture Notes.

Gilleron, R., Tison, S. and Tommasi, M. (1996). Set constraints and automata.

Höhfeld, M. and Smolka, G. (1988). Definite relations over constraint languages, *LILOG Report 53*, IBM Deutschland, Stuttgart, Germany.

Holzbaur, C. and Früwirth, T. (1998a). Compiling constraint handling rules, *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, The Netherlands.

Holzbaur, C. and Früwirth, T. (1998b). Join evaluation schemata for constraint handling rules, *Workshop logische Programmierung (WLP 98)*, Technical University of Vienna.

Jaffar, J. and Lassez, J.-L. (1986). Constraint logic programming, *Technical report*, IBM-Thomas J. Watson Research Center, Yorktown Heights, N.Y. 10598.

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey, *J. of Logic Programming* **19/20**: 503–581.

Jeavons, P., Cohen, D. and Cooper, M. (1998). Constraints, consistency and closure, *Artificial Intelligence* **101**(1-2): 251–265.

Johnson, M. (1995). Constraint-based natural language parsing, ESSLLI '95, Barcelona, Course notes.

Kowalski, R. A. (1979). Algorithm= logic + control, *Comm. of the ACM* **22**: 424–431.

Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey, *A.I. Magazine* **13**(1): 32–44.

Ladkin, P. B. and Maddux, R. D. (1994). On binary constraint problems, *Journal of the ACM* **41**(3): 435–469.

Le Provost, T. and Wallace, M. (1993). Generalized constraint propagation over the clp scheme, *Journal of Logic Programming* **16**: 319–359.

Lloyd, J. (1984). *Foundations of Logic Programming*, Springer Verlag.

Mackworth, A. (1977). Consistency in networks of constraints, *Artificial Intelligence*.

Mackworth, A. K. (1992). Constraint satisfaction, *in* S. C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, Vol. 1, J. Wiley & Sons, New York, pp. 285–293.

Mackworth, A. K. and Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problem, *Artificial Intelligence* **25**(1): 65–74.

Mackworth, A. K. and Freuder, E. C. (1992). The complexity of constraint satisfaction revisited, *Technical Report TR-92-26*, Department of Computer Science, University of British Columbia. Tue, 22 Jul 1997 22:21:32 GMT.

Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints*, MIT Press.

Maxwell, J. T. and Kaplan, R. M. (1989). An overview of disjunctive constraint satisfaction, *Proceedings of International Workshop on Parsing Technologies*, pp. 18–27.

Maxwell, J. T. and Kaplan, R. M. (1991). The interface between phrasal and functional constraints, *in* M. Rosner, C. J. Rupp and R. Johnson (eds), *Contraint Propogation, Linguistic Description, and Computation*, Istituto Dalle Molle IDSIA, Lugano.

Palmgren, E. (1994). *Denotational Semantics of Constraint Logic Programming – A Nonstandard Approach*, NATO ASI Series F, Springer.

Saraswat, V. A. (1993). *Concurrent Constraint Programming*, MIT Press, Cambridge, Massachusetts.

Saraswat, V. and van Hentenryck, P. (eds) (1995). *Principles and Practice of Constraint Programming.*, MIT Press, Cambridge, MA.

Shieber, S. M. (1992). *Constraint–Based Grammar Formalisms*, MIT Press.

Smolka, G. (1993). Residuation and guarded rules for constraint logic programming, *in* F. Benhamou and A. Colmerauer (eds), *Constraint Logic Programming: Selected Research*, MIT Press, pp. 405–419.

Smolka, G. (1995). The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today*, Vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 324–343.

van Hentenryck, P. (1989). *Constraint Staisfaction in Logic Programming*, MIT Press.

van Hentenryck, P. (1990). Constraint logic programming: From theory to applications, *in* M. Debray, Saumya; Hermenegildo (ed.), *Proceedings of the 1990*

North American Conference on Logic Programming, MIT Press, Austin, TX, pp. 831–831.

van Hentenryck, P. (1991). Constraint logic programming, *Knowledge Engineering Review* **6**(3): 151–194. (Also available as Brown University Technical Report CS-91-05).

van Hentenryck, P. (1993). Constraint programming languages, *Proceedings of the 5th International Conference on Tools with Artificial Intelligence*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 10–13.

van Hentenryck, P. and Dincbas, M. (1987). Forward checking in logic programming, *in* J.-L. Lassez (ed.), *Proceedings of the Fourth International Conference on Logic Programming (ICLP '87)*, MIT Press, Melbourne, Australia, pp. 229–256.

van Hentenryck, P., Simonis, H. and Dincbas, M. (1991). Constraint satifaction using constraint logic programming, *Technical Report CS-91-62*, Department of Computer Science, Brown University. Sun, 13 Jul 1997 18:30:16 GMT.

Vempaty, N. R. (1992). Solving constraint satisfaction problems using finite state automata, *in* W. Swartout (ed.), *Proceedings of the 10th National Conference on Artificial Intelligence*, MIT Press, San Jose, CA, pp. 453–458.

Wetzel, G., Kowalski, R. and Toni, F. (1995). A theorem-proving approach to CLP, *11th Workshop on Logic Programming*.

1

A Non-Generative Constraint-Based Formalism

PHILIPPE BLACHE AND FRANK MORAWIETZ

LPL-CNRS, Université de Provence
29 Avenue Robert Schuman
13621 Aix-en-Provence, France

SFS, Universität Tübingen
Wilhelmstr. 113
72074 Tübingen, Germany

pb@lpl.univ-aix.fr
frank@sfs.nphil.uni-tuebingen.de

ABSTRACT. Most of modern linguistic theories rely on the notion of constraints. However, only few applications try to implement the parsing process directly with constraint programming: in most cases, constraints are interpreted in a passive sense, the parsing process itself not being different than the one of classical derivational approaches. We show in this paper that the problem comes from the representation of linguistic information and the generative interpretation of the relation between grammar and language. We propose a new approach, called *Property Grammar* in which all syntactic information is represented with constraints, the parsing process directly being constraint resolution. Our ideas are implemented using Constraint Handling Rules (Frühwirth; 1998).

1. INTRODUCTION

Modern linguistic theories often make use of the notion of *constraint*: most constraint-based theories (cf. Shieber (1992), Borsley (1996)) as well as other paradigms such as representational versions of the principle and parameters approach (including optimality theory, Archangeli and Langendoen (1997)) or dependency grammars (constraints dependency grammars, Maruyama (1990)) follow this tendency. A constraint approach offers several advantages, in particular a fine-grained representation of information, a clear distinction between linguistic objects and their properties, and a better declarativity.

In computational linguistics, the idea that constraints have a direct counterpart in the implementation is implicit. Unfortunately, this is only partly true and constraints only have local scope during the processing. In spite of several works trying to use constraint programming (see for example Günthner (1988), Maruyama (1990) or more recently Blache and Paquelin (1996), Götz and Meurers (1997), Duchier and Thater (1999)), the parsing process cannot be interpreted as actual constraint resolution. This doesn't follow the original idea (see in particular Sag and Wasow (1999)) which proposes the direct use of the linguistic information as a set of constraints over the categories. We show in this paper that this problem has two main origins: the representation of the linguistic information and the influence of the generative conception of the relation between grammars and their generated languages on the linguistic analysis.

We propose a new approach called *Property Grammar* representing all the linguistic information by means of simple constraints. Since these constraints constitute an actual constraint system (in the same sense as a set of equations forms a system), it is then possible to consider parsing as a pure constraint program.

2. Some Desiderata for Constraint-Based Approaches

Constraints in linguistic theories generally play only a local role[1]: they are used to control the instantiation of linguistic objects (the categories) and represent relations between these objects (or their components). Then, a clear distinction between the objects and their properties becomes possible, which favors declarativity. For example, in a theory like GPSG, one can find constraints at different levels of the description: linear precedence, feature cooccurrence restriction or universal instantiation principles. But not all the information is represented with constraints and during a parse, one has to select a local tree first and then verify that this tree satisfies the different constraints. This is typically a passive use of constraints.

The problem is different in HPSG in which most of the information is actually represented by means of constraints (cf. Sag and Wasow (1999) and especially for an implementation Blache and Paquelin (1996) or Meurers and Minnen (1998)). However, the hierarchical information remains predominant in the sense that verification of constraints can only be applied to an implicit local tree. The verification of the principles for example requires the construction of a hierarchical structure containing a mother and its daughters. This doesn't mean

[1]Constraints in this case are not active all along the parsing process, but only for specific structures built during the parse. Such constraints are local because their scope is reduced according to the knowledge of the constrained structure.

that such a structure has to be fully specified. Some delaying mechanisms can obviously be applied and active constraints can be used locally. But in this case too, we can globally characterize the use of constraints for the parsing process as passive and parsing doesn't amount to pure constraint resolution.

One of the reasons for these problems is the generative interpretation of the relation between grammars and languages. In this case, the notion of derivation is central and parsing an input consists in finding a derivation generating it. Such a conception is revealed in the approaches cited above with the role played by the local trees (which usually amounts to one derivation step). Parsing is then conceived as a derivation mechanism controlled by constraints (often reduced to unification). This certainly stems from the fact that licensing of constraints is implemented as improved versions of the generate-and-test paradigm to ensure some sort of efficiency (and sometimes even termination). So far, we do not know of an implementation which directly uses only constraints to drive the parsing (or the generation) process.

An optimal use of constraints, both from the linguistic and the computational point of view, should meet some requirements. We present in this section some basic desiderata for a genuine constraint-based approach.

The first important property concerns knowledge representation: all linguistic information has to be represented by means of constraints which form an actual system. This property has an important consequence on the declarativity of the approach. Indeed, it allows a clear distinction between the representation of information (in particular the form of the objects) and the mechanisms for calculating with them.

The second important point is the fact that, insofar as the set of constraints constitutes a system, all the constraints are at the same level. This means that the order of verification of the constraints is not relevant. Obviously, some heuristics can rely on a hierarchization of the constraints or more precisely on the control of their relaxation. But this doesn't alter the fact that the system of constraints as to be taken as a whole. Our approach is thus totally different from the Optimality Theory (see Archangeli and Langendoen (1997)) in which constraint ranking contains implicitly linguistic information.

Encapsulation is another important property for constraint-based theories: a constraint must represent homogeneous information. We can then distinguish different types of constraints, each one representing a specific part of linguistic information. Such a requirement avoids implicit information and each type of constraint becomes linguistically motivated.

The last requirement concerns the notion of grammaticality. One interesting property of constraint solving is the possibility to build an approximate solution (when no exact solution can be found) which is nothing but the state of the constraint system after verifying its satisfiability for a given input. As for parsing, finding an exact solution consists in associating a syntactic structure with a given input, e.g. finding a derivation from a distinguished symbol to this input, answering the question of grammaticality: an input is grammatical if a solution can be found. However, the relevant question concerning parsing of real natural language should not be grammaticality, but rather the robustness of providing information about the input. Any kind of input has to be treated. Therefore, we propose to replace the notion of grammaticality with that of *characterization* which is much more general: a characterization is the state of the constraint system for a given input.

3. Property Grammar

Property Grammar (cf. Blache (1999), Bès and Blache (1999)) constitutes an approach implementing these requirements: declarativity, encapsulation, satisfiability. We present in this section their main characteristics. As specified above, all the linguistic information is represented here by means of *properties*.

3.1. The Basic Properties. We use for the representation of syntactic information 7 properties[2] defined as follows :

- **Constituency** (*const*): Set of categories constituting a phrase.
- **Obligation** (*oblig*): Set of compulsory, unique categories (heads).
- **Unicity** (*uniq*): Set of categories which cannot be repeated in a phrase.
- **Requirement** ($\Rightarrow$): Cooccurrence between sets of categories.
- **Exclusion** ($\not\Leftrightarrow$): Restriction of cooccurrence between sets of categories.
- **Linearity** ($\prec$): Linear precedence constraints.
- **Dependency** ($\leadsto$): Dependency relations between categories.

Let us take some examples (all our examples are from French):

- *Constituency: const(NP) =* $\{$*Det, N, AP, PP, Sup, Pro*$\}$
  All these constituents can appear within a sequence characterizing a *NP*.
- *Obligation*: *oblig(NP) =*$\{$*N, AP, Pro*$\}$
  In a NP, these categories can be the head.
- *Unicity*: *unic(NP) =* $\{$*Det, N, AP, PP, Sup, Pro*$\}$
  Set of unique categories in a *NP*.
- *Linearity*: $Det \prec AP$ $\qquad AP \prec N$

[2]This list of properties can be extended in order to integrate other levels of linguistic analysis such as prosody or semantics.

- *Requirement*:
  - $\{le\ \hat{e}tre_{[N,P]}\} \overset{VP}{\Rightarrow} ClR_{[N,P]}$ (Je me le suis dit/*I told that to myself*)
    If the categories *le* (clitic) and a finite verb *être* (with agreement features *N* and *P*) cooccur (characterizing a *VP*), then a reflexive clitic (which agrees with the verb) also has to be present.
- *Exclusion*:
  - $ClR \overset{VP}{\not\Leftrightarrow} lui$ (*Je me lui dis / **I told him myself*)
    In a *VP*, a reflexive clitic cannot cooccur with the clitic *lui*.
- *Dependency*: $Det \rightsquigarrow N$ $\quad$ $AP \rightsquigarrow N$

The first important remark is that characterizations are associated with sets of categories. This is interesting in the sense that a characterization can be given independently from the characterization of the non-lexical constituents belonging to this set. The second important point concerns the fact that properties can be expressed over sets of categories, allowing then to represent contextual information. It is the case in particular for the *requirement* and *exclusion* properties, see the indices.

4. Parsing as Constraint Resolution

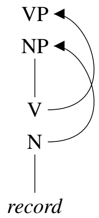
Insofar as syntactic information forms a constraint system, parsing can then be implemented using constraint programming techniques such as constraint resolution. This approach has several advantages, in particular flexibility and reusability. For example, one can choose to verify only a subset of the constraint system. It is also possible to satisfy constraints with different granularity levels (e.g. a shallow description of the sentence level together with a fine-grained definition of the *NP*). The possibility of integrating several constraints coming from different levels of the linguistic analysis also constitutes an interesting property of using such an approach. But the most important advantage of constraint resolution lies in the fact that it allows a direct interpretation of the notion of characterization.

We present in this section the general architecture of the parsing process. We first describe the elementary trees constituting the basic information and then the constraint resolution process itself.

4.1. Elementary trees. Interpreting the parsing process as checking constraint satisfaction can offer a new point of view. Indeed, constraints are applied directly over the objects (the categories) rather than over structures (a local tree). It is thus possible to apply the constraints to the set of all the possible categories that can be associated with the input.

Insofar as constraints are applied over sets of categories (whatever their level, lexical or not), we can associate with each word its possible categories and their possible projections.

Such structures can be represented as an elementary tree as described in (1).

(1) 

This example presents the elementary tree of the word *"record"* which can be a noun or a verb. These categories can respectively belong to the *constituent* property of the *NP* and *VP*. The elementary tree is then composed of three nodes: the lexical item, its categorizations and their projections. It is possible to associate such a tree with each lexical item, without any distinction between major and minor categories (i.e. our notion of projection is reduced to the membership of a category to a *const* property).

4.2. Satisfying Constraints to Build Characterizations. The parsing process takes as input the set of elementary trees that can be associated with the sentence and builds all the characterizations of this input. A characterization is the state of the constraint system for a particular set of categories, in other words the set of satisfied properties and that of unsatisfied ones.

A *characterization* is then defined over a set of categories by the set P^+ (satisfied properties) and P^- (unsatisfied properties).

Let us take an example. Fig. 1 presents a grammar of the noun phrase in French. It describes in particular the properties of the noun phrase, the adjective phrase and the superlative. In (2), the elementary trees associated with the words of the noun phrase *le livre le plus rare (the rarest book)* are given.

