A Uniform Architecture for Parsing, Generation and Transfer

Rémi Zajac
Project POLYGLOSS*
IMS-CL/IfI-AIS, University of Stuttgart
Keplerstraße 17, D-7000 Stuttgart 1
zajac@informatik.uni-stuttgart.de

Abstract

We present a uniform computational architecture for developing reversible grammars for parsing and generation, and for bidirectional transfer in MT. We sketch the principles of a general reversible architecture and show how they are realized in the rewriting system for typed feature structures developed at the University of Stuttgart. The reversibility of parsing and generation, and the bidirectionality of transfer rules fall out of general properties of the uniform architecture.

1 PRINCIPLES FOR A UNIFORM ARCHITECTURE

The principles for a uniform architecture for parsing/generation and bidirectional transfer are already contained in some PROLOG implementations of logic grammars like DCGs. For example, [Shieber 88] proposes to apply the idea of Earley deduction [Pereira/Warren 83] to generation. With the noticeable exception of [Dymetman et al. 90], all of these approaches use a context-free based mapping to relate a string of words with a semantic structure. Almost all of these approaches also rely on some specific properties of the grammars intended to be processed (semantic heads, guides, leading features, specific representation of subcategorization, etc.). They are also dependent on the direction in which they are used: even if the grammar specification is the same, two different compilers generate two different programs for parsing and generation. Using the PROLOG deduction mechanism to have a simple and direct implementation of a parser/generator, one has to solve some problems due to the PROLOG evaluation method, for example termination on uninstantiated goals: goals have to be evaluated in a different order for parsing and generation. A reordering of goals performed by a rule compiler can be based on a direct specification of the ordering by the grammar writer [Dymetman/Isabelle 88], or can be derived

*Research reported in this paper is partly supported by the German Ministry of Research and Technology (BMFT, Bundesminister für Forschung und Technologie), under grant No. 08 B3116 3. The views and conclusions contained herein are those of the author and should not be interpreted as representing official policies.

by a compiler by analysing the dataflow using only input/output specifications [Strzalkowski 90].

But if we regard the grammar as a set of constraints to be satisfied, parsing and generation differ only in the nature of the "input", and there is no reason to use two different programs. An interesting approach which uses only one program is described in [Dymetman/Isabelle 88]. Within this approach, a lazy evaluation mechanism, based on the specification of input/output arguments, is implemented, and the evaluation is completly data-driven: the same program parses or generates depending only on the form of the input term. Furthermore, a reversible grammar need not to be based only on constituency. [Dymetman et al. 90] describes a class of reversible grammars ("Lexical Grammars") based on a few composition rules which are very reminiscent of categorial grammars. Other kinds of approaches can also be envisaged, e.g. using a dependency structure and linear precedence relations [Reape 90] (see also [Pollard/Sag 87]).

From these experiments, we can outline desirable properties of a computational framework for implementing reversible grammars:

- A unique general deductive mechanism is used. Grammars define constraints on the set of acceptable structures, and there is no distinction between "input" and "output".
- To abolish the input/output distinction, the same kind of data structure is used to encode both the string and the linguistic structure, and they are embedded into one data structure that represents the relation between the string and the associated linguistic structure (c.f. the HPSG sign [Pollard/Sag 87]).
- Specific mapping properties, based on constituency, linear precedence or functional composition, are not part of the formalism itself but are encoded explicitly using the formalism.
- The deductive mechanism should be computationally well-behaved, especially with respect to completeness.

In the next section, we show how these properties are realized in the Typed Feature Structure rewriting system implemented at the University of Stuttgart[1]. We then discuss the parsing and generation problem, and bidirectionality of transfer in MT. Assuming that we have the proper machinery, problems in parsing or generation can arise only because of a deficiency in the grammar[2]: in the last section, the termination problem and efficiency issues are addressed.