(2)

NP	.	NP	AP	AP
Sup	NP	Sup	Sup	Sup
Det	N	Det	Adv	Adj
le	*livre*	*le*	*plus*	*rare*
(1)	(2)	(3)	(4)	(5)

The first stage consists in determining the set of categories to be verified. The most general approach consists in characterizing all subsets of categories

Properties of the *NP*

(1)	*Const* = {*Det, N, AP, Sup*}
(2)	*Oblig* = {*N, AP*}
(3)	$N[com] \Rightarrow Det$
(4)	$Det \prec N$
(5)	$Det \prec AP$
(6)	$Det \prec Sup$
(8)	$N \prec Sup$
(9)	$AP \not\Leftrightarrow Sup$
(10)	$Det \leadsto N$
(11)	$AP \leadsto N$
(12)	$Sup \leadsto N$

Properties of the *AP*

(1)	*Const* = {*Adj, Adv*}
(2)	*Oblig* = {*Adj*}
(3)	$Adv \prec Adj$
(4)	$Adv \leadsto Adj$

Properties of the *Sup*

(1)	*Const* = {*Det, Adv, Adj*}
(2)	*Oblig* = {*Adj*}
(3)	$Adj \Rightarrow Det$
(4)	$Adj \Rightarrow Adv$
(5)	$Det \prec Adv$
(6)	$Det \prec Adv$
(7)	$Adv \prec Adj$
(8)	$Det \leadsto Adj$
(9)	$Adv \leadsto Adj$

FIGURE 1. A property grammar for the *NP*

belonging to the set of elementary trees. Considering all the possible subsets of categories has an exponential cost but some simple control mechanisms can be applied. In the following, we will only take into account the sets of juxtaposed categories (implementing then the projectivity constraint); in this case, sets of constraints are in fact lists.

Building the characterizations consists for each subset of categories of verifying the satisfiability of the properties. The properties that cannot be evaluated remain untouched. This means that a set of properties belonging to a characterization always specifies a phrase-level category. More precisely, it is possible to associate with a set of categories an upper-level category C (or several categories in case of ambiguity). We also say in this case that C is *characterized* by its P^+ and P^- sets.

The following example illustrates the construction of such characterizations for (2) according to the grammar presented in Fig. 1. Each category is associated with its position in the input. The characterization sets follow the sequence of categories with their properties being represented by their indices. For example, the characterization of the set $\{Det_1, N_2\}$ indicates that all the properties specified for these categories (i.e. constituency, obligation, requirement, precedence and dependency) are satisfied. The set $\{Adv_4, AP_5\}$ presents two characterizations: this sequence can specify an *AP* (all the properties of the grammar are verified) or a *Sup* in which a requirement property is not satisfied.

- Det_1/N_2 :	$P^+(NP) = \{1, 2, 3, 4, 10\}$
	$P^-(NP) = \emptyset$
- Sup_1/N_2 :	$P^+(NP) = \{1, 2, 12\}$
	$P^-(NP) = \{6,3\}$
- Det_3/Sup_4 :	$P^+(NP) = \{1, 6\}$
	$P^-(NP) = \{2\}$
- Det_3/AP_4 :	$P^+(NP) = \{1, 5\}$
	$P^-(NP) = \{2\}$
- Sup_3/AP_4 :	$P^+(NP) = \{1\}$
	$P^-(NP) = \{2, 9\}$
- Adv_4/AP_5 :	$P^+(AP) = \{1, 2, 3, 4\}$
	$P^-(AP) = \emptyset$
	$P^+(Sup) = \{1, 2, 4, 9\}$
	$P^-(Sup) = \{3\}$
- $Det_1/N_2/Sup_3$:	$P^+(NP) = \{1\text{-}4, 6, 8, 9, 10, 12\}$
	$P^-(NP) = \emptyset$
- $Sup_1/N_2/Det_3$:	$P^+(NP) = \{1, 2, 3, 9, 10, 12\}$
	$P^-(NP) = \{4, 6, 8\}$
- $Det_3/Adv_4/Adj_5$:	$P^+(Sup) = \{1, 2, 3, 4, 5, 6, 8, 9\}$
	$P^-(NP) = \emptyset$
- $Det_3/AP_4/Sup_5$:	$P^+(Sup) = \{1, 4, 5, 6\}$
	$P^-(NP) = \{2, 9\}$
- $Det_1/N_2/Sup_3/AP_4$:	$P^+(NP) = \{1\text{-}8, 10, 11, 12\}$
	$P^-(NP) = \{9\}$

FIGURE 2. Some characterizations of (2)

Note that some characterizations have an empty set of unsatisfied properties (let us call them *positive* characterizations). But no particular status is given to them. This allows the robust treatment of any kind of input, i.e., we do not restrict the analysis to the set of "grammatical" structures.

4.3. Determining Coverage. The core of the process is the construction of the characterization sets. A general mechanism makes use of these sets in order to find a characterization covering the entire input.

A characterization made up of lexical categories (cf. the superlative's characterization in the previous example), constitutes a covering of the segment of the input corresponding to the positions of the constituents. The question is different for embedded constituents. In this case, the set of categories covers the corresponding part of the input if there exists a characterization for all the embedded constituents describing non-lexical categories. In the previous example, the characterization $\{Det_1, N_2, Sup_3\}$ of the *NP* covers the input because there exists a characterization $\{Det_3, Adv_4, Adj_5\}$ for its embedded constituent, the *Sup*.

Let us be more precise on the evaluation of covering categorizations. Characterizations don't take into account the status of their embedded constituents. In other words, lexical and non-lexical categories play the same role. This means that each position corresponds to a category. In the case of embedded constituents, we then have several successive categories with the same type as described in (3).

(3)

NP	AP	AP	
Sup	Sup	Sup	NP
|	|	|	|
Det	Adv	Adj	N
|	|	|	|
un	*très*	*vieux*	*livre*
(1)	(2)	(3)	(4)

In this example, one covering of the input will be the set of categories: $\{Det_1, AP_2, AP_3, N_4\}$ which characterizes a noun phrase. The condition for this set to be an actual covering is the presence of a characterization of the *AP* (which will be in this case the set of categories $\{Adv_2, Adj_3\}$).

4.4. Controls and Heuristics. A naive implementation of these mechanisms is obviously not efficient. Indeed, let m be the number of words of the input, c the maximum number of categorizations for each word, then the number n of categories to analyze is bounded by $n = 2mc$. Then, the number of sets of categories to analyze is bounded by 2^n.

Several mechanisms can be added in order to reduce the number of sets to analyze and control the resolution process. In the first case, we can choose to build only sets of juxtaposed categories. At a higher level, one can consider only sets corresponding to subsets of properties (i.e. filtering the sets using the *const* property for example). The satisfaction process itself can be controlled using several heuristics. In particular, it is possible to filter the satisfiability according to a threshold given by the cardinality of the set of unsatisfied properties P^-: it is possible to build only characterizations with less than a certain amount of unsatisfied constraints. At the extreme, we can reduce the satisfiability to positive characterizations.

Another kind of heuristic consists in defining a hierarchization within the set of properties. In this case, the technique of the threshold mentioned above can be modulated with such a hierarchization, some constraints playing a more important role than others.

5. Implementing Property Grammars

Constraint programming is an elegant, declarative way of solving problems with higher-level programming languages. It represents a generalization of logic programming in the sense that it allows the (logical) specification of a problem on an arbitrary domain and tries to solve it with a combination of inference and search. A complete introduction to constraint programming can be found for example in Marriott and Stuckey (1998).

We describe in this section an implementation of Property Grammars in a particular constraint programming paradigm. We present this paradigm, Constraint Handling Rules (CHR), in a first step and then explain how to take advantage of it for implementing the resolution process. We want to show with such an implementation that property grammar can be implemented directly using only constraint resolution, which is not actually possible with classical generative approaches.

5.1. Constraint Handling Rules. There are several constraint programming environments available. The most recent and maybe the most flexible is the Constraint Handling Rules package included in SICStus Prolog (Intelligent Systems Laboratory; 1995; Frühwirth; 1998). All of these systems provide mechanisms for solving constraints. They maintain a constraint base or store which is continually monitored for possible rule applications, i.e., whether there is enough information present to successfully use a rule to simplify constraints or to derive new constraints. Whereas usually one deals with a fixed constraint domain and a specialized solver, CHR is an extension of the Prolog language which allows for the specification of user-defined constraints and arbitrary solvers. The strength of the CHR approach lies in the fact that it allows for multiple (conjunctively interpreted) heads in rules, that it is flexible and that it is tightly and transparently integrated into the Prolog engine.

In CHR constraints are just distinguished sets of (atomic) formulas. CHR allow the definition of rule sets for constraint solving with three types of rules: Firstly simplification rules (<=>) which replace a number of constraints in the store with new constraints; secondly propagation rules (==>) which add new constraints to the store in case a number of constraints is already present; and thirdly "simpagation" rules (<=> in combination with a \ in the head of the rule) which replace only those constraints with new ones which are to the right of the backslash.

To improve efficiency, rules can have guards and pragma declarations. A guard (separated from the rest of the body by a |) is a condition which has to be met before the rule can be applied. A pragma declaration affects the way the rule is compiled. Our approach will only introduce the *passive* declaration, so we ignore the other possibilities. Declaring a constraint in the head passive simply means that this constraint will not trigger the rule on its appearance in the constraint store.

We cannot go into the details of the formal semantics of CHR here. The interested reader is referred to Frühwirth (1998). Let us just note that logically, simplification rules are equivalences and propagation rules are implications if their guard is satisfied. Simpagation rules are special cases of simplification rules. Soundness and completeness results for CHR are available (Abdennadher et al.; 1996; Abdennadher; 1998).

5.2. The Implementation in CHR. For the implementation in CHR, we simply have to declare the basic properties as constraints, see Sec. 3. Additionally, we use constraints for the sets of satisfied/nonsatisfied constraints, i.e., we have constraints `pplus/4` and `pminus/4` which encode in their first argument the set of constraints we are considering and an identifier, in their second argument the type of constraint and in the last two the categories involved.

The program takes the input string, a Prolog list of words, and traverses it.[3] For each lexical entry, it posts the corresponding category constraint, i.e., `cat(X,Y)` where `X` is a category and `Y` its position in the numeration.

The CHR implementation immediately starts out by building the elementary trees of the initial categories from the posted `cat/2` constraints via constraint propagation. In general, the CHR engine, i.e., all constraint resolution steps, is triggered as soon as constraints matching the heads are posted.

Basically, the CHR program looks at two category constraints and – if they are in the set of constraints we are considering – uses them together with a constraint from the grammar, as the head of constraint propagation rules. The constraints propagated contain the information whether the pair of categories satisfy the constraint from the grammar or not. In this way, we build a complete table of results on all pairs of categories.[4]

So, we build a complete table of `pplus/4, pminus/4` constraints for all pairs of categories matching the heads of our propagation rules. These constraints store the information about the interaction of any two categories with the resulting rule and set of constraints which licensed them. Nothing more has to be implemented to achieve this behaviour since it is built into the CHR engine. All we have to do is specify the right propagation rules and to post the

[3]Currently from left-to-right although this is arbitrary and can be varied.

[4]Since in the end we are interested in bigger chunks of lexical items then just two, we have to do some post-processing to interpret the entries in the table in the right way. For example, if a precedence constraint succeeds on two pairs of categories, this does not imply that it also succeeds for the entire triple. So, one violation among all the pairs is sufficient to cause failure of that constraint.

constraints from the grammar and input which makes for a very simple and easy to understand program.

Generally, in all the rules dealing with the constraints from the grammar, we apply the rule if we can find two categories within the constraint store which are in the same constituent and a constraint which checks some configuration on them. Only this last constraint triggers the rule since we declared all other constraints to be passive. Note that this ensures that we did all possible projections in each constituent before we post other constraints.

Since, if two category constraints match the head of a rule, they do so in either order, we use the guard to force the linear order as it appears in the input string.

```
prec @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
                                   prec(Z,Cat1,Cat2,Id) ==>
       N < M |
       ( ( X == Cat2, Y == Cat1 ) ->
             pminus(Z-Id,prec,X-N,Y-M)
       ;
             pplus(Z-Id,prec,X-N,Y-M) )
       pragma passive(I1), passive(I2),
              passive(I3), passive(I4).
```

FIGURE 3. An example CHR rule

An example of such a propagation rule is given in Fig. 3. We are considering a precedence constraint `Id` on the categories `X` and `Y` within the constituent `Z` with indices `N` and `M`. We check satisfiability of the constraint in the head of the implication and then add the corresponding `pplus` or `pminus` constraint to the store.[5] Note that all the constraints actually appearing in the constraint store are ground such that we can test for literal identity.

The post-processing then just collects all the sets, gets all pairs of appearing categories if they are within the same range and collects all positive and negative constraints attached to them. While doing so, we interpret its members in the following way:

- if we have a successful obligatory or implicational constraint on any two categories within a constituent, we can ignore all the failed ones;

[5]We need four `cat` constraints to ensure that both categories belong to the same constituent, i.e., that `Z` indeed covers `N` and `M`.

- if we have a failed precedence or co-occurrence constraint on any two categories within a constituent, we have to ignore all the successful ones.

Furthermore, the output of the program – consisting of the `pplus/pminus`-sets generated by the resolution process – can be varied according to the given threshold of failed constraints, in the extreme case reducing the process to finding only the positive characterizations.

The Appendix contains the full commented code of the implementation.

6. Conclusion

The representation of linguistic information by means of constraints is interesting both for knowledge representation (no implicit information, encapsulation) and from an implementational point of view as well (parsing is seen as constraint solving). Property Grammar offers a framework taking advantage of these characteristics and presenting several interesting insights for natural language processing. The first one concerns the generality and the robustness of the approach: constraints allow the introduction of the notion of characterization to replace that of grammaticality. The parsing process consists in verifying constraints for the possible set of categories associated with the input instead of trying to build a structure according to the grammar. One other interesting point concerns the integration of different information sources: the properties can concern all kind of information (prosodic, syntactic, pragmatic etc.).

A possible direct application of Property Grammars would be the development of tools to work with corpora. Since the modular and flexible nature of these simple constraints is close to the descriptions used by empirical linguists, it could be used to test or find examples of particular grammatical constructions (or even violations of principles) easily by focusing the analysis and using a shallow analysis for the rest. But obviously this would presuppose the development of a dedicated, more efficient constraint solver first.

Bibliography

Abdennadher, S. (1998). *Analyse von regelbasierten Constraintlösern*, PhD thesis, Ludwig-Maximilians-Universität München.

Abdennadher, S., Fruehwirth, T. and Meuss, H. (1996). On confluence of constraint handling rules, *Lecture Notes in Computer Science* **1118**: 1–15.

Archangeli, D. and Langendoen, D. (eds) (1997). *Optimality Theory*, Blackwell.

Bès, G. and Blache, P. (1999). Propriétés et analyse d'un langage, *Actes de TALN'99*.

Blache, P. (1999). Filtering and fusion: A technique for parsing with properties, *Proceedings of NLPRS'99*.

Blache, P. and Paquelin, J.-L. (1996). Active constraints for a direct interpretation of hpsg, *Proceedings of HPSG'96*.

Borsley, R. (1996). *Modern Phrase Structure Grammars*, Blackwell.

Duchier, D. and Thater, S. (1999). Parsing with tree descriptions: a constraint based approach, *Proceedings of* NLULP'99.

Frühwirth, T. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1–3): 95–138. Special Issue on Constraint Logic Programming.

Götz, T. and Meurers, D. (1997). Interleaving universal principles and relational constraints over typed feature logic, *Proceedings of the ACL/EACL Conference '97*, Association for Computational Linguistics, Madrid, Spain, pp. 1–8.

Günthner, F. (1988). Features and München.

Intelligent Systems Laboratory (1995). SICStus Prolog user's manual.

Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints*, MIT Press.

Maruyama, H. (1990). Structural disambiguation with constraint propagation, *Proc. of the 28th Annual Meeting of the Association for Computational Linguistics. Pittsburgh, PA, 1990*, pp. 31–38.

Meurers, D. and Minnen, G. (1998). Off-line constraint propagation for efficient hpsg processing, *in* G. Webelhuth, J.-P. Koenig and A. Kathol (eds), *Lexical and Constructional Aspects of Linguistic Explanation*, CSLI.

Sag, I. and Wasow, T. (1999). *Syntactic Theory. A Formal Introduction*, CSLI.

Shieber, S. M. (1992). *Constraint–Based Grammar Formalisms*, MIT Press.

Appendix: The Implementation of Property Grammars in CHR

```
:- use_module(library(chr)).
:- use_module(library(lists),[sublist/2,member/2,append/3]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Constraint "Solver"
%
% Input:  a list of lexical entries
%         predicate process/1
%         example: process([la,fille,la,plus,jolie]).
%
% Output: the sets P+ and P- for all subsets of categories
%         and all models and the final constraint store
%
% Currently the grammar is built into this file, but it is
% easy to change this to some general procedure or change
% the example. Look for the predicate post_grammar/0 and
% lex/2.
%
% Note that the dependency constraints from the grammar are
% not used for anything at the moment since there is no
% information which makes it worthwhile checking them. If we
% just test whether we could add a dependency link, we could
% simply use implicational constraints which are already
% present.
%
% We do not create P+ and P- sets for singleton sets of
% categories although this might be important in the future?
%
%============================================================
%
% Description of the program:
%
% First of all we build the projection lines of the
% categories gained by lexical lookup by propagation via the
% alphabet constraint.
%
% In general the CHR program looks at two catagory
```

```
% constraints and - if they are in the same model - uses
% them together with a constraint from the grammar, as the
% head of constraint propagation rules. The constraints
% propagated contain the information whether the pair of
% categories satisfy the constraint from the grammar or
% not. In this way, we build a complete table of results on
% all pairs of categories.
%
% Since in the end we are interested in bigger chunks of
% lexical items then just two, we have to do some
% post-processing to interpret the entries in the table in
% the right way. For example, if a precedence constraint
% succeeds on some pairs of categories, this does not imply
% that it also succeeds for the entire triple. So, one
% violation among all the pairs is sufficient to cause
% failure of that constraint.
%
% I implemented this in a pretty nasty Prolog routine which
% collects all failed and successful constraints on the
% elements of any subset of the numeration of the input and
% then massages them into shape. This predicate is highly
% wasteful and has to be reimplemented. Maybe the entire
% approach of looking at pairs of categories has to be
% abandoned?
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% name the constraint handler
handler propgrams.

% declare all constraints
constraints cat/2, pplus/4, pminus/4,
            obli/3, alph/3, impl/4, prec/4, cooc/4, depe/4.
% cat(X,Y)
%     X is a category and Y its position in the numeration
% pplus(A-Id,Name,X-N,Y-M)
% ppminus(A-Id,Name,X-N,Y-M)
%     pplus  are the successful constraints
%     pminus are the failed constraints
%     A is the module name, Id the identifier of the
```

```
%     constraint, Name is the type of constraint
%     X-N, Y-M are categories with their numeration
% The following are the "real" linguistic constraints from
% the grammar:
% obli(Mod,Cat,Id)
%     Mod is the module and Cat is the obligatory category,
%     Id the identifier
% alph(Mod,Cats)
%     Mod is the module and Cats is the list of possible
%     categories
% impl(Mod,Cat1,Cat2,Id)
%     Mod is the module and Cat1 is the antecedent, cat2 the
%     succedent of the implication, Id the identifier
% prec(Mod,Cat1,Cat2,Id)
% cooc(Mod,Cat1,Cat2,Id)
% depe(Mod,Cat1,Cat2,Id)
%     parallel to the above

% enable the code for CHR debugging
% option(debug_compile,on).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% CONSTRAINT HANDLING RULES
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% this rules projects the input categories within their
% module if we have a corresponding alphabet constraint
project @ cat(X,N), alph(Cat,Alph,_) ==>
        member(X,Alph) |
        cat(Cat,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% We build a complete table of pplus/4, pminus/4 constraints
% for all pairs of categories matching the heads of our
% propagation rules. These constraints store the information
% about the interaction of any two categories with the
% resulting rule and model which licensed them.
```

```
%
% Generally, in all the rules dealing with the constraints
% from the grammar, we apply the rule if we can find two
% categories within the constraint store which are in the
% same module and a constraint which checks some
% configuration on them. This last constraint triggers the
% rule since we declared all other constraints to be
% passive. Note that this assumes that we did all possible
% projections in a model before we post the other
% constraints.
%
% Since, if two category constraints match the head of a
% rule, they do so in either order, we use the guard to
% force the linear order as it appears in the input string.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Obligatory constraints:
% if we have a model which matches an obligatory constraint,
% then we also have to have a category constraint with the
% right Cat.
        obli @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3,
        cat(Z,M)#I4, obli(Z,Cat,Id) ==>
        N < M |
        ( ( X == Cat ; Y == Cat ) ->
            pplus(Z-Id,obli,X-N,Y-M)
        ;
            pminus(Z-Id,obli,X-N,Y-M)
        )
        pragma passive(Id1),passive(Id2),
               passive(Id3),passive(Id4).
% (Classical) Implicational constraints:
% succeeds if both anteceedent and succedent are present
impl @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
       impl(Z,Cat1,Cat2,Id) ==>
        N < M |
        ( ( (X == Cat1, Y \== Cat2);
            (X \== Cat2, Y == Cat1) )  ->
            pminus(Z-Id,impl,X-N,Y-M)
        ;
```

```
            pplus(Z-Id,impl,X-N,Y-M)
        )
        pragma passive(Id1),passive(Id2),
               passive(Id3),passive(Id4).
% Precedence constraints:
% fails only if both category constraints appear in the
% wrong order
prec @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
       prec(Z,Cat1,Cat2,Id) ==>
        N < M |
        ( ( X == Cat2, Y == Cat1 )  ->
            pminus(Z-Id,prec,X-N,Y-M)
        ;
            pplus(Z-Id,prec,X-N,Y-M)
        )
        pragma passive(Id1),passive(Id2),
               passive(Id3),passive(Id4).
% Cooccurrence constraints:
% fail if both category constrainst appear at the same time
cooc @ cat(X,N)#I1, cat(Z,N)#I2, cat(Y,M)#I3, cat(Z,M)#I4,
       cooc(Z,Cat1,Cat2,Id) ==>
        N < M |
        ( ( ( X == Cat1, Y == Cat2 );
            ( X == Cat2, Y == Cat1 ) )   ->
            pminus(Z-Id,cooc,X-N,Y-M)
        ;
            pplus(Z-Id,cooc,X-N,Y-M)
        )
        pragma passive(Id1),passive(Id2),
               passive(Id3),passive(Id4).

% some rules for absorbing spurious double constraints
absorb1 @ cat(X,N) \ cat(X,N) <=> true.
absorb2 @ pplus(X,Y,A,B) \ pplus(X,Y,A,B) <=> true.
absorb3 @ pminus(X,Y,A,B) \ pminus(X,Y,A,B) <=> true.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% GRAMMAR
```

```
% A simple collection of basic constraints and a
% lexicon. The lexicon are standard prolog facts.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the lexicon
% lex(+Word,+Category)
lex(la,det).
lex(fille,n).
lex(plus,adv).
lex(jolie,adj).

% this predicate justs posts the grammar constraints. beware
% that the projections within a model have to be done before
% any other processing.  the constrainst can be identified
% uniquely by their first and last arguments: the model and
% an identifier, i.e., a number.
post_grammar :-
        alph(np,[det,n,adjp,sup],1),
        alph(adjp,[adj,adv],1),
        alph(sup,[det,adv,adj],1),
        obli(np,n,2),
        obli(adjp,adj,2),
        obli(sup,adj,2),
        impl(np,n,det,3),
        impl(sup,adj,det,3),
        impl(sup,adj,adv,4),
        prec(np,det,n,4),
        prec(np,det,adjp,5),
        prec(np,det,sup,6),
        prec(np,n,adjp,7),
        prec(np,n,sup,8),
        prec(adjp,adv,adj,3),
        prec(sup,det,adj,5),
        prec(sup,det,adv,6),
        prec(sup,adv,adj,7),
        cooc(np,adjp,sup,9),
        depe(np,det,n,10),
        depe(np,adjp,n,11),
        depe(np,sup,n,12),
```

```
        depe(adjp,adv,adj,4),
        depe(sup,det,adj,8),
        depe(sup,adv,adj,9).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Processor (not CHR any more but standard Prolog):
% We enumerate and post the input category-constraints
% followed by searching the database of pplus/4 and pminus/4
% constrainst for the generation of answers
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% process(+ListOfLexicalItems)
        process(In) :-
        numeration(In,1),
        post_grammar,
        generate_output.

% numeration(+List,+Num)
% posts cat/2 constrainst for all lexical elements in List and
% associates their string position with them.
numeration([],_).
numeration([H|T],N):-
        lex(H,Cat),
        cat(Cat,N),
        N1 is N + 1,
        numeration(T,N1).

% finds all category constraints and transforms them into a
% list of Cat-Num pairs which is then used to generate the
% possible subsets which in turn triggers reporting the
% results.
%
% the alphabet constraints, i.e., identity 1 is never part
% of the solution set since it is always true.
generate_output :-
        findall_constraints(cat(_X,_Y),List),
```

```
        extract_categories(List,Cats),
        powerset(Cats,PowCats),
        nl,
        write('+++++++++++++++++++++++++++++++++++++++'),nl,
        report_results(PowCats).

% extract_categories(+ListOfCatConstraints,-ListOfCat-NumPairs)
% careful! we reverse the order of the Cat-Num pairs such
% that we can use keysort on the list.
extract_categories([],[]).
extract_categories([cat(X,Y)#_|T],[Y-X|R]):-
        extract_categories(T,R).

% powerset(+ListOfCat-NumPairs,-ListOfSubLists)

% creates a list of lists such that the memebers are the
% non-trivial subsets of the list such that no two items in
% any sublist have the same string position.
powerset(L,PowerSet) :-
        findall(Sub1,
                (   sublist(Sub,L),
%                   length(Sub,3),  % for tests only sets
                                    % with card N
                    length(Sub,K),
                    K > 1,
                    \+ (   member(N-X,Sub),
                           member(M-Y,Sub),
                           X \== Y,
                           N == M
                       ),
                    sort(Sub,Sub1)
                ),
                Pow),
        sort(Pow,PowerSet).

% report_results(+ListOfSubLists)

% we take each set, get all pairs of appearing categories if
% they are in the same model and collect all positive and
% negative constraints attached to them. All of these
```

```
% results are collected in a list, post processed, sorted,
% and passed to the predicate report_results/1 for output.
%
% nasty, nondeterministic => has to be changed!!
report_results([]).
report_results([Set|T]):-
        findall(Model,
          ( find_constraint(alph(Model,List,_),_),
            all_member(Set,List)
          ),
                ModList),
        ( ModList == [] ->          % No common model
            Solutions = []
        ;
            findall(ResList,
              ( member(Mod,ModList),
                findall(res(Mod,Set,LPlus-LMinus),
                  ( member(N-X,Set),
                    member(M-Y,Set),
                    \+ X-N == Y-M,
                    N < M,
                    findall_constraints(
                       pplus(Mod-_,_,X-N,Y-M),LPlus),
                    findall_constraints(
                       pminus(Mod-_,_,X-N,Y-M),LMinus),
                    \+ (LPlus == [], LMinus == [])
                   ),
                        ResList)
                ),
                    Sols),
            post_process(Sols,Solutions)
        ),
        ( Solutions == [] ->
            true                    % switch reporting of
                                    % empty solutions
%           report_empty(Set)       % on or off
        ;
            report(Solutions)
        ),
        report_results(T).
```

```
% all_member(+ListOfCats,+AlphabetList)
% checks wether all elements of the ListOfCats are members
% of one model
all_member([],_).
all_member([_-H|T],L):-
        member(H,L),
        all_member(T,L).


% post_process(+ListOfResults,-ListOfResults)
% this is the awful predicate ;-)
% we flatten the list of results to one result which has
% been massaged into shape....
% we recurse on the list of results from report_results/1.
post_process([],[]).
post_process([H|T],[res(Mod,Set,LPlus-LMinus)|Out]) :-
        post_proc(H,Mod,Set,[],LPlus,[],LMinus),
        post_process(T,Out).

% post_proc(+ListOfResults,-Model,-Set,
%           -ListOfPosCons,-ListOfNegCons)
% the recursive case just flattens the results into two
% lists: the successfull constraints and the failed ones.
% the base case takes each lists and interprets its members
% in the following way:
% - if we have a successful obligatory or implicational
%   constraint on any two categories from the set, we can
%   ignore all the failed ones
% - if we have a failed precedence or co-occurrence
%   constraint on any two categories from the set, we have
%   to ignore all the successful ones
% finally, we strip all the ids generated by the CHR part
% from the constraints and sort the resulting lists.
post_proc([],_,_,LPlus,NewLPlus,LMinus,NewLMinus) :-
        findall(N1,
                member(pplus(_-N1,obli,_,_)#_,LPlus),
               LN1),
        remove(obli-LN1,LMinus,LMinus1),
        findall(N2,
```

```
                member(pplus(_-N2,impl,_,_)#_,LPlus),
                LN2),
        remove(impl-LN2,LMinus1,LMinus2),
        findall(N3,
                member(pminus(_-N3,prec,_,_)#_,LMinus2),
                LN3),
        remove(prec-LN3,LPlus,LPlus1),
        findall(N4,
                member(pminus(_-N4,cooc,_,_)#_,LMinus2),
                LN4),
        remove(cooc-LN4,LPlus1,LPlus2),
        strip_ids(LPlus2,[],LPlus3),
        strip_ids(LMinus2,[],LMinus3),
        sort(LPlus3,NewLPlus),
        sort(LMinus3,NewLMinus).
post_proc([res(Mod,Set,P1-M1)|T],Mod,Set,Pin,Pout,
                                            Min,Mout) :-
        append(Pin,P1,P2),
        append(Min,M1,M2),
        post_proc(T,Mod,Set,P2,Pout,M2,Mout).

% remove(+Keyword-RuleId,+ListOfCons,-ListOfCons)
% takes as input a list of constraints, a keyword and the
% list of identities of constraints from the grammar and
% removes them from the list.
remove(_-[],In,In) :- !.
remove(obli-N,In,Out) :-
        removem(In,obli,N,Out).
remove(impl-N,In,Out) :-
        removem(In,impl,N,Out).
remove(prec-N,In,Out) :-
        removep(In,prec,N,Out).
remove(cooc-N,In,Out) :-
        removep(In,cooc,N,Out).

% removem(+ListOfConstraints,+Keyword,
%         +ListOfIdentities-LotsOfCons)
% removes negative constraints matching the keyword and an
% identity
removem([],_,_,[]).
```

```
removem([pminus(_-N,X,_,_)#_|T],X,LN,Out) :-
        member(N,LN),
        !,
        removem(T,X,LN,Out).
removem([pminus(A-M,Y,C1,C2)#I|T],X,LN,
        [pminus(A-M,Y,C1,C2)#I|Out]) :-
        ( X \== Y ; \+ member(M,LN) ),
        !,
        removem(T,X,LN,Out).
% removep(+ListOfConstraints,+Keyword,
%         +ListOfIdentities-LotsOfCons)
% removes positive constraints matching the keyword and an
% identity
removep([],_,_,[]).
removep([pplus(_-N,X,_,_)#_|T],X,LN,Out) :-
        member(N,LN),
        !,
        removep(T,X,LN,Out).
removep([pplus(A-M,Y,C1,C2)#I|T],X,LN,
        [pplus(A-M,Y,C1,C2)#I|Out]) :-
        ( X \== Y ; \+ member(M,LN) ),
        !,
        removep(T,X,LN,Out).

% strip_ids(+ListOfCons,+Acc,-Acc)
% removes the trailing Identities form the CHR constraints
strip_ids([],_,[]).
strip_ids([H#_|R],In,Out):-
        H =.. [_,_-N|_],
        ( member(N,In) ->
            NewIn = In,
            Out = Out1
        ;
            NewIn = [N|In],
            Out = [H|Out1]
        ),
        strip_ids(R,NewIn,Out1).


% report(+ListOfResults)
```

```
% outputs the Model, the categories and lists of the
% positive and negative constraints.
report([]) :-
        write('++++++++++++++++++++++++++++++++++++++++'),
        nl,nl.
report([res(Mod,Set,H1-H2)|T]) :-
        keysort(Set,Set1),
        write('Categories: '),write_set(Set1),nl,