2 A REWRITE MACHINE FOR TYPED FEATURE STRUCTURES

The basic motivation behind the Typed Feature Structure rewriting system is to provide a language which has the same deductive and logical properties of logic programming languages such as PROLOG, but which is based on feature terms instead of first order terms [Aït-Kaci 84, Aït-Kaci 86, Emele/Zajac 90a]. Such a language has a different semantics than the Herbrand semantics: this semantics is based on the notion of approximation, which captures in a computational

[1]The TFS system has been implemented by Martin Emele and the author as part of the POLYGLOSS project.
[2]As it is often the case in generation when using a grammar built initially for parsing.
[3]See also [Emele/Zajac 90a] for a fixed-point semantics.

framework the idea that feature structures represent partial information [Zajac 90b][3]. Of course, as in PROLOG, problems of completeness and efficiency have to be addressed.

The universe of feature terms is structured in an inheritance hierarchy which defines a partial ordering on kinds of available information. The backbone of the hierarchy is defined by a partial order $\leq$ on a set of *type symbols* $\mathcal{T}$. To this set, we add two more symbols: $\top$ which represents completly underspecified information, and $\bot$ which represents inconsistent information. Two type symbols have a common most general subtype (Greatest Lower Bound – GLB): this subtype inherits all information associated with all its super-types. We define a *meet* operation on two type symbols A and B as $A \wedge B = glb(A, B)$. Formally, a type hierarchy defined as a tuple $\langle \mathcal{T}, \leq, \wedge \rangle$ is a meet semi-lattice. A technicality arises when two types A and B have more than one GLB: in that case, the set of GLBs is interpreted as a disjunction.

As different sets of attribute-value pairs make sense for different kind of objects, we divide our feature terms into different types. Terms are closed in the sense that each type defines a specific association of features (and restrictions on their possible values) which are appropriate for it, expressed as a feature structure (the definition of the type). Since types are organized in an inheritance hierarchy, a type inherits all the features and value restrictions from all its super-types. This type-discipline for feature structures enforces the following two constraints: a term cannot have a feature which is not appropriate for its type[4] and conversely, a pair of feature and value should always be defined for some type. Thus a feature term is always typed and it is not possible to introduce an arbitrary feature in a term (by unification): all features added to some term should be appropriate for its type. We use the attribute-value matrix (AVM) notation for feature terms and we write the type symbol for each feature term in front of the opening square bracket of the AVM. A type symbol which does not have any feature defined for it is atomic. All others types are complex.

A type definition has the following form: the type symbol to be defined appears on the left-hand side of the equation. The right-hand side is an expression of conjunctions and disjunctions of typed feature terms (Figure 1). Conjunctions are interpreted as meets on typed feature terms (implemented using a typed unification algorithm [Emele 91]). The definition may have conditional constraints expressed as a logical conjunction of feature terms and introduced by ':-'. The right-hand side feature term may contain the left-hand side type symbol in a subterm (or in the condition), thus defining a recursive type equation which gives the system the expressive power needed to describe complex linguistic structures.

A subtype inherits all constraints of its super-types monotonically: the constraints expressed as an expression of feature terms are conjoined using unification; the conditions are conjoined using the logical *and* operation.

[4] Checked at compile time.

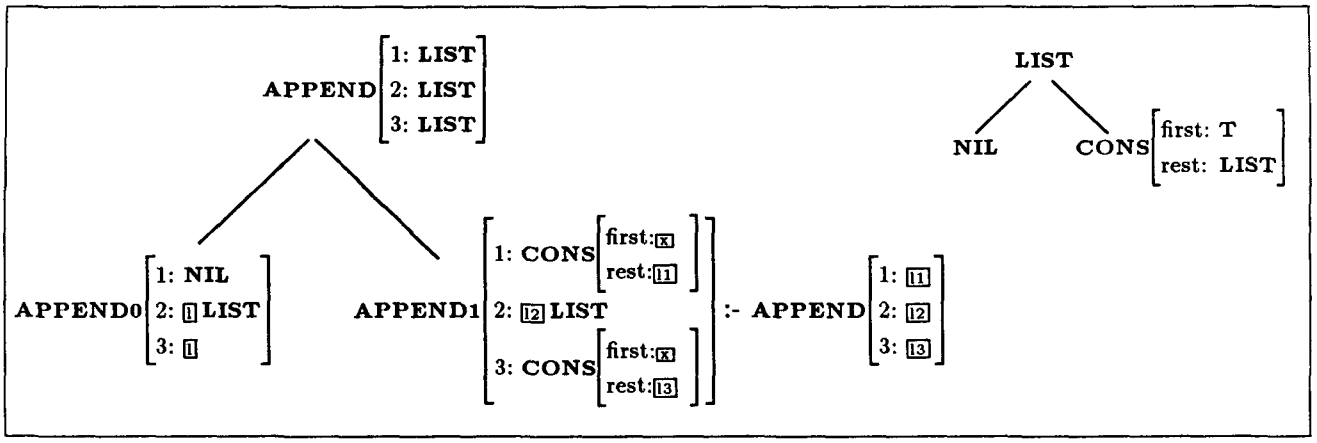

Figure 2: Type hierarchy for LIST and APPEND ($\top$ and $\bot$ omitted).

A set of type definitions defines an inheritance hierarchy of feature terms which specifies the available approximations. Such a hierarchy is compiled into a rewriting system as follows: each direct link between a type A and a subtype B generates a rewrite rule of the form $A[a] \rightarrow B[b]$ where $[a]$ and $[b]$ are the definitions of A and B, respectively.

The interpreter is given a "query" (a feature term) to evaluate: this input term is already an approximation of the final solution, though a very rough approximation. The idea is to incrementally add more information to that term using the rewrite rules in order to get step by step closer to the solution: we stop when we have the best possible approximation.

A rewrite step for a term t is defined as follows: if u is a subterm of t of type A and there exists a rewrite rule $A[a] \rightarrow B[b]$ such that $A[a] \sqcap u \neq \bot$, the right-hand side $B[b]$ is unified with the subterm u, giving a new term t' which is more specific than t. This rewrite step is applied nondeterministically everywhere in the term until no further rule is applicable[5]. Actually, the rewriting process stops either when all types are minimal types or when all subterms in a term correspond exactly to some approximation defined by a type in the hierarchy. A term is "solved" when any subterm is either more specific than the definition of a minimal type, or does not give more information than the definition of its type.

This defines an *if and only if* condition for a term to be a solved-form, where any addition of information will not bring anything new and is implemented using a *lazy rewriting* strategy: the application of a rule $A[a] \rightarrow B[b]$ at a subterm u is actually triggered only when $A[a] \sqcap u \sqsubset A[a]$. This lazy rewriting strategy implements a fully data-driven computation scheme and avoids useless branches of computation. Thus, there is no

[5]Conditions do not change this general scheme and are omitted from the presentation for the sake of simplicity. See for example [Dershowitz/Plaisted 88], and [Klop 90] for a survey.

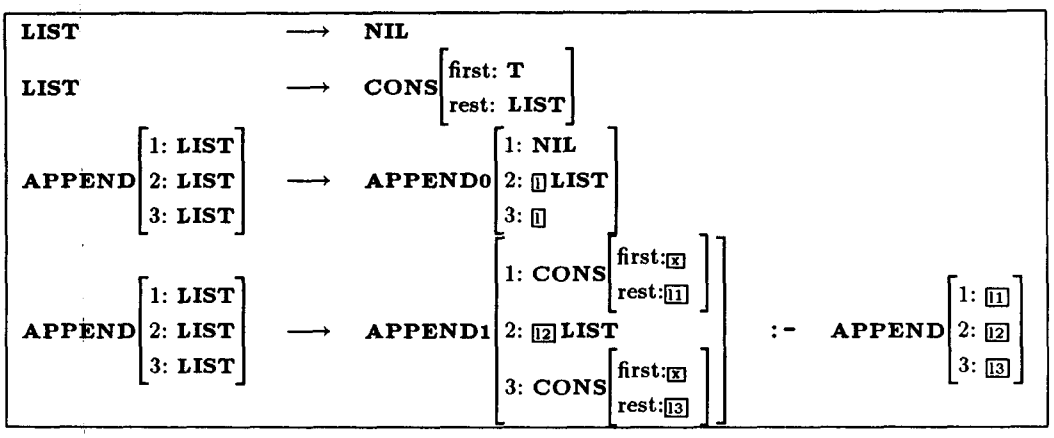

Figure 3: Rewrite rules for LIST and APPEND.

need to have a special treatment to avoid what corresponds to the evaluation of un-instantiated goals in PROLOG, since a general treatment based on the semantics of the formalism itself is built in the evaluation strategy of the interpreter.