        write('P+( '),write(Mod), write(' ) = [ '),
        write_listp(H1),
        write(' ]'),nl,
        write('P-( '),write(Mod), write(' ) = [ '),
        write_listm(H2),
        write(' ]'),nl,nl,
        report(T).

% report_empty(+L)
% this usually means that we had a subset of categories
% which did not have a common model
report_empty(L) :-
        write('The following categories did not'),
        write(' produce any results: '),nl,
        write('    '),write(L),nl,nl,
        write('++++++++++++++++++++++++++++++++++++++++'),
        nl,nl.

% outputs the list of categories we are looking at.
% reverses the elements back.
write_set([]) :- write('ERROR....'),!,fail.
write_set([H-N]):-
        write(N-H),!.
write_set([H-N|T]):-
        write(N-H), write(' and '),
        write_set(T).

% outputs the identities of positive constraints
write_listp([]).
write_listp([pplus(_-K,_,_,_)]) :-
        write(K), !.
write_listp([pplus(_-K,_,_,_)|T]):-
```

```
        write(K), write(', '),
        write_listp(T).
% outputs the identities of negative constraints
write_listm([]).
write_listm([pminus(_-K,_,_,_)]) :-
        write(K), !.
write_listm([pminus(_-K,_,_,_)|T]):-
        write(K), write(', '),
        write_listm(T).
%
% end of file
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

2

Chart Parsing as Constraint Propagation

FRANK MORAWIETZ

Seminar für Sprachwissenschaft
Universität Tübingen
Wilhelmstrasse 113
72074 Tübingen, Germany

frank@sfs.nphil.uni-tuebingen.de

ABSTRACT. In this paper, parsing as deduction and constraint programming are brought together to outline a procedure for the specification of constraint-based chart parsers. Extending the proposal in Shieber et al. (1995) which provides a meta interpreter for several deduction systems, we show how to directly realize the inference rules for chart parsers as Constraint Handling Rules (Frühwirth; 1998) by viewing the items of a conventional chart parser as constraints and the constraint base as a chart. This allows the direct use of the constraint resolution process to parse sentences.

1. INTRODUCTION: PARSING AS DEDUCTION

The parsing-as-deduction approach proposed in Pereira and Warren (1983) and extended in Shieber et al. (1995) and the parsing schemata defined as special deduction systems in Sikkel (1997) are well established parsing paradigms in computational linguistics. Their main strengths are their flexibility and level of abstraction concerning control information inherent in parsing algorithms. This allows for a uniform presentation and the comparison of a variety of parsing algorithms. Furthermore, they are easily extensible to more complex formalisms, e.g., augmented phrase structure rules or the ID/LP format.

Constraint Programming (CP) has been used in computational linguistics in several areas, for example in (typed) feature-based systems based on Oz (Smolka; 1995), or conditional constraints (Matiasek; 1994), or advanced compilation techniques (Götz and Meurers; 1997) or specialized constraint solvers

(Manandhar; 1994). But to my knowledge, none of these approaches uses constraint programming techniques to implement standard chart parsing algorithms directly in a constraint system.

In this paper, I will bring these two paradigms together, thereby showing how to implement algorithms from the parsing-as-deduction scheme by viewing the parsing process as constraint propagation.

The core idea is that the items of a conventional chart parser are constraints on labeled links between the words and positions of an input string. Then the inference rules allow for the deduction of new constraints, again labeled and spanning parts of the input string, via constraint propagation. The resulting constraint store represents the chart which can be accessed to determine whether the parse was successful or to reconstruct a parse tree.

While this may seem a trivial observation it allows for a rapid and flexible method of implementation. The goal is not necessarily to build the fastest parser, but rather to build – for an arbitrary algorithm – a parser fast and perspicuously. The advantage of our approach compared to the one proposed in Shieber et al. (1995) is that we do not have to design a special deduction engine and we do not have to handle chart and agenda explicitly, i.e., the process can be used in any constraint-based formalism which allows for constraint propagation and therefore can be seamlessly integrated into existing applications within the CHR framework.

The paper proceeds by reviewing the parsing-as-deduction approach and a particular way of implementing constraint systems, Constraint Handling Rules (CHR) as presented in Frühwirth (1998).[1] Then it shows how to implement several parsing algorithms very naturally with constraint propagation rules before concluding with an outlook on how to extend the technique to more advanced applications.

Parsing in general is the process of analyzing the structure of an input by decomposing it into subparts according to some given grammar. If the process can be completed successfully, the input is a valid expression of the language defined by the grammar.

Therefore, to define how to parse a string, one needs the following ingredients:

- a way of initializing the parsing process, i.e., some statements about the acceptability of (parts of) strings which are used to start the parsing process;
- some rules which allow to derive new statements from existing ones, e.g., the initial ones, which actually define the parsing mode;

[1] CHR are provided as a package for SICStus Prolog and ECLiPSe.

- a way of controlling the process, i.e., of deciding which statements to try next for possible rule applications;
- and a way of identifying a final statement, i.e., a statement which means that we successfully decomposed the string.

Parsing-as-deduction basically takes formulas of a logic and uses its axioms and deduction rules to drive the parsing process.

Obviously, parsing-as-deduction is an instantiation of the scheme given above. It derives statements about the acceptability of (parts of) strings from other such statements via inference rules, starts the process with axioms and stops when the possibilities for further deductions have been exhausted or a predefined goal has been reached. What does not have to be defined and which actually is seen as an advantage for the specification of parsing algorithms, is the control information. This search procedure is not relevant for the specification of a specific algorithm, but rather for its realization or implementation.

I will recall some basic definitions for convenience. The notations and the three basic algorithms are directly taken from Shieber et al. (1995).

As usual, strings w result from concatenation of symbols from some alphabet set Σ, i.e., $w \in \Sigma^*$. We refer to the decomposition of such a string into its alphabet symbols with indices. We fix this notation using $w = w_1 \ldots w_n$, for the given input string. A *grammatical deduction system* or, in Sikkel's terminology a *parsing schema*, is defined as a set of deduction schemes and a set of axioms. These are given with the help of *formula schemata* which contain (syntactic) meta-variables which are instantiated with concrete terms on application of the rules. A deduction scheme R has the general form

$$\frac{A_1 \; \ldots \; A_n}{C} \; \langle \text{ side conditions on } A_1 \ldots A_n, C \rangle$$

where the A_i and C are formula schemata. The A_i are called antecedents and C the consequence. Note that the deduction schemes may refer to the string positions, i.e., the indices of the alphabet symbols of the input string, in their side conditions. We say that such a scheme R *applies* if we have as antecedents a sequence of formulas $F_1, \ldots, F_k$, $k \geq n$ such that each F_i, $1 \leq i \leq n$ matches A_i under an appropriate substitution of terms for the meta-variables and the side conditions are satisfied. For a given deduction system, the derivation of a formula F from assumptions $A_1, \ldots, A_p$ ($A_1, \ldots, A_p \vdash F$) is defined to be a sequence of formulas $F_1, \ldots, F_q$ such that $F = F_q$ and each F_i, $1 \leq i \leq q$, is either an instance of an axiom, one of the A_j, $1 \leq j \leq p$, or the result of applying a deduction scheme with $F_i = C$ such that there exist the necessary antecedents $A_{i_1}, \ldots, A_{i_k}$, $i_1, \ldots, i_k \leq i$.

To improve the performance of the implementation of a parser, one also wants to record work which has been done such that recomputations of (partial) results can be avoided. To simply store recognized substrings is not enough since one cannot differentiate between for example two identical occurrences of substrings. Therefore the statements are enriched by indexing the category recognized, i.e., the terminal or nonterminal symbol labeling a node in the parse tree, with the position(s) spanned by that category. These statements are then stored in a well-formed substring table, also called a chart. The chart does not play a direct role for the specification of grammatical deduction systems but since I will present examples below, it is necessary to introduce the complex formulas or statements used there.

Unless specified differently, we assume that we are given a context-free grammar $\mathcal{G} = \langle N, T, S, P \rangle$ with nonterminals N, terminals T, start symbol S and set of productions P.[2] Each production is of the form $A \longrightarrow \alpha$ with $A \in N$, $\alpha \in (N \cup T)^*$.

As a running example, I will use the simple PP-attachment grammar given in Fig. 1. It is left to the reader to calculate example derivations for the three example algorithms for a sentence such as *John hit the dog with the stick*.

$$\mathcal{G} = \langle N, T, S, P \rangle$$
$$NT = \{\text{S, NP, VP, V, PP, P, PN, Det, N1, N}\}$$
$$T = \{\textit{hit, John, dog, stick, with, the}\}$$

and P as given below:

S	$\longrightarrow$	NP VP	N1	$\longrightarrow$	N1 PP
VP	$\longrightarrow$	V NP	V	$\longrightarrow$	*hit*
VP	$\longrightarrow$	V NP PP	PN	$\longrightarrow$	*John*
PP	$\longrightarrow$	P NP	N	$\longrightarrow$	*dog*
NP	$\longrightarrow$	PN	N	$\longrightarrow$	*stick*
NP	$\longrightarrow$	Det N1	P	$\longrightarrow$	*with*
N1	$\longrightarrow$	N	Det	$\longrightarrow$	*the*

FIGURE 1. Example Grammar: PP-attachment

All the parsing systems used in this paper are then defined by specifying a class of items, a set of axioms, a set of inference rules and a subset of the items,

[2] For Earley's algorithm we also assume a new start symbol S' which is not in N.

the goals or final items. For better readability, I follow Shieber et al. in using the familiar dotted items.

Notational conventions are: n for the length of the string to be parsed, for arbitrary formulas or nonterminals $A, B, C, \ldots$, for terminals $a, b, c, \ldots$, ε for the empty string and $\alpha, \beta, \gamma, \ldots$ for strings of terminals and nonterminals. Formulas used in parsing will also be called items or edges.

The three example algorithms, we will use to illustrate our technique can now be presented as grammatical deduction systems as given in Tab. 1 (taken from Shieber et al.; 1995). In the table i, j, k are always between 1 and n.

Although I assume familiarity with these three basic algorithms, I will briefly comment on the intended interpretations of the respective items. In the case of the naive bottom-up algorithm, the items are of the form $[j, \alpha \bullet]$ with the interpretation that we have a stack α of items we recognized reaching up to position j (which implies that we are still looking forward to parsing the substring starting at j). In the case of the top-down algorithm, the items $[j, \bullet \beta]$ mean that we are looking for categories β starting at position j (implying that we already recognized the string up to that position). The items in Earley's algorithm $[i, j, A, \alpha \bullet \beta]$ are to be interpreted that we recognized a substring from i to j with the rule $A \longrightarrow \alpha\beta$ having already found α, but looking for categories β.

2. Constraint Handling Rules

Constraint programming is an elegant, declarative way of solving problems with higher-level programming languages. It represents a generalization of logic programming in the sense that it allows the (logical) specification of a problem on an arbitrary domain and tries to solve it with a combination of inference and search. A complete introduction to constraint programming can be found for example in Marriott and Stuckey (1998).

There are several constraint programming environments available. The most recent and maybe the most flexible is the Constraint Handling Rules (CHR) package included in both SICStus Prolog and ECLiPSe (Intelligent Systems Laboratory; 1995; Frühwirth; 1998). All of these CP systems provide mechanisms for solving constraints. They maintain a constraint base or store which is continually monitored for possible rule applications, i.e., whether there is enough information present to successfully use a rule to simplify constraints or to derive new constraints. Whereas usually one deals with a fixed constraint domain and a specialized solver, CHR is an extension of the Prolog language which allows for the specification of user-defined constraints and arbitrary solvers. The

TABLE 1. Parsing algorithms as Grammatical Deduction Systems

	Bottom-Up	**Top-Down**	**Earley**
Items	$[j, \alpha \bullet]$	$[j, \bullet \beta]$	$[i, j, A, \alpha \bullet \beta]$
Axiom	$[0, \bullet]$	$[0, \bullet S]$	$[0, 0, S', \bullet S]$
Goal	$[n, S \bullet]$	$[n, \bullet]$	$[0, n, S', S \bullet]$
Scan	$\dfrac{[j, \alpha \bullet]}{[j+1, \alpha w_{j+1} \bullet]}$	$\dfrac{[j, \bullet w_{j+1}\beta]}{[j+1, \bullet \beta]}$	$\dfrac{[i, j, A, \alpha \bullet w_{j+1}\beta]}{[i, j+1, A, \alpha w_{j+1} \bullet \beta]}$
Predict		$\dfrac{[j, \bullet B\beta]}{[j, \bullet \gamma\beta]} \langle B \longrightarrow \gamma \rangle$	$\dfrac{[i, j, A, \alpha \bullet B\beta]}{[j, j, B, \bullet \gamma]} \langle B \longrightarrow \gamma \rangle$
Complete	$\dfrac{[j, \alpha\gamma \bullet]}{[j, \alpha B \bullet]} \langle B \longrightarrow \gamma \rangle$		$\dfrac{[i, k, A, \alpha \bullet B\beta] \quad [k, j, B, \gamma \bullet]}{[i, j, A, \alpha B \bullet \beta]}$

strength of the CHR approach lies in the fact that it allows for multiple (conjunctively interpreted) heads in rules, that it is flexible and that it is tightly and transparently integrated into the Prolog engine.

In CHR constraints are just distinguished sets of (atomic) formulas. CHR allow the definition of rule sets for constraint solving with three types of rules:

Firstly simplification rules (symbol <=>) which replace a number of constraints in the store with new constraints; secondly propagation rules (symbol ==>) which add new constraints to the store in case a number of constraints is already present; and thirdly "simpagation" rules (symbol <=> in combination with a \ in the head of the rule) which replace only those constraints with new ones which are to the right of the backslash.

To improve efficiency, rules can have guards and pragma declarations. A guard (separated from the rest of the body by a |) is a condition which has to be met before the rule can be applied.[3] A pragma declaration affects the way the rule is compiled. Our approach will only introduce the *passive* declaration, so we ignore the other possibilities. Declaring a constraint in the head passive simply means that this constraint will not trigger the rule on its own.

We cannot go into the details of the formal semantics of CHR here. The interested reader is referred to Frühwirth (1998). Since I will refer back to it let us just note that logically, simplification rules are equivalences and propagation rules are implications if their guard is satisfied. Simpagation rules are special cases of simplification rules. Soundness and completeness results for CHR are available (Abdennadher et al.; 1996; Abdennadher; 1998).

3. Parsing as Constraint Propagation

The basic observation which turns parsing-as-deduction into constraint propagation is simple: items of a chart parser are just special formulas which are used in an inference process. Since constraints in constraint programming are nothing but atomic formulas and constraint handling rules nothing but inference rules, the connection is immediate.

In more detail, I will present in this section how to implement the three parsing algorithms given in Tab. 1 in CHR and discuss the advantages and drawbacks of this approach. Since CHR are integrated in SICStus Prolog, I will present constraints and rules in Prolog notation, i.e., strings starting with uppercase letters stand for variables and words with lowercase ones for atoms.

We use the following two types of constraints. The constraints corresponding to the items will be called `edge` constraints, see Tab. 3. They have two arguments in case of the two naive algorithms and five in the case of Earley's algorithm, i.e.,