The choice of which subterm to rewrite is only partly driven by the availability of information (using the lazy rewriting scheme). When there are several subterms that could be rewritten, the computation rule is to choose the outer-most ones (inner-most strategies are usually non-terminating)[6]. Such an outer-most rewriting strategy has interesting termination properties, since there are problems where a TFS program will terminate when the corresponding PROLOG program will not[7].

For a given subterm, the choice of which rule to apply is done non-deterministically, and the search space is explored depth-first using a backtracking scheme. This strategy is not complete, though in association with the outer-most rule and with the lazy evaluation scheme, it seems to terminate on any "well-defined" problem, i.e. when terms introduced by recursive definitions during execution are strictly decreasing according to some mesure (for example, see the definition of guides in [Dymetman et al. 90] for the parsing and generation problems). A complete breadth-first search strategy is planned for debugging purposes.

The interpreter described above is implemented[8] and has been used to test several models such as LFG, HPSG, or DCG on toy examples [Emele/Zajac 90b, Emele et al. 90, Zajac 90a].

[6]This outer-most rewriting strategy is similar to hyper-resolution in logic programming. The lazy evaluation mechanism is related to the 'freeze' predicate of, e.g. Prolog-II and Sicstus Prolog, though in Prolog, it has to be called explicitly.

[7]e.g. the problem of left-recursive rules in naive PROLOG implementations of DCGs

[8]A prototype version is publically available.

3 PARSING, GENERATION, AND BIDIRECTIONAL TRANSFER

3.1 Parsing/generation

A grammar describes the relation between strings of words and linguistic structures. In order to implement a reversible grammar, we have to encode both kinds of structure using the same kind of data structure provided by the TFS language: typed feature structures. A linguistic structure will be encoded using features and values, and the set of valid linguistic structures has to be declared explicitly. A string of words will be encoded as a list of word forms, using the same kind of definitions as in Figure 1.

To abolish the distinction between "input" and "output", the relation between a string and a linguistic structure will be encoded in a single term with, for example, two features, **string** and **syn** and we can call the type of such a structure SIGN[9]. The type SIGN is divided into several subtypes corresponding to different mappings between a string and a linguistic structure. We will have at least the classification bewteen phrases and words. The definition of a phrase will recursively relate subphrases and substrings, and define the phrase as a composition of subphrases and the string as the concatenation of substrings. The formalism does not impose constraints on how the relations between phrases and strings are defined, and the grammar writer has to define them explicitly. One possibility is to use context-free like mappings, using for example the same kind of encoding as in DCGs for PATR-like gramars or HPSG [Emele/Zajac 90b]. But other possibilities are available as well: using a kind of functional composition reminiscent of categorial grammars as in [Dymetman et al. 90], or linear precedence rules [Pollard/Sag 87, Reape 90].

For example, a rule like [Shieber 86][10]

$$\begin{aligned} S \rightarrow\ & NP\ VP: \\ & \langle S\ head \rangle = \langle VP\ head \rangle \\ & \langle S\ headform \rangle = finite \\ & \langle VP\ syncat\ first \rangle = \langle NP \rangle \\ & \langle VP\ syncat\ rest \rangle = \langle end \rangle. \end{aligned}$$

is encoded in TFS using a type S for the sentence type with two features **np** and **vp** for encoding the constituent structure, and similarly for NPs and VPs. The string associated with each constituent is encoded under the feature **string**. The string associated with the sentence is simply the concatenation of the string associated with the VP and the string associated with the NP: this constraint is expressed in a condition using the APPEND relation on lists (Figure 4).

The difference between the parsing and the generation problem is then only in the form of the term given to the interpreter for evaluation. An underspecified term where only the string is given defines the parsing problem:

$$\mathbf{S}\left[\mathbf{string}\text{: }\langle \mathbf{Uther\ storms\ Cornwall} \rangle\right]$$

An underspecified term where only the semantic form is given defines the generation problem:

[9]This is of course very reminiscent of HPSG, and it should not come as a surprise: HPSG is so far the only formal linguistic theory based on the notion of typed feature structures [Pollard/Sag 87]. A computational formalism similar to TFS is currently under design at CMU for implementing HPSG [Carpenter 90, Franz 90].