```
edge(X,N)
```

[3] A more careful distinction separates *ask* and *tell* guards. Ask guards have to be entailed by the constraint store whereas tell guards only have to be consistent with it (Saraswat; 1993). This is also part of CHR, but not used in our approach.

TABLE 2. Parsing systems as CHR programs: Items, Axioms & Goals

	Bottom-Up	**Top-Down**
Items	`edge(X,N)`	`edge(X,N)`
Axiom	`edge([],0)`	`edge([s],0)`
Goal	`edge([s],Len)`	`edge([],Len)`

	Earley
Items	`edge(A,Alpha,Beta,I,J)`
Axiom	`edge(sprime,[],[s],0,0)`
Goal	`edge(sprime,[s],[],0,Len)`

means in the case of the bottom-up algorithm that we have recognized a list of categories `X` up to position `N`, in the case of the top-down algorithm that we are looking for a list of categories `X` starting at position `N` and in the case of Earley's algorithm

```
edge(A,Alpha,Beta,I,J)
```

means that we found a substring from `I` to `J` by recognizing the list of categories `Alpha`, but we are still looking for a list of categories `Beta` to yield category `A`. The second constraint `word/2` is only used for efficiency. Since in grammar rules we do not use the lexical items directly but rather their categories, I do not use edges to represent that we found a word, but the `word` constraint, as will become apparent in the scanning rules. This avoids using constraints on the input *words* (compared to their category) in the other inferences needlessly.

The grammars are given as Prolog facts, lexical items as

```
lex(Word,Category)
```

and grammar rules as

```
rule(RHS,LHS)
```

where `RHS` is a list of categories representing the right hand side and `LHS` is a single category representing the left hand side of the rule.

The resulting algorithms are then simple to implement by specifying the inference rules as constraint propagation rules, the axioms and the goal items. A summarization is presented in Tab. 3.

It is obvious how to link the code for the axioms and goals given to the definitions in Tab. 1. Let us consider Earley's algorithm for a closer look as to what is going on in the CHR propagation rules.

In principle the inference rules are translated into CHR in the following way: The antecedents are transformed into constraints appearing in the head of the propagation rules, the side conditions into the guard and the consequence is posted in the body.[4]

In the scanning step, we can move the head of the list of categories we are looking for to those we already recognized in case we have an appropriately matching edge and word constraint in our constraint store. The result is posted as a new edge constraint with the positional index appropriately incremented.

The prediction step is a little more complex. There is only one head in a rule, namely an edge which is still looking for some category to be found. If one can find rules with a matching LHS, we collect all of them in a list and post the appropriate fresh edge constraints for each element of that list (predicate `post_ea_edges/3` which posts edges of the following kind:

```
edge(LHS,[],RHS,J,J).
```

The collection of all matching rules in a call to `findall/3` is necessary since CHR are a committed choice language. One cannot enumerate all solutions via backtracking. If there are no matching rules, i.e., the list of RHSs we found is empty, the corresponding tests avoid vacuous predictions and therefore nontermination of the predictor.

Lastly, the completion step is a pure propagation rule which translates literally. The two antecedents are in the head and the consequence in the body with appropriate instantiations of the positional variables and with the movement of the category recognized by the passive edge from the categories to be found to those found. Additionally, the rule contains a pragma declaration on the first head which is declared to be *passive*. While I cannot go into detail here, this declaration leads to the fact that this rule is only tried for inference if a constraint matching the non-passive head is posted. More on pragma declarations can be found in Frühwirth (1998).

In the table there is one more rule, called an absorption rule. It discovers those cases where we posted an edge constraint which is already present in the

[4]Note that calling a CHR constraint as a Prolog goal means to insert it into the constraint store such that it becomes available for constraint resolution steps.

TABLE 3. Parsing systems as CHR programs: Inference Rules

Scan

Bottom-Up
```
edge(Stack,N), word(N,Cat-_Word) ==>
      N1 is N+1,
      edge([Cat|Stack],N1).
```
Top-Down
```
edge([Cat|T],N), word(N,Cat-_Word) ==>
      N1 is N+1,
      edge(T,N1).
```
Earley
```
edge(A,Alpha,[Cat|Beta],I,J),
             word(J,Cat-_Word) ==>
      J1 is J+1,
      edge(A,[Cat|Alpha],Beta,I,J1).
```

Predict

Top-Down
```
edge([LHS|T],N) ==>
      setof(RHS, rule(RHS,LHS), List) |
      post_td_edges(List,T,N).
```
Earley
```
edge(_A,_Alpha,[B|_Beta],_I,J) ==>
      setof(Gamma, rule(Gamma,B), List) |
      post_ea_edges(List,B,J).
```

Complete

Bottom-Up
```
edge(Stack,N) ==>
      setof(Rest-LHS,
            split(Stack,Rest,LHS), List) |
      post_bu_edges(List,N).
```
Earley
```
edge(A,Alpha,[B|Beta],I,K)#Id,
             edge(B,Gamma,[],K,J) ==>
      edge(A,[B|Alpha],Beta,I,J)
             pragma passive(Id).
```

Absorb

Bottom-Up
```
edge(L,N) \ edge(L,N) <=> true.
```
Top-Down
```
edge(L,N) \ edge(L,N) <=> true.
```
Earley
```
edge(A,Alpha,Beta,I,J) \
             edge(A,Alpha,Beta,I,J) <=>
      true.
```

chart and simply absorbs the newly created one. We come back to this point a little below.

Note that we do not have to specify how to insert edges into either a chart or into an agenda. The chart and the agenda are in fact represented by the constraint store and therefore built-in. We do not need a specialized deduction engine as was necessary for the implementation described in Shieber et al. (1995). In fact, the utilities needed are extremely simple, see Fig. 2.

```
parse(L) :-
         axiom,
         post_constraints(L, 0, Length),
         report(Length).

post_constraints([], Length, Length).
post_constraints([Word|String], InLength, Length) :-
         findall(Cat, lex(Word, Cat), Cats),
         post_words(Cats, InLength, Word),
         NewLength is InLength + 1,
         post_constraints(String, NewLength, Length).

post_words([],_Position,_Word).
post_words([Cat|Cats], Position, Word):-
        word(Position, Cat-Word),
        post_words(Cats, Position, Word).
```

FIGURE 2. The utilities for CHR-based deductive parsing

All we have to do for parsing (predicate `parse/1`) is to post the axiom[5] and on traversal of the input string to post the word constraints according to the lexicon of the given grammar. Then the constraint resolution process with the inference rules will automatically build a complete chart. The call to `report/1` will just determine whether there is an appropriate edge with the correct length in the chart and print that information to the screen.

Coming back to the issues of chart and agenda: the constraint store functions as chart and agenda at the same time since as soon as a constraint is added all rules are tried for applicability. If none apply, the edge will remain dormant

[5] `axiom/0` is a predicate which just calls the edge(s) defined in Tab. 3.

until another constraint is added which triggers a rule together with it.[6] So, the parser works incrementally and recursively tries all possible inferences for each constraint added to the store before continuing with the posting of new constraints from the `post_constraints/3` predicate. The way this predicate works at the moment is to traverse the string from left-to-right. It is trivial to alter the predicate such as to post the constraints from right-to-left or any arbitrary order chosen. This can be used to easily test different parsing strategies.

The testing for applicability of new rules also has a connection with the absorption rules. We absorb the newer edge since we can assume that all possible propagations have already been done with the old identical edge constraint so that we can safely throw the other one away.

As an example for the resulting chart, the output of an Earley-parse for *John hit the dog with the stick* assuming the grammar given in Fig. 1 is presented in Fig. 3.

The entire constraint store is printed to the screen after the constraint resolution process stops producing new edges. And furthermore, the order of the constraints actually reflects the order of the construction of the edges, i.e., the chart constitutes a trace of the parse at the same time. Naturally the given string was ambiguous but only a single solution is visible in the chart. This is due to the fact that we only did recognition. No explicit parse was built which could have differentiated between the two solutions. It is an easy exercise to either write a predicate to extract all possible parses from the chart or to alter the edges in such a way that an explicit parse tree is built during parsing.

By using a built-in deduction engine, one gives up control of its efficiency. As it turns out, this CHR based approach is slower than the specialized engine developed and provided by Shieber et al. (1995) by about a factor of 2, e.g., for a six word sentence and a simple grammar the parsing time increased from 0.01 seconds to 0.02 seconds on a LINUX PC running SICStus Prolog. This factor was preserved under 5 and 500 repetitions of the same parse. This may or may not be seen as a serious problem. However, there are two remarks I want to make. Firstly, speed was not the main issue in developing this setup, but rather simplicity and ease of implementation. And secondly it seems that the implementation of the attributed variables package of SICStus Prolog which is the basis for the CHR package, is too general to be maximally efficient. Gerald Penn (Penn; pc,i) reports two instances of special applications where the efficiency of the implementation was improved by using specific predicates and not the full library. On implementing co-routining in a typed feature language built

[6]Another way to "wake" a constraint is to instantiate any of its variables in which case it will be matched against the rules again. Since all our constraints are ground, this does not play a role here.

```
| ?- parse([john, hit, the, dog, with, the, stick]).

Input recognized.
 word(0,pn-john),
 word(1,v-hit),
 word(2,det-the),
 word(3,n-dog),
 word(4,p-with),
 word(5,det-the),
 word(6,n-stick),
 edge(sprime,[],[s],0,0),
 edge(s,[],[np,vp],0,0),
 edge(np,[],[det,n1],0,0),
 edge(np,[],[pn],0,0),
 edge(np,[pn],[],0,1),
 edge(s,[np],[vp],0,1),
 edge(vp,[],[v,np],1,1),
 edge(vp,[],[v,np,pp],1,1),
 edge(vp,[v],[np,pp],1,2),
 edge(np,[],[det,n1],2,2),
 edge(np,[],[pn],2,2),
 edge(vp,[v],[np],1,2),
 edge(np,[det],[n1],2,3),
 edge(n1,[],[n],3,3),
 edge(n1,[],[n,pp],3,3),
 edge(n1,[n],[pp],3,4),
 edge(pp,[],[p,np],4,4),
 edge(n1,[n],[],3,4),
 edge(np,[n1,det],[],2,4),
 edge(vp,[np,v],[],1,4),
 edge(s,[vp,np],[],0,4),
 edge(sprime,[s],[],0,4),
 edge(vp,[np,v],[pp],1,4),
 edge(pp,[p],[np],4,5),
 edge(np,[],[det,n1],5,5),
 edge(np,[],[pn],5,5),
 edge(np,[det],[n1],5,6),
 edge(n1,[],[n],6,6),
 edge(n1,[],[n,pp],6,6),
 edge(n1,[n],[pp],6,7),
 edge(pp,[],[p,np],7,7),
 edge(n1,[n],[],6,7),
 edge(np,[n1,det],[],5,7),
 edge(pp,[np,p],[],4,7),
 edge(vp,[pp,np,v],[],1,7),
 edge(s,[vp,np],[],0,7),
 edge(sprime,[s],[],0,7),
 edge(n1,[pp,n],[],3,7),
 edge(np,[n1,det],[],2,7),
 edge(vp,[np,v],[],1,7),
 edge(vp,[np,v],[pp],1,7) ?
```

FIGURE 3. Chart for *John hit the dog with the stick*

on top of Prolog, it turned out that a speed-up by a factor of about 50 could be achieved by using `when/2`, and on implementing typed feature structure unification that, by using some undocumented internal predicates directly rather than `get_atts/2` and `put_atts/2`, a speed-up of a little over a factor of 28 could be achieved with the ALE (Carpenter and Penn; 1998) Head-Driven Phrase Structure (HPSG) grammar. If we can apply similar techniques, the implementation may be able to improve on its performance.

The Shieber et al. paper contains more advanced parsing algorithms which can be transferred into the CHR setting. To transfer the parsing algorithm for unification grammars we simply include the necessary unifications explicitly into the body of the rules. Below we give the completion rule for unification grammars defined in such a way that the unification for the relevant categories `B` and `B2` computes the result `B1` and an explicit most general unifier `Sigma` which is then applied to the other categories `A`, `Alpha` and `Beta`.

```
edge(A,Alpha,[B|Beta],I,K), edge(B2,Gamma,[],K,J) ==>
        unify(B,B2,B1,Sigma),
        apply(Sigma,A,A1),
        apply(Sigma,Alpha,Alpha1),
        apply(Sigma,Beta,Beta1)       |
        edge(A1,[B1|Alpha1],Beta1,I,J).
```

Even the Shieber et al. approach to Combinatory Categorial Grammars can be transferred seamlessly. Consider for example forward application:

$$\frac{[X/Y,i,k] \quad [Y,k,j]}{[X,i,j]}$$

This results in the most trivial completion rule:

```
edge(X/Y,I,K), edge(Y,K,J) ==>
        edge(X,I,J).
```

I think that this short discussion shows convincingly how trivially the Parsing-as-Deduction algorithms can be transferred.

To sum up this section, the advantages of the approach lie in its flexibility and its availability for rapid prototyping of different parsing algorithms since it avoids any explicit handling of chart or agenda. While we used the examples from the Shieber et al. article, one can also implement all the different parsing schemata presented as deduction schemes in Sikkel (1997). This also includes advanced schemes such as left-corner or head-corner parsing, the refined Earley-algorithm proposed by Graham et al. (1980), or (unification-based) ID/LP parsing as defined in Morawietz (1995) (we just have to include guards to check for precedence, see Fig. 4), or any improved version of any of these. Furthermore, because of the logical semantics of CHR with their soundness and completeness, all correctness and soundness proofs for the algorithms can be directly applied to this constraint propagation proposal. The main disadvantage of the proposed approach certainly lies in its apparent lack of efficiency. One way to overcome this limitation is discussed in the next section.

```
edge(A,Alpha,[B|Beta],I,K), edge(B,Gamma,[],K,J) ==>
          precedes(Alpha,B),
          precedes(B,Beta)    |
          edge(A,[B|Alpha],Beta,I,J).
```

FIGURE 4. The completion step for ID/LP grammars

4. EXTENSIONS OF THE BASIC TECHNIQUE

There are two directions the extensions of the presented technique of CHR parsing might take. Firstly, one might consider parsing of more complicated grammars compared to the CF ones which were assumed so far. Following Shieber et al., one can consider unification-based grammars or tree adjoining grammars. Since I think that the previous sections showed that the Shieber et al. approach is transferable in general, the results they present are applicable to this approach as well. Instead, I want to consider parsing minimalist grammars (Chomsky; 1995) as defined in recent work by Stabler (1997, 1999).[7]

4.1. Minimalist Parsing. We cannot cover the theory behind the derivational minimalism approach presented in Stabler's papers in any detail. Very briefly, lexical items are combined with each other by a binary operation *merge* which is triggered by the availability of an appropriate pair of clashing features, here noted as `cat(C)` for Stabler's categories `c` and `comp(C)` and `spec(C)` for `=c`.[8] Furthermore, there is a unary operation *move* which, again on the availability of a pair of clashing features (e.g., `-case`, `+case`), triggers the extraction of a (possibly trivial) subtree of the tree under consideration and its merging in at the root node. On completion of these operations the clashing feature pairs are removed (checked). The lexical items are of the form of linked sequences of trees. Accessibility of features is defined via an order on the nodes in this chain of trees.[9] A parse is acceptable if all features have been checked, apart from one category feature which spans the length of the string. The actual algorithm

[7]The code for the implementation underlying the paper was kindly provided by Ed Stabler. Apart from the implementation in CHR, all the rest is his work and his ideas.

[8]The split of the *merge* rule between merging complementizers and specifiers allows for a more perspicuous presentation of the inference rules, as one can see in Tab. 5.

[9]We ignore certain aspects for simplicity. The reader is referred to the original literature for the full details.

works naively bottom-up and since the operations are at most binary, the algorithm is CYK-based. This is the nice thing about Stabler's proposal – minimalist parsing can be reduced to CYK parsing.

An initial edge or axiom in this minimalist parsing system cannot simply be assumed to only cover the part of the string where it was found since it could have been the result of a movement operation. So the elements of the lexicon which will have to be moved (they contain a phrasal movement trigger `-X`) actually have the positional indices instantiated in the last of those features appearing. All other movement triggers and the position it will be base generated are assumed to be traces and therefore empty. Their positional markers are identical variables, i.e., they span no portion of the string and one does not know their value at the moment of the construction of the axioms. They have to be instantiated during the minimalist parse. The same is true for the empty elements appearing explicitly in the lexicon.

TABLE 4. A CHR-based minimalist parser: Items, Axioms and Goals

Items	`edge(I, J, Chain, Chains)`
Axiom	`edge(I, I, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow \varepsilon\rangle$, `I` a variable
	`edge(I, I+1, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$
	and there is no `-X` in `[Chain|Chains]`
	`edge(J, J, Chain, Chains)`
	$\langle$`[Chain|Chains]` $\longrightarrow w_{i+1}\rangle$, `J` a variable
	and there is a `-X` in `[Chain|Chains]`
	and `I` and `I+1` occur in `[Chain|Chains]`
Goal	`edge(0, Length, [cat(C)], [])`

As an example consider the set of items as defined by the axioms, see Tab. 4.1. The general form of the items is such that we have the indices first, then we separate the chain of trees into the first one and the remaining ones for better access. The axioms post edges corresponding to the lexical items. As an example for the actual edges and to illustrate the discussion about the possibly variable string positions in the edges, consider the recognition of the string *believe it* with a remnant movement analysis. The lexical item for *it* takes the

TABLE 5. A CHR-based minimalist parser: Inference Rules

Absorption

```
edge(I, J, L, Ls) \ edge(P, Q, L, Ls) <=>
      unif_vars(I, J, P, Q) |
      true.
```

Merge$_{Comp}$

```
edge(I, J, [comp(X)|RestHead], Chains0),
   edge(K, L, [cat(X)|RestComp], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestComp, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Merge$_{Spec}$

```
edge(I, J, [cat(X)|RestSpec], Chains0),
   edge(K, L, [spec(X)|RestHead], Chains1) ==>
      test_vars(I, J, K, L, P, Q, R, S),
      check(RestHead, NewHead, P, S, A, B,
            RestSpec, Chains0, Chains1, Chains) |
   edge(A, B, NewHead, Chains).
```

Move

```
edge(I, J, [+(X)|RestHead], Chains0) ==>
      test_vars(I, J, P, Q),
      Link =.. [X,K,P],
      select([-Link|RestLinks], Chains0, Chains1),
      add(RestLinks, Chains1, Chains),
      linkStart(RestHead, NewHead, K, Q, L, M) |
   edge(L, M, NewHead, Chains).
```

form:

```
lex(it,I,[(K,K)=[cat(d),-case(I,J)]]) :- J is I+1.
```

where `I=1` if we instantiate it as an axiom. The resulting edge will be `edge(K, K, [cat(d),-case(1,2)]], [])`. We know that *it* has been moved to cover positions `1` to `2`, but we do not know (yet) where it was base generated.

These non-ground edges actually have an effect on the inference rules, see Tab. 5. Apart from this complication, they pretty much reflect a bottom-up approach to merging and moving lexical items. Therefore we will not discuss the details of how the conditions in the guards achieve to test for the correct configurations. The interested reader is referred to Stabler (1999).

I will briefly discuss the Prolog predicates `unif_vars/4`, `test_vars/4` and `test_vars/8`.[10] In the Shieber at al. system, we compared ground positional indices to find matching edges. Since CHR are sensitive to the instantiation of variables, it is not possible to do the same here.[11] The above mentioned predicates basically test for their arguments to be variables. If they are, new ones are returned such that the relationship (identity) between the members of pairs are preserved. If they are ground, they are unified. It is obvious that the price we pay is a decrease in efficiency since the CHR system actually has to enter the guards to determine applicability of the rules.

We cannot go into any detail how the actual parser work. Nevertheless, the implementation shown in Tab. 5 demonstrates how easily one can implement even complicated parsers without the need for extra utilities to deal with chart and agenda. Furthermore, it offers the possibility to implement feature cancellation as real constraint solving which opens up an interesting new perspective for processing.

4.2. Compiling the Grammar Rules into the Inference Rules. Another proposal for extending the approach consists in moving the test for rule applicability from the guards into the heads of the CHR rules. One can translate a given context-free grammar under a given set of inference rules into a CHR program which contains constraint propagation rules for each grammar rule, thereby making the process more efficient. For simplicity, we discuss only the simplest case of bottom-up parsing.

---

[10] Since we are dealing with the positions, the arguments always are members of a pair stemming from one edge.

[11] CHR have a compile time option which allows the disabling of this sensitivity. But doing this does not help since CHR are committed choice and the unification cannot be undone via backtracking.

For the translation from a CF grammar into a constraint framework we have to distinguish two types of rules – those with from those without an empty RHS – since empty RHSs constitute a problem for bottom-up chart parsing.[12] We treat the easier case of the conversion first. For each rule in the CF grammar with a non-empty RHS we create a constraint propagation rule such that each daughter of the rule introduces an edge constraint in the head of the propagation rule with variable, but appropriately matching string positions but a fixed label. The new, propagated edge constraint spans the entire range of the positions of the daughters and is labeled with the (nonterminal) symbol of the LHS of the CF rule. In our example, the resulting propagation rule for `s` looks as follows:

```
edge(I,K,np), edge(K,J,vp) ==> edge(I,J,s)
```

As usual for bottom-up parsing, the translation is a little bit more complicated for rules with empty RHSs. Basically, we create a propagation rule for each empty rule, e.g., $\mathtt{A} \longrightarrow \varepsilon$, such that the head is an arbitrary edge, i.e., both positions and the label are arbitrary variables, and post new edge constraints with the LHS of the CF rule as label, using the positional variables and spanning no portion of the string, resulting in CHR rules of the following type:

```
edge(I,J,_Sym) ==> edge(I,I,A), edge(J,J,A)
```

But obviously rules of this type lead to nontermination since they would propagate further constraints on their own output which is avoided by including a guard which ensures that empty edges are only propagated for every possible string position once by testing whether the edge spans a string of length one. Recall that storing and using already existing edge constraints is avoided with an absorption rule.[13] Since these empty constraints can be reused an arbitrary number of times, we get the desired effect without having to fear nontermination. Although this is not an elegant solution, it seems that other alternatives such as analyzing and transforming the entire grammar or posting the empty constraints while traversing the input string are not appealing either since they give up the one-to-one correspondence between the rules of the CF grammar and the constraint program which is advantageous in debugging.

With this technique, the parsing times achieved were better by a factor of a third compared to the Shieber et al. implementation. Although now the process of the compilation obscures the direct connection between parsing-as-deduction

[12] Top-down parsing, on the other hand, has to be careful with left recursion and care has to be taken in the parser to avoid nontermination.

[13] The proper and more elegant treatment of this problem would test in the guard of the CHR rule whether the edge constraint to be posted is already entailed by the constraint store. Unfortunately, having a constraint in the guard of a clause in CHR causes that constraint to be posted which is definitely not what is needed.

and constraint propagation somewhat, the increase in speed makes it a worthwhile exercise. There are some further ways to improve the performance of the resulting CHR parsers involving for example pragma declarations or more sophisticated ways of handling empty categories via a precompiled link relation which I cannot go into here for reasons of space.

5. Conclusion

In the paper, the similarity between parsing-as-deduction and constraint propagation is used to propose a flexible and simple system which is easy to implement and therefore offers itself as a testbed for different parsing strategies (such as top-down or bottom-up), for varying modes of processing (such as left-to-right or right-to-left) or for different types of grammars (such as for example minimalist grammars). Compared to the Shieber approach, the pure version seems to be lacking in efficiency. This can be remedied by providing an automatic compilation into more efficient specialized parsers.

More work has to be invested in the realization of a larger system combining these techniques with constraint solvers for existing constraint-based natural language theories to see whether further benefits can be gotten from using parsing as constraint propagation. To be more specific, due to the flexibility of the CHR system, one can now use the constraint propagation approach to drive other constraint solving or constraint resolution techniques (also implemented in CHR) resulting in a homogenous environment which combines both classical constraint solving with a more operational generator.

Specifically, one can use each created edge to post other constraints, for example about the well-formedness of associated typed feature structures. By posting them, they become available for other constraint handling rules. In particular, systems implementing HPSG seem to suffer from the problem how to drive the constraint resolution process efficiently. Some systems, as for example ALE (Carpenter and Penn; 1998) use a phrase structure backbone to drive the process. The proposal here would allow to use the ID/LP schemata directly as constraints, but nevertheless as the driving force behind the other constraint satisfaction techniques. However, this remains speculative.

Bibliography

Abdennadher, S. (1998). *Analyse von regelbasierten Constraintlösern*, PhD thesis, Ludwig-Maximilians-Universität München.

Abdennadher, S., Fruehwirth, T. and Meuss, H. (1996). On confluence of constraint handling rules, *Lecture Notes in Computer Science* **1118**: 1–15.

Carpenter, B. and Penn, G. (1998). ALE: The attribute logic engine, version 3.1, *User manual*, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213.

Chomsky, N. (1995). *The Minimalist Program*, Vol. 28 of *Current Studies in Linguistics*, MIT Press.

Frühwirth, T. (1998). Theory and practice of constraint handling rules, *Journal of Logic Programming* **37**(1–3): 95–138. Special Issue on Constraint Logic Programming.

Götz, T. and Meurers, D. (1997). Interleaving universal principles and relational constraints over typed feature logic, *ACL/EACL Conference '97*, Madrid, Spain, pp. 1–8.

Graham, G., Harrison, M. G. and Ruzzo, W. L. (1980). An improved context–free recognizer, *ACM Transactions on Programming Languages and Systems 2 (3)*, ACM, pp. 415–462.

Intelligent Systems Laboratory (1995). SICStus Prolog user's manual, *Technical report*, Swedish Institute of Computer Science.

Manandhar, S. (1994). An attributive logic of set descriptions and set operations, *Proceedings of the 32nd. Annual Meeting of the Association for Computational Linguistics*, ACL.

Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints*, MIT Press.

Matiasek, J. (1994). *Principle-Based Processing of Natural Language Using CLP Techniques*, PhD thesis, TU Wien.

Morawietz, F. (1995). A Unification-Based ID/LP Parsing Schema, *Proceedings of the International Workshop on Parsing Technologies*, ACL/SIGPARSE, Prag, pp. 162–173.

Penn, G. (in press). An optimized prolog encoding of typed feature structures, *Proceedings of the 16th International Conference on Logic Programming*, Las Cruces, New Mexico.

Penn, G. (pc). Implementing co-routining in a typed language, *To be published in the Arbeitspapiere des SFB 340*, Universität Tübingen.

Pereira, F. C. N. and Warren, D. H. D. (1983). Parsing as deduction, *ACL Proceedings, 21st Annual Meeting*, Vol. 13.

Saraswat, V. A. (1993). *Concurrent Constraint Programming*, MIT Press, Cambridge, Massachusetts.

Shieber, S. M., Schabes, Y. and Pereira, F. C. N. (1995). Principles and implementation of deductive parsing, *Journal of Logic Programming* **24**(1–2): 3–36.

Sikkel, K. (1997). *Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms*, ETACS Series: Texts in Theoretical Computer

Science, Springer.

Smolka, G. (1995). The Oz programming model, *in* J. van Leeuwen (ed.), *Computer Science Today*, Vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 324–343.

Stabler, E. (1997). Derivational minimalism, *in* C. Retoré (ed.), *Logical Aspects of Computational Linguistics*, Springer, Berlin, pp. 68–95. LNAI 1328.

Stabler, E. (1999). Performance models for a derivational minimalism, Presented at the Workshop *Linguistic Form and its Computation* of the SFB 340 in Bad Teinach. Draft.

3

Applying Constraint Handling Rules to HPSG

GERALD PENN

Bell Laboratories, 2D-414
Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974 USA

gpenn@sfs.nphil.uni-tuebingen.de

ABSTRACT. Constraint Handling Rules (CHR) have provided a realistic solution to an over-arching problem in many fields that deal with constraint logic programming: how to combine recursive functions or relations with constraints while avoiding non-termination problems. This paper focuses on some other benefits that CHR, specifically their implementation in SICStus Prolog, have provided to computational linguists working on grammar design tools. CHR rules are applied by means of a subsumption check and this check is made only when their variables are instantiated or bound. The former functionality is at best difficult to simulate using more primitive coroutining statements such as SICStus when/2, and the latter simply did not exist in any form before CHR.

For the sake of providing a case study in how these can be applied to grammar development, we consider the Attribute Logic Engine (ALE), a Prolog preprocessor for logic programming with typed feature structures, and its extension to a complete grammar development system for Head-driven Phrase Structure Grammar (HPSG), a popular constraint-based linguistic theory that uses typed feature structures. In this context, CHR can be used not only to extend the constraint language of feature structure descriptions to include relations in a declarative way, but also to provide support for constraints with complex antecedents and constraints on the co-occurrence of feature values that are necessary to interpret the type system of HPSG properly.

1. Introduction

Constraint Handling Rules (CHR, Frühwirth and Abdennadher; 1997) have provided a realistic solution to an over-arching problem in many fields that deal with constraint logic programming: how to combine recursive functions or relations with constraints while avoiding non-termination problems. Potential applications of constraint logic programming within computational linguistics have certainly not been immune to this problem, and for them, CHR is currently the only existing tractable alternative to manually specifying a set of delay statements to avoid non-termination in a space of constraints with huge numbers of non-deterministic constraints and relational attachments that are almost invariably recursive.

This paper focuses on some other benefits that CHR, specifically its implementation in SICStus Prolog, has provided to computational linguists working on grammar design tools. CHR rules are applied through a subsumption check and this check is made only when their variables are instantiated or bound. The former functionality is at best difficult to simulate using more primitive coroutining statements such as SICStus `when/2`, and the latter simply did not exist in any form before CHR, to the best of this author's knowledge.

For the sake of providing a case study in how these can be applied to grammar development, we will consider the Attribute Logic Engine (ALE) (Carpenter and Penn; 1996), a Prolog preprocessor for logic programming over the logic of typed feature structures (Carpenter; 1992), and its extension to a complete grammar development system for Head-driven Phrase Structure Grammar (HPSG) (Pollard and Sag; 1987, 1994), a popular constraint-based linguistic theory that uses typed feature structures. A great many details of HPSG have changed since its inception and still more are yet to be agreed upon, so the presentation here is by necessity a tremendous simplification. Nevertheless, it is hoped that this paper will stimulate a discussion of what limit CHR permits us to extend the state of the art in grammar development to, and what future developments on the part of implementors of logic programming languages such as Prolog would be most beneficial in order to push that limit even further.

The next three sections give a very brief introduction to the view of human languages that HPSG takes, how typed feature structures fit into that view, and what the basic constraint-solving problem is that grammar developers are faced with. Section 5 then shows how to compile these general constraints into more atomic ones and where CHR fills in the gap in providing an elegant implementation of this in Prolog. Section 6 illustrates another kind of constraint that must be dealt with internally by HPSG grammar development systems, and how CHR provides the unique ability to do that efficiently.

Finally, Section 7 points to a few issues that still remain to be resolved relative to what Prolog with its CLP extensions is currently able to provide, namely a more efficient implementation of attributed variables and a means for universal quantification of constraints.

2. Head-driven Phrase Structure Grammar

HPSG is a theory that seeks to predict the grammaticality, or syntactic well-formedness, of utterances in any human language. It is sometimes called a *realist* theory of language, as opposed to a *mentalist* theory of language, because grammatical utterances from human languages are taken to be actual entities that exist in the world. As such, these entities constitute a universe of objects for a model that we seek to describe using HPSG. The principal role of a grammar development system in this context is to provide the means to state theories of human language formally, and then to ask whether the formal representation of a candidate utterance with a particular phonological realization, i.e., one comprised of a particular string of words, has a non-empty denotation in this model.