[10]Using a more condensed notation for lists with angle brackets provided by the TFS syntax: a list `CONS[first: Mary, rest: CONS[first: sings, rest: NIL]]` is written as `<Mary sings>`.

$$S\left[\text{head:}\left[\text{trans:}\begin{bmatrix}\text{pred: STORM}\\ \text{arg1: UTHER}\\ \text{arg2: CORNWALL}\end{bmatrix}\right]\right]$$

In both cases, the same interpreter uses the same set of rewrite rules to fill in "missing information" according to the grammar definitions. The result in both cases is exactly the same: a fully specified term containing the string, the semantic form, and also all other syntactic information like the constituent structure (Figure 5).

3.2 Bi-directional transfer in MT

We have sketched above a very general framework for specifying mappings between a linguistic structure, encoded as a feature structure and a string, also encoded as a feature structure. We apply a similar technique for specifying MT transfer rules, which we prefer to call "contrastive rules" since there is no directionality involved [Zajac 89, Zajac 90a].

The idea is rather simple: assume we are working with linguistic structures similar to LFG's functional structures for English and French [Kaplan et al. 89]. We define a translation relation as a type `TAU-LEX` with two features, `eng` for the English structure and `fr` for the French structure. This "bilingual sign" is defined on the lexical structure: each subtype of `TAU-LEX` defines a lexical correspondence between a partial English lexical structure and a partial French lexical structure for a given lexical equivalence. Such a lexical contrastive definition also has to pair the arguments recursively, and this is expressed in the condition part of the definition (Figure 6). The translation of syntactic features, like tense or determination, is also specified in the condition part, and these contrastive definitions are defined separately from the lexical definitions.

The transfer problem for one direction or the other is stated in the same way as for parsing or generation: the input term is an under-specified "bilingual sign" where only one structure for one language is given. Using the contrastive grammar, the interpreter fills in missing information and builds a completely specified bilingual sign[11].

4 THE TERMINATION PROBLEM AND EFFICIENCY ISSUES

For parsing and generation, since no constraint is imposed on the kind of mapping between the string and the semantic form, termination has to be proved for each class of grammar and the for the particular evaluation mechanism used for either parsing or generation with this grammar. If we restrict ourselves to class of grammars for which terminating evaluation algorithms are known, we can implement those directly in TFS. However, the TFS evaluation strategy allows more naive implementations of grammars and the outermost evaluation of "sub-goals" terminates on a strictly larger class of programs than for corresponding logic programs implemented in a conventional PROLOG. Furthermore, the grammar writer does not need, and actually should not, be aware of the control which follows the shape of the input rather than a fixed strategy, thanks to the lazy evaluation mechanism.

HPSG-style grammars do not cause any problem: completeness and coherence as defined for LFG, and extended to the general case

[11] See also [Reape 90] for a "Shake'n'Bake" approach to MT (Whitelock).

by [Wedekind 88], are implemented in HPSG using the "subcategorization feature principle" [Johnson 87]. Termination conditions for parsing are well understood in the framework of context-free grammars. For generation using feature structures, one of the problems is that the input could be "extended" during processing, i.e. arbitrary feature structures could be introduced in the semantic part of the input by unification with the semantic part of a rule. However, if the semantic part of the input is fully speficied according to a set of type definitions describing the set of well-formed semantic structures (and this condition is easy to check), this cannot arise in a type-based system. A more general approach is described in [Dymetman et al. 90] who define sufficient properties for termination for parsing and generation for the class of "Lexical Grammars" implemented in PROLOG. These properties seem generalizable to other classes of grammars as well, and are also applicable to TFS implementations. The idea is relatively simple and says that for parsing, each rule must consume a non empty part of the string, and for generation, each rule must consume a non empty part of the semantic form. Since Lexical Grammars are implemented in PROLOG, left-recursion must be eliminated for parsing and for generation, but this does not apply to TFS implementations.

Termination for reversible transfer grammars is discussed in [van Noord 90]. One of the problems mentioned is the extension of the "input", as in generation, and the answer is similar (see above). However, properties similar to the "conservative guides" of [Dymetman et al. 90] have to hold in order to ensure termination.

The lazy evaluation mechanism has an almost optimal behavior on the class of problems that have an exponential complexity when using the "generate and test" method [van Hentenryck/Dincbas 87, Aït-Kaci/Meyer 90]. It is driven by the availability of information: as soon as some piece of information is available, the evaluation of constraints in which this information appears is triggered. Thus, the search space is explored "intelligently", never following branches of computation that would correspond to uninstanciated PROLOG goals. The lazy evaluation mechanism is not yet fully implemented in the current version of TFS, but with the partial implementation we have, a gain of 50% for parsing has already been achieved (in comparison with the previous implementation using only the outer-most rewriting strategy).

The major drawback of the current implementation is the lack of an efficient indexing scheme for objects. Since the dictionaries are accessed using unification only, each entry is tried one after the other, leading to an extremely inefficient behavior with large dictionaries. However, we think that a general indexing scheme based on a combination of methods used in PROLOG implementations and in object-oriented database systems is feasible.

CONCLUSION

We have described a uniform constraint-based architecture for the implementation of reversible unification grammars. The advantages of this architecture in comparison of more traditional logic (i.e. PROLOG) based architectures are: the input/output distinction is truly abolished; the evaluation terminates on a strictly larger class of problems; it is directly based on typed feature structures, not first order terms; a single fully data-driven constraint evaluation scheme is used; the

constraint evaluation scheme is directly derived from the semantics of typed feature structures. Thus, the TFS language allows a direct implementation of reversible unification grammars. Of course, it does not dispense the grammar designer with the proof of general formal properties that any well-behaved grammar should have, but it does allow the grammar writer to develop grammars without thinking about any notion of control or input/output distinction.

References

[Aït-Kaci 84] Hassan Aït-Kaci. *A Lattice Theoretic Approach to Computation based on a Calculus of Partially Ordered Types Structures.* Ph.D Dissertation, University of Pennsylvania.

[Aït-Kaci 86] Hassan Aït-Kaci. "An Algebraic Semantics Approach to the Effective Resolution of Type Equations". *Theoretical Computer Science 45*, 293-351.

[Aït-Kaci/Meyer 90] Hassan Aït-Kaci and Richard Meyer. "Wild_LIFE, a user manual". PRL Technical Note 1, Digital Equipement Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[Calder et al. 89] Jonathan Calder, Mike Reape and Henk Zeevat. "An algorithm for generation in unification grammars". Proc. of the *4th Conference of the European Chapter of the Association for Computational Linguistics,* 10–12 April 1989, Manchester.

[Carpenter 90] Bob Carpenter. "Typed feature structures: inheritance, (in)equality and extensionality". Proc. of the Workshop on Inheritance in Natural Language Processing, Institute for Language Technology and AI, Tilburg University, Netherlands, August 1990.

[Dershowitz/Plaisted 88] N. Dershowitz and D.A. Plaisted. "Equational programming". *In* Hayes, Michie and Richards (eds.). *Machine Intelligence 11.* Clarendon Press, Oxford, 1988.

[Dymetman/Isabelle 88] Marc Dymetman and Pierre Isabelle. "Reversible logic grammars for machine translation". Proc. of the *2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language*, June 1988, Pittsburgh.

[Dymetman et al. 90] Marc Dymetman, Pierre Isabelle and François Perrault. "A symmetrical approach to parsing and generation". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, Helsinki, August 1990.

[Emele 91] Martin Emele. "Unification with lazy non-redundant copying". *29th Annual Meeting of the ACL*, June 1991, Berkeley, CA.

[Emele/Zajac 90a] Martin Emele and Rémi Zajac. "A fixed-point semantics for feature type systems". Proc. of the *2nd Workshop on Conditional and Typed Rewriting Systems - CTRS'90*, Montreal, June 1990.

[Emele/Zajac 90b] Martin Emele and Rémi Zajac. "Typed Unification Grammars". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, Helsinki, August 1990.

[Emele et al. 90] Martin Emele, Ulrich Heid, Stefan Momma and Rémi Zajac. "Organizing linguistic knowledge for multilingual generation". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, Helsinki, August 1990.