By convention, the theory itself is stated using the formal language of the logic of typed feature structures. Typed feature structures are a semantically typed kind of record, which is a very convenient representation for linguistic purposes because the combination of typing with named attributes allows for a very terse description language that can easily make reference to a sparse amount of information in what are usually extremely large structures/records. There are actually several variations of that language in circulation, e.g., Smolka; 1988; King; 1989; Carpenter; 1992, which are more or less well-suited to the purposes of HPSG, or, perhaps more historically accurate, to the rules of which linguists working with HPSG more or less elected to adhere. The language that the seminal work on HPSG of Pollard and Sag (1994) actually uses was finally formalized in its own right by Richter et al. (1999); Richter (2000). The bad news is that satisfiability in this language is not even compact. The good news is that no one particularly seems to care about the bad news. This is due to the fact that: (1) most of the work performed by linguists in HPSG is not actually formalized in this language, thus creating a gap between theory and practice that must be bridged in the course of formalization in a grammar development system anyway; and (2) most grammar development systems have, for years, elected to follow a much more conservative route inspired by work in constraint logic programming, in which a core constraint language for describing typed feature structures has been extended with a relational definite clause language (Höhfeld and Smolka; 1988) for conveniently stating the "constraints," or as they are called in linguistics, *principles*, that define grammaticality in human language. The core constraint language that will be assumed here is essentially the one defined by Carpenter (1992), with an eye towards the view of feature structures in HPSG taken by King (1999).

3. Typed Feature Structures and their Descriptions

Each typed feature structure is a tuple $\langle Q, \theta, \delta \rangle$, where Q is a set of nodes (corresponding to substructures of feature structures), θ is a total typing function that maps nodes to a fixed finite meet semi-lattice of types, *Type*, and δ is a partial feature value function that maps pairs consisting of a feature, drawn from a finite set, *Feat*, and a node in Q to nodes in Q. Typed feature structures are conventionally depicted as records, such as:

$$\begin{bmatrix} \text{throwing} & \\ \text{THROWER} & \begin{bmatrix} \text{index} & \\ \text{PERSON} & \text{third} \\ \text{NUMBER} & \boxed{1}\ \text{singular} \\ \text{GENDER} & \text{masculine} \end{bmatrix} \\ \text{THROWN} & \begin{bmatrix} \text{index} & \\ \text{PERSON} & \text{third} \\ \text{NUMBER} & \boxed{1} \\ \text{GENDER} & \text{neuter} \end{bmatrix} \end{bmatrix}$$

In this example, there is a total of eight nodes, one of type *throwing*, two of type *index*, two of type *third*, and one each of type *singular*, *masculine* and *neuter*. Note that the numeric tag (called a *re-entrancy*) indicates that the two number features have extensionally the same value, which is of type *singular*, whereas the two person features have different feature structure values, i.e., different nodes, that happen to be of the same type, *third*. Typed feature logics are normally intensionally typed, so this identity of type does not imply an extensional identity of feature values. We can refer to substructures of feature structures along paths, provided that the values of each feature on the path exist. The THROWER:GENDER value of this example is of type *masculine*.

The core constraint language itself is a description logic for talking about feature structures relative to the finite meet semi-lattice of types, *Type*, a fixed finite set of features, *Feat*, and a countable set of variables, *Var*:

> The description language, Φ, is the least set of descriptions that contains:
> - $v, v \in Var$,
> - $\tau, \tau \in Type$,
> - $\text{F} : \phi, \text{F} \in Feat, \phi \in \Phi$,

- $\phi_1 \wedge \phi_2, \phi_1, \phi_2 \in \Phi$, and
- $\phi_1 \vee \phi_2, \phi_1, \phi_2 \in \Phi$.

A nice property of this language is that every non-disjunctive description with a non-empty denotation has a unique most general feature structure in its denotation. This is called its *most general satisfier*. Of course, this is not the same kind of denotation as the one that connects our theory to language itself — there are two levels of abstraction at work. Descriptions denote sets of feature structures, and feature structures denote sets of utterances in the world.

A given set of types and features is also typically augmented with a set of *appropriateness conditions* that mediate their occurrence, following King (1989) and Carpenter (1992). Appropriateness conditions stipulate, for every type, a finite set of features that can and must have values in feature structures of that type. This effectively forces feature structures to be finite-branching terms with named attributes, although the introduction of new features combined with subtyping means that the arities of these terms are not completely fixed. Appropriateness conditions also usually restrict the value that appropriate features can take by specifying a type that must be a supertype of the value's type.

In the formulation of Carpenter (1992), appropriateness must also guarantee that there is a unique most general type for which a given feature is appropriate. This is called *unique feature introduction*, and for feature F, its introducing type is called $Intro(\text{F})$. Until recently, it was generally believed that Prolog systems that work with typed feature structures must meta-interpret unification and SLD resolution over them. As shown by Penn (2000), when unique feature introduction is assumed, every statically typable set of appropriateness conditions admits a Prolog term encoding of its typed feature structures, thus reducing feature structure unification to Prolog unification in the underlying Warren abstract machine. When it is not assumed, furthermore, unique feature introduction can be added automatically in polynomial time while preserving meet-semi-latticehood. This result opens the door to using pieces of Prolog implementations' enhanced functionality, including CHR, that would have been prohibitively less elegant if even only unification needed to be meta-interpreted.

4. The Language of Principles

Principles can be divided into two classes: those that apply to all languages, and those that only apply to specific languages. The lexicon itself can be regarded as a very disjunctive principle — language-specific, of course — associating each word to a specific syntactic and semantic representation. This paper will not say much more about the lexicon.

In HPSG, and, in fact, in nearly every constraint-based theory of grammar, all principles, regardless of class, take the form of a statement, “for all terms/feature structures, statement π holds.” In contrast to most work in constraint logic programming, every constraint is thus implicitly universally quantified over typed feature structures, and thus not explicitly posted to a store. In practice, nearly every principle’s π takes the form of an implication, and for most complex principles, π makes reference to instances of relations. The approach taken here is that all π are of the form:

$$\alpha \Longrightarrow (\gamma \wedge \rho)$$

where α and γ are descriptions from the core constraint language and ρ is drawn from a definite clause language of relations, whose arguments are also descriptions from the core constraint language. Typically, these arguments include variables that are also used in γ, so that variables take scope over the entire consequent.

This form is somewhat of an idealization itself. Linguists often formulate principles in which variables are shared between α and the consequent, with the intention that $\alpha \Longrightarrow (\gamma \wedge \rho)$ should actually be $\alpha \Longrightarrow (\alpha \wedge \gamma \wedge \rho)$. Sometimes, they also use relations in α rather than just descriptions from the core language, for the interpretation of which negation by failure or some other extra convention must be used. These will be ignored in the rest of this paper, although the method described below for dealing with this form of principle can be extended in several ways to attempt to handle these more complicated antecedents as well.

The presumed form of these principles is not new. The ALE system permits the same form, but where α can only be a type description, τ. The ConTroll system (Goetz and Meurers; 1997) permits essentially the same form, but interprets the implications classically, which creates a very large search space with severe non-termination problems because of the presence of potentially recursively defined relations in ρ. ConTroll attempted to remedy this problem by providing a language of delays, similar to coroutining predicates in Prolog but with the extra implicit universal quantification built in. Repeated attempts at large-scale development with this method proved unpromising, as finding the right set of delays actually proved to be more work than simply using a more traditional deductive strategy such as SLD resolution or definite clause grammars. Such delays are also non-modular relative to changes in other parts of the grammar, which is of particular concern since grammar development is largely characterized by a large number of small incremental changes.

By contrast, the approach taken here is to allow for arbitrary α but to interpret the implications using subsumption by α, i.e., for every feature structure (the implicit universal quantification is still there), either the consequent holds, or the feature structure is not subsumed by the most general satisfier of α. This decision draws its inspiration from three sources. First, ALE used the same interpretation, although somewhat more controversially given that α could only be a type. Second, it is widely believed in the linguistics community (Hinrichs and Nakazawa; 1996) that *lexical rules*, closure conditions on the well-formedness of a lexicon, are intended to apply by subsumption. They also take the form of implications, as "if λ_1 is part of the lexicon, then so is λ_2," in which λ_1 and λ_2 are specified using descriptions with or without some extra level of default reasoning. One can, in fact, also find many instances of principles in theoretical linguistics literature in which a subsumption-based interpretation is the intended one. The third source is the approach to CHR in its SICStus Prolog implementation, in which the head of a rule is matched using a subsumption check. The fact that linguistic theory, conventional grammar development practice and the CHR package agree on this point is very significant, and is exploited by our implementation.

The subsumption-based approach is sound with respect to the classical interpretation of implication for those cases of principles when the classical interpretation really is the correct one. For completeness, other resolution methods must be used. These are available from true constraint logic programming, which as of yet is surprisingly absent from linguistic research on constraint-based grammar. CHR can provide those additional resolution methods as well, of course.

5. Principle Decomposition

Under these assumptions, every linguistic principle can be decomposed into a series of more atomic constraints that can easily be implemented. It should first be noted that the existence of a CHR package for Prolog terms easily admits a CHR package for typed feature structures, where feature descriptions in the arguments of constraints are translated to the term encodings of their most general satisfiers. The result can then be compiled by the underlying Prolog CHR library's compiler. This immediately provides grammar developers with an alternative to providing explicit delay statements that has proven to be far more robust and tractable for large-scale grammar development. The rest of this section concerns the compilation of the universally quantified principles themselves.

5.1. Finding a Trigger. Let $\mathit{trigger}(\alpha)$ be defined such that:

- $\mathit{trigger}(v) = \bot$,
- $\mathit{trigger}(\tau) = \tau$,

- $trigger(\text{F} : \phi) = Intro(\text{F})$,
- $trigger(\phi_1 \wedge \phi_2) = trigger(\phi_1) \sqcup trigger(\phi_2)$, and
- $trigger(\phi_1 \vee \phi_2) = trigger(\phi_1) \sqcap trigger(\phi_2)$,

where $\perp$ is the most general type in *Type*, and $\sqcup$ and $\sqcap$ are join (unification) and meet (generalization) in *Type*. $\perp$ and meet exist given the assumption that *Type* is a meet semi-lattice. From that same assumption, join exists between any pair of consistent types.

If we had a predicate, `fswhen(V=Desc,Goal)`, available such that we could delay `Goal` until `V` was subsumed by the most general satisfier of `Desc`, then all principles:

$$\alpha \Longrightarrow (\gamma \wedge \rho)$$

could be converted into:

$$trigger(\alpha) \Longrightarrow v \wedge \mathtt{fswhen}((v = \alpha), ((v = \gamma) \wedge \rho))$$

This form (with the exception of `fswhen/2`) is the form already implemented by ALE, including its implicit universal quantification. This typically involves compiling unifications of feature structures into unifications of their term encodings plus a type check to see whether one or more principle consequents need to be applied. The conditions under which that check must be made can, of course, be statically optimized to a bare minimum, but in the worst case, it significantly compromises the speed of what would otherwise be a simple Prolog-level unification. This point is discussed further in Section 7.

As a running example, we can consider a principle from the HPSG grammar for German under development at the University of Tübingen:

```
synsem:loc:cat:(head:verb, marking:fin)
  ==> synsem:loc:cat:head:vform:bse.
```

This is called the Finiteness Marking Principle. The description to the left of the implication selects feature structures whose substructure on the path SYNSEM: LOC: CAT satisfy two requirements: that their HEAD value is of the type *verb* and that their MARKING value is of the type *fin*. The principle says that every feature structure that satisfies that description must also have a SYNSEM: LOC: CAT: HEAD: VFORM value of type *bse*. The problem is that feature structures may not have substructures that are specific enough to determine whether this constraint holds or not. This can happen when a type subsumes several more specific subtypes of which only one meets the requirements set forth by the constraint, for example. So we must wait until it is known whether the antecedent is true or false before applying the consequent. If we reach a deadlock, where several constraints are suspended on their antecedents, then we must use another resolution method to begin testing more specific extensions of the feature structure in turn.

To find the trigger in this example, we can observe that the antecedent is a feature value description (F:ϕ), so the trigger is *Intro*(SYNSEM), the unique introducer of the SYNSEM feature, which happens to be the type *sign*. We can then transform this constraint to:

```
sign cons X
     goal fswhen((X=synsem:loc:cat:(head:verb, marking:fin)),
                 (X=synsem:loc:cat:head:vform:bse)).
```

The `cons` and `goal` operators are defined by ALE, and used to separate the type antecedent from the description component of the consequent (in this case, just the variable, `X`), and the description component of the consequent from its relational attachment. We know that any feature structure subsumed by the original antecedent will also be subsumed by the most general feature structure of type *sign*, because *sign* introduces SYNSEM.

5.2. Reducing Complex Conditionals. The next step is to decompose `fswhen/2` itself. For simplicity, it can be assumed that its first argument is actually drawn from a more general conditional language, including those of the form $V = Desc$ and closure under conjunction and disjunction. Also for simplicity, it is assumed that the variables of each $Desc_i$ are distinct.

Such a complex conditional can easily be converted into a normal form in which for each atomic conditional, $V_i = Desc_i$, $Desc_i$ is non-disjunctive. Just as with the principles themselves, we can begin by isolating types and assuming that we have a predicate `typewhen(Type,V,Goal)` that delays `Goal` until `V` is of type `Type`. We can then convert all of the other conditionals to this by converting `fswhen(C,Goal)` to `Conv`, i.e., `reduce(C,Goal,Conv)`, where `reduce/3` is defined such that:

```
reduce((VD1,VD2),Goal,Conv) :- % fswhen((VD1,VD2),Goal) iff
  reduce(VD2,Goal,Goal2),      %  fswhen(VD1,fswhen(VD2,Goal))
  reduce(VD1,Goal2,Conv).

reduce((VD1;VD2),Goal,(Conv1,Conv2)) :-
  reduce(VD1,(prolog(Trigger = 0) -> Goal ; true),Conv1),
  reduce(VD2,(prolog(Trigger = 1) -> Goal ; true),Conv2).

reduce(V=Desc,Goal,Conv) :-
  vars_of(Desc,Vars),
  desc_reduce(Desc,V,Goal,Vars,_,Conv).
```

and `desc_reduce/6` is defined such that `desc_reduce(Desc, V, Goal, Vars, VarsRest, Conv)` holds if and only if `fswhen(V=Desc, Goal)` reduces to `Conv` and `Vars-VarsRest` is the collection of all `Vars` that occur in `Desc`:

```
desc_reduce(X,V,Goal,Vars,VarsRest,Conv) :-
  var(X),
  !,( select(Vars,X,VarsRest) -> Conv = (V = X,call(Goal))
      % if first occurrence of X, bind to V

    ; Conv = when(?=(V,X),((V==X) -> call(Goal) ; true)),
      VarsRest = Vars
      % otherwise, wait until V==X
    ).

desc_reduce(F:Desc,V,Goal,Vars,VarsRest,
            typewhen(IntroType,V,(farg(F,V,FVal),
                                  call(DescConv)))) :-
  intro(F,IntroType),
  desc_reduce(Desc,FVal,Goal,Vars,VarsRest,DescConv).

desc_reduce(T,V,Goal,Vars,Vars,Conv) :-
  type(T),
  !, Conv = typewhen(T,V,Goal).

desc_reduce((Desc1,Desc2),V,Goal,Vars,VarsRest,Conv2) :-
  desc_reduce(Desc1,V,Goal,Vars,VarsMid,Conv1),
  desc_reduce(Desc2,V,Conv1,VarsMid,VarsRest,Conv2).
```

Here, the `prolog/1` goals are inserted without modification by a compiler, as goals in that position are normally interpreted to be relations over feature structure descriptions. The Prolog convention of comma for AND and semi-colon for OR is also used in these definitions. In the disjunctive case of `reduce/3`, the binding of the variable `Trigger` is necessary to ensure that Goal is only resolved once in case the goals for both conditionals eventually unsuspend. In the variable case, `desc_reduce/6` simply binds the feature structure in question when it first encounters a variable, but subsequent occurrences of that variable create a suspension that compiles directly down to Prolog `when/2`, checking for identity with the previous occurrences. Notice also that *Intro*(`F`), here called `IntroType`, can be used as the type to delay on in the case of a feature value description. `farg(F,V,FVal)` binds `FVal` to the argument position of `V` that corresponds to the feature `F` in the term encoding of that feature structure once `V` has been instantiated to a type for which `F` is appropriate.