[Franz 90] Alex Franz. "A parser for HPSG". CMU report CMU-LCL-90-3, Laboratory for Computational Linguistics, Carnegie Mellon University, July 1990.

[Isabelle et al. 88] Pierre Isabelle, Marc Dymetman and Eliot Macklovitch. "CRITTER: a translation system for agricultural market reports.". Proc. of the *12th International Conference on Computational Linguistics - COLING'88*, August 1988, Budapest.

[Johnson 87] Mark Johnson. "Grammatical relations in attribute-value grammars". Proc. of the *West Coast Conference on Formal Linguistics*, Vol.6, Stanford, 1987

[Kaplan et al. 89] Ronald M. Kaplam, Klaus Netter, Jürgen Wedekind, Annie Zaenen. "Translation by structural correspondences". Proc. of the *4th European ACL Conference*, Manchester, 1989.

[Klop 90] Jan Willem Klop. "Term rewriting systems". To appear in S. Abramsky, D. Gabbay and T. Maibaum. *Handbook of Logic in Computer Science*, Vol.1, Oxford University Press.

[Newman 90] P. Newman. "Towards convenient bidirectional grammar formalisms". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, August 1990, Helsinki.

[Pereira/Warren 83] Fernando C.N. Pereira and David Warren. "Parsing as deduction". Proc. of the *21st Annual Meeting of the ACL*, 15–17 June 1983, Cambridge, MA.

[Pollard/Sag 87] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics.* CSLI Lecture Notes 13, Chicago University Press, 1987.

[Pollard/Moshier 89] Carl Pollard and Drew Moshier. "Unifiying partial descriptions of sets". In P. Hanson (ed.) *Information, Language and Cognition*, Vancouver Studies in Cognitive Science 1, University of British Columbia Press, Vancouver.

[Reape 90] Mike Reape. "Parsing semi-free word order and bounded discontinuous constituency and "shake 'n' bake" machine translation (or 'generation as parsing')". Presented at the *International Workshop on Constraint Based Formalisms for Natural Language Generation*, Bad Teinach, Germany, November 1990.

[Russell et al. 90] Graham Russell, Susan Warwick and John Carroll. "Asymmetry in parsing and generation with unification grammars: case studies from ELU". Proc. of the *28th Annual Meeting of the ACL*, 6–9 June 1990, Pittsburgh.

[Shieber 86] Stuart Shieber. *An Introduction to Unification-based Grammar Formalisms.* CSLI Lectures Notes 4, Chicago University Press, 1986.

[Shieber 88] Stuart Shieber. "A uniform architecture for parsing and generation". Proc. of the *12th International Conference on Computational Linguistics - COLING'88*, August 1988, Budapest.

[Shieber et al. 89] Stuart Shieber, Gertjan van Noord, Robert Moore and Fernando Pereira. "A uniform architecture for parsing and generation". Proc. of the *27th Annual Meeting of the ACL*, 26–27 June 1989, Vancouver.

[Strzalkowski 90] Tomek Strzalkowski. "How to invert a natural language parser into an efficient generator: an algorithm for logic grammars". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, August 1990, Helsinki.

[van Hentenryck/Dincbas 87] P. van Hentenryck and M. Dincbas. "Forward checking in logic programming". Proc. of the *4th International Conference on Logic Programming*, Melbourne, May 1987.

[van Noord 90] Gertjan van Noord. "Reversible unification based machine translation". Proc. of the *13th International Conference on Computational Linguistics - COLING'90*, August 1990, Helsinki.

[Wedekind 88] Jürgen Wedekind. "Generation as structure driven generation". Proc. of the *12th International Conference on Computational Linguistics - COLING'88*, August 1988, Budapest.

[Zajac 89] Rémi Zajac. "A transfer model using a typed feature structure rewriting system with inheritance". Proc. of the *27th Annual Meeting of the ACL*, 26–27 June 1989, Vancouver.

[Zajac 90a] Rémi Zajac. "A relational approach to translation". Proc. of the *3rd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Language*, 11–13 June 1990, Austin.

[Zajac 90b] Rémi Zajac. "Computing partial information using approximations - Semantics of typed feature structures". Presented at the *International Workshop on Constraint Based Formalisms for Natural Language Generation*, Bad Teinach, Germany, November 1990.