In the running example, we must convert the large `fswhen/2` relational attachment to simpler `typewhen/2` goals. The second clause of the relation, `desc_reduce/6` tells us that this can be achieved by successively waiting for the types that introduce each of the features, SYNSEM, LOC, CAT. Those types are *sign*, *syntax_semantics* and *local*, respectively:

```
sign cons X
     goal typewhen(sign,X,(farg(synsem,X,SynVal),
          typewhen(syntax_semantics,SynVal,
                   (farg(loc,SynVal,LocVal),
          typewhen(local,LocVal,(farg(cat,LocVal,CatVal),
          fswhen((CatVal=(head:verb, marking:fin)),
                 (X=synsem:loc:cat:head:vform:bse)))))))).
```

In practice, a great deal of static analysis is possible to reduce the complexity of the resulting relational goals. In this case, static analysis of the type system tells us that since *sign* is the trigger type, *syntax_semantics* is the least appropriate type of a SYNSEM value, and *local* is the least appropriate type of a LOC value, all three of these `typewhen/2` calls can be eliminated:

```
sign cons X goal (farg(synsem,X,SynVal),
                  farg(loc,SynVal,LocVal),
                  farg(cat,LocVal,CatVal),
                  fswhen((CatVal=(head:verb, marking:fin)),
                         (X=synsem:loc:cat:head:vform:bse))).
```

The description that `CatVal` is suspended on is a conjunctive description, so the last clause of `desc_reduce/6` tells us that we should successively suspend on each conjunct. The type that introduces both HEAD and MARKING is *category*.

```
sign cons X
     goal (farg(synsem,X,SynVal),
           farg(loc,SynVal,LocVal),
           farg(cat,LocVal,CatVal),
           typewhen(category,CatVal,(farg(head,CatVal,HdVal),
           typewhen(verb,HdVal,(
           typewhen(category,CatVal,
                    (farg(marking,CatVal,MkVal),
           typewhen(fin,MkVal,
                    (X=synsem:loc:cat:head:vform:bse))
           ))))))
          ).
```

Since *category* is the least appropriate type of a CAT value, static analysis again allows us to simplify the code:

```
sign cons X
     goal (farg(synsem,X,SynVal),
           farg(loc,SynVal,LocVal),
           farg(cat,LocVal,CatVal),
           farg(head,CatVal,HdVal),
           typewhen(verb,HdVal,
             (farg(marking,CatVal,MkVal),
              typewhen(fin,MkVal,
                       (X=synsem:loc:cat:head:vform:bse))
             )
           )
          ).
```

5.3. Compiling Delays on Types. Delaying until a feature structure becomes at least as specific as a certain type is not so easy because of appropriateness conditions. The term encoding of a feature structure may actually reflect the type in the conditional before all of its values have been combined to create a well-formed feature structure due to the check that must be performed to enforce constraints. CHR allows one very simply to suspend until subsumption by an entire term is achieved. We define a CHR constraint `typewhen/3`, such that for each `Type`, we add a rule:

```
typewhen(Type,MGSat,Goal) <=> call(Goal).
```

where `MGSat` is the most general satisfier of `Type`. This constraint, when posted by the calls in `desc_reduce/6` above, will wait until the variable in question is bound to something at least as specific as `MGSat`, which is a term encoding for which all appropriateness conditions are satisfied. `Type` sits in the first argument position in order to index the constraint into the correct rule more quickly.

In the running example, the term encoding of the most general satisfier of *sign* is:

```
'$sign'(_,_,'$mod_synsem'(syntax_semantics,_,_,_),_).
```

The first three argument positions correspond to the three features that are appropriate to *sign*, the first two of which have not been mentioned here and, again by static analysis, do not need to be instantiated. The third, however, is the SYNSEM value, and the subterm in that position is the most general satisfier of the type, *syntax_semantics*. The CHR compiler will create term subsumption code that causes the rule handler to wait for this feature value to be instantiated to at least this subterm, and thus satisfy the appropriateness conditions of *sign*.

The last argument position is always an anonymous variable and preserves the intensionality of the logic. The `typewhen/2` clause for *syntax_semantics* is thus:

```
typewhen(sign,
 '$sign'(_,_,'$mod_synsem'(syntax_semantics,_,_,_),_),Goal)
 <=> call(Goal).
```

This, of course, only uses a small part of CHR's functionality, but it has proven to be a vital part for combining constraint resolution, coroutining, appropriateness and logic programming in the logic of typed feature structures. It is actually possible to use `when/2` directly to achieve the same effect, but in principle, CHR should be more efficient, because it could identify at the abstract machine level when subsumption fails, thus eliminating the constraint from the store, as soon as possible. Finding the right order in which to check the various components of a term encoding directly with `when/2` is not at all trivial. This point is discussed further in Section 7.

6. Subtype Covering Constraints

Another use of CHR as a very sophisticated suspension tool arises with the enforcement of a different kind of internal constraint within feature-structure-based grammar development systems.

The logic of typed feature structures is a logic with subtyping, and this ordering among types induces an ordering among feature structures, called *subsumption*. Our view of language, however, is one of utterances that are maximally informative, and therefore discretely ordered. As a result, what we actually care about determining in the context of grammar development is whether there is a *maximally informative* feature structure for a given utterance with a non-empty denotation in our realist model of language. Maximal feature structures are those in which every node is assigned a maximally specific type, and every pair of nodes with identical types is either re-entrant or explicitly inequated. [1] Non-maximally-specific types can simply be interpreted as a shorthand for sets of their maximally specific subtypes, and non-maximal feature structures can thus be viewed as a very compact kind of representation for disjunctions of maximal feature structures.

Because of appropriateness and re-entrancies, not every feature structure has a maximal extension. Suppose type *a* has two appropriate features, F and G, both of whose values are restricted to the type *polarity*, which has maximal

[1] An inequation is a negative constraint that prohibits re-entrancies, much as in Prolog II. Their implementation is not discussed in detail here, but they can essentially be reduced to an underlying Prolog's `dif/2` statement using the term encoding mentioned above.

subtypes $+$ and $-$. *a* could have exactly two subtypes, *b* and *c*, with appropriateness conditions such that *b* must have an F value of $+$ and a G value of $-$, and *c* must have an F value of $-$ and a G value of $+$. In this case, the non-maximal feature structure:

$$\begin{bmatrix} \text{a} & \\ \text{F} & \boxed{1}\ \text{bool} \\ \text{G} & \boxed{1} \end{bmatrix}$$

has no maximal extension because $+$ and $-$ are inconsistent. Situations such as these do occasionally arise in the realm of linguistic knowledge representation, and a grammar development system based on feature structures must detect them early in order to fail as soon as possible.

These can be implemented in ALE by adding relational attachments to ALE's type-antecedent universal constraints that will suspend a goal on candidate feature structures with types such as *a* that could have this problem (called *deranged types*). The suspended goal unblocks whenever the deranged type or the type of one of its appropriate features' values is updated to a more specific subtype, and checks the types of the appropriate features' values. CHR's ability not only to suspend, but to suspend until a particular variable is instantiated or even bound to another variable is the powerful kind of mechanism required to check these constraints efficiently, i.e., only when necessary. Re-entrancies in a term encoding may only show up as the binding of two uninstantiated variables, and re-entrancies are an important case where these constraints need to be checked.

We can create a constraint, `subtype_cover/2`, whose first argument is a deranged type. For each such type, we can add three kinds of rules. The first retrieves the current type of a feature structure from its term encoding and compares it to a deranged type that it used to have.

```
subtype_cover(Type,FS) <=>
   true & (type_index(FS,FSType),FSType \== Type)
 | true.
```

In the case of our example with *a*, *b* and *c*, *a* is deranged, so the following constraint handling rule is generated:

```
subtype_cover(a,FS) <=>
   true & (type_index(FS,FSType),FSType \== a)
 | true.
```

If a feature structure of type *a* is refined to be a well-formed feature structure of type *b*, then it is no longer in danger (unification over the term encoding itself fails on appropriateness failures, i.e., non-empty denotations, in such a case). In fact, no maximally specific type can be deranged. If the type has changed, then if it is still deranged, it is the responsibility of the rules for the new deranged type to check it.

The second kind of rule compares a feature structure's appropriate feature values to a product of types that belong to one of the deranged type's "safe" subtypes (such as its maximally specific subtypes). If this product subsumes the types of the feature structure's current values, then it is out of danger. So for each such product, there is one rule of the form:

```
subtype_cover(Type,ApprogFeatProduct) <=> true.
```

In the case of *a*, the following rules are generated:

```
subtype_cover(a,'$a'('$polarity'(+,_),'$polarity'(-,_),_))
 <=> true.
subtype_cover(a,'$a'('$polarity'(-,_),'$polarity'(+,_),_))
 <=> true.
```

The first rule contains a term encoding of the feature structure:

$$\begin{bmatrix} \mathrm{a} & \\ \mathrm{F} & + \\ \mathrm{G} & - \end{bmatrix}$$

Its feature values' types correspond to the appropriateness conditions for *b*. The second rule's feature values' types correspond to the appropriateness conditions for *c*. In either case, if a feature structure of type *a* has these, then it is guaranteed to have a maximally specific extension and thus denote in the language we are modeling.

Finally, it could be that the feature structure already has no maximal extensions, or only one maximal extension. In the former case, we should fail, and in the latter case, we should extend the feature structure since that is the only possible extension it can have. The following rule uses a guard to count how many extensions are still consistent with the current feature structure: 0, 1 or more than 1. If there is more than 1, it fails, resuspending the constraint — we may still need to check later because it could lose consistency with one of those extensions. If it has 0 or 1, the guard succeeds and the body of the rule fails or extends the feature structure through unification as appropriate.

```
subtype_cover(Type,FS)
               <=> true & stc_unify_test(Type,FS,N)
                   | ( N == 0 -> fail
                     ; % N == 1
                       stc_unify_product(Type,FS)).
```

7. Further Desiderata

There are two significant shortcomings to the view of logic programming with typed feature structures presented here. The first is that feature structure unification, by far the most common operation performed by a parser or generator based on this logic, cannot always be reduced to Prolog unification. The reason, as mentioned earlier, is that certain checks must be made in order to detect whether to apply the consequent of a "universal constraint," a constraint that applies to literally every typed feature structure, although restrictions such as limiting the antecedent to a type and applying constraint consequences under subsumption ameliorate the cost of that task. The view of constraints in both `when/2` and CHR is that a constraint is posted relative to one or more terms and only then is its satisfaction tracked by the underlying abstract machine. What grammar developers (more specifically, people who write grammar development systems) need right now is the ability to state that a constraint universally applies, and apply that constraint to terms / feature structures efficiently at the abstract machine level. This is the only level at which it will not be necessary to disrupt term unification as ALE currently must.

The second shortcoming is the efficiency of the current CHR implementation in SICStus Prolog. This relies crucially on an attributed variables package that is rather inefficient. That inefficiency is what in fact prevents grammar developers working with feature structures from reducing those feature structures directly to attributed variables, which are at least cosmetically quite similar. This is in part due to the fact that there is no notion of appropriateness nor a strong enough notion of typing to support appropriateness, and thus encode typed feature structures as efficiently as a term encoding currently can. Simply looking at its application within CHR, however, even there, the number of allowable constraints in a handler does not scale up well, and even with a small number of constraints, delaying works far too slowly. As mentioned earlier, one can implement `typewhen/2` directly in terms of `when/2` at the cost of possibly having an overwhelming number of residual suspensions due to an unfortunate ordering of atomic suspensions that collectively implement delaying on subsumption by an entire term. Even with this large number of residues, our experiments with a large HPSG grammar for German show that a direct implementation in terms of `when/2` in SICStus Prolog is faster than using SICStus's CHR library by a factor of 60 or more on our test suites. In principle, CHR should be much faster. Currently, the SICStus CHR compiler simply partially evaluates the necessary run-time subsumption checks, where possible, at the Prolog level.

8. Conclusion

The compilation of two kinds of constraints was proposed relative to a preprocessor for a logic programming language that term-encodes typed feature structures for a very transparent treatment of unification and constraint logic programming with feature structures within Prolog. CHR has essentially retired the problem of how best to handle the delaying of constraints within grammar development systems, although there remain a few bottlenecks to efficiency such as the universal quantification implicit to linguistic constraints and in the SICStus Prolog implementation, the underlying dependence of the CHR library on the current implementation of attributed variables.

The extension of ALE described here, called TRALE, was developed at the the University of Tübingen in parallel to a grammar of an extensive fragment of German, both of which will be made publicly available at the end of the year 2000 as deliverables of the Sonderforschungsbereich 340.

Bibliography

Carpenter, B. (1992). *The Logic of Typed Feature Structures*, Cambridge.

Carpenter, B. and Penn, G. (1996). Compiling typed attribute-value logic grammars, *in* H. Bunt and M. Tomita (eds), *Recent Advances in Parsing Technologies*, Kluwer, pp. 145–168.

Frühwirth, T. and Abdennadher, S. (1997). *Constraint-Programmierung*, Springer Verlag.

Goetz, T. and Meurers, W. (1997). Interleaving universal principles and relational constraints over typed feature logic, *Proceedings of the 35th ACL / 8th EACL*, pp. 1–8.

Hinrichs, E. and Nakazawa, T. (1996). Applying lexical rules under subsumption, *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pp. 543–549.

Höhfeld, M. and Smolka, G. (1988). Definite relations over constraint languages, *LILOG Report 53*, IBM Deutschland.

King, P. (1989). *A Logical Formalism for Head-driven Phrase Structure Grammar*, PhD thesis, University of Manchester.

King, P. (1999). Towards truth in head-driven phrase structure grammar, *in* V. Kordoni (ed.), *Tübingen Studies in Head-Driven Phrase Structure Grammar*, Arbeitspapier des Sonderforschungsbereichs 340 (SFB 340), Nr. 132.

Penn, G. (2000). *The Algebraic Structure of Attributed Type Signatures*, PhD thesis, Carnegie Mellon University.

Pollard, C. and Sag, I. (1987). *Information-based Syntax and Semantics*, number 13 in *CSLI Lecture Notes*, CSLI Publications.

Pollard, C. and Sag, I. (1994). *Head-driven Phrase Structure Grammar*, Chicago.

Richter, F. (2000). *A Mathematical Formalism for Linguistic Theories with an Application in Head-driven Phrase Structure Grammar and a Fragment of German*, PhD thesis, Universität Tübingen.

Richter, F., Sailer, M. and Penn, G. (1999). A formal interpretation of relations and quantification in hpsg, *in* G. Bouma, E. Hinrichs, G.-J. M. Kruijff and R. Oehrle (eds), *Constraints and Resources in Natural Language Syntax and Semantics*, CSLI Publications, pp. 281–298.

Smolka, G. (1988). A feature logic with subsorts, *Technical Report LILOG Report 33*, IBM Germany, Stuttgart.

Seminar für Sprachwissenschaft, SFB 340, Projekt A8, Universität Tübingen, Wilhelmstrasse 113, D-72074 Tübingen, Germany
E-mail address: frank@sfs.nphil.uni-tuebingen.de