Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Research Report
RR-94-39

Typed Feature Formalisms as a Common Basis for Linguistic Specification

Hans-Ulrich Krieger

November 1994

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

Typed Feature Formalisms as a Common Basis for Linguistic Specification.

Hans-Ulrich Krieger

DFKI-RR-94-39

To appear in: Machine Translation and the Lexicon,
Lecture Notes in Artificial Intelligence,
Springer, 1995.

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-9002 0).

ISSN 0946-008X

Typed Feature Formalisms as a Common Basis for Linguistic Specification.

Hans-Ulrich Krieger
krieger@dfki.uni-sb.de
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany

Abstract. Typed feature formalisms (TFF) play an increasingly important role in CL and NLP. Many of these systems are inspired by Pollard and Sag's work on Head-Driven Phrase Structure Grammar (HPSG), which has shown that a great deal of syntax and semantics can be neatly encoded within TFF. However, syntax and semantics are not the only areas in which TFF can be beneficially employed. In this paper, I will show that TFF can also be used as a means to model finite automata (FA) and to perform certain types of logical inferencing. In particular, I will (i) describe how FA can be defined and processed within TFF and (ii) propose a conservative extension to HPSG, which allows for a restricted form of semantic processing within TFF, so that the construction of syntax and semantics can be intertwined with the simplification of the logical form of an utterance. The approach which I propose provides a uniform, HPSG-oriented framework for different levels of linguistic processing, including allomorphy and morphotactics, syntax, semantics, and logical form simplification.

Acknowledgements. This paper has benefited from numerous people at various workshops where parts of it have been presented, in particular, at the *Sprachwissenschaftliches Kolloquium* (Univ. of Tübingen), the *31st Annual Meeting of the ACL* (Columbus, Ohio), and the *International Workshop on "Machine Translation and the Lexicon" of the European Association for Machine Translation* (Heidelberg).
I would like to thank Elizabeth Hinkelman for reading a draft of this paper. I'm especially indebted to Petra Steffens for carefully reading the pre-final version and for making detailed suggestions.

Table of Contents

1 Introduction

Pollard&Sag's seminal work on Head-Driven Phrase Structure Grammar has shown that a great deal of syntax and semantics can be neatly encoded within typed feature structures, thus leading for the first time to a highly lexicalized theory of language [20, 21]. Moreover, the formalisms underlying these structures can be given a precise set-theoretical semantics along the lines of Smolka and others.[1] However, there are certain areas within computational linguistics, for which, until recently, no satisfactory formulation in a uniform, constraint-based (or more specifically, HPSG-oriented) theory has been provided. Two of these representation problems will be addressed in this paper, viz., *finite automata* and *logical form simplification.*

2 Finite Automata as Typed Feature Structures

Finite automata (FA) and similar devices are heavily used in computational linguistics and natural language processing as a descriptive means of stating certain facts about natural language. They have been employed in the description of morphophonemics [11, 3] and in the formulation of word order constraints [26]; moreover, the use of FA allows for the integration of allomorphy and morphotactics [15, 12].

While it is unsurprising that the languages accepted by FA may also be encoded as typed feature descriptions, it is not clear how FA themselves can be specified as feature structures, how they can be processed, and, furthermore, what closure properties they have within TFF. These questions and, of course, their solutions will be addressed in this section.

2.1 Preliminaries

Assuming a familiarity with the basic inventory of automata theory and formal languages [6], we shall, in the following, formally refer to a *deterministic finite automaton* (DFA) by a 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of *states*, Σ a finite *input alphabet*, $\delta : Q \times \Sigma \mapsto Q$ is the *transition function*, $q_0 \in Q$ the *initial state*, and $F \subseteq Q$ the set of *final states*. A *nondeterministic finite automaton* (NFA) differs from a deterministic one in that the transition function δ maps to elements of the power set of Q, i.e., $\delta : Q \times \Sigma \mapsto 2^Q$ (Q, Σ, q_0, and F as before).

This is all we need to explain the encoding technique for FA within a typed feature logic. For reasons of simplicity, we start with the simplest form of FA, viz., *deterministic* finite automata *without* ϵ-moves, which consume *exactly one* input symbol at a time. Note that this is not a restriction w.r.t. the set of recognized words: given an arbitrary NFA, we can always construct a deterministic one which recognizes the same language (however, in the worst case with exponentially more states).

[1] In the following, we will assume a basic familiarity with unification-based grammar theories [23, 25] and their logics [9, 7, 24].

Fortunately, our approach is also capable of directly representing and processing non-deterministic FA with ϵ-moves, and allows for edges which are multiple-symbol consumers (see next section). It is worth noting that edges may not only be annotated with atomic symbols. They can also be labelled with complex ones, i.e., with possibly underspecified feature structures, where unification is a means for testing equality (for instance, in case of 2-level morphological descriptions; see [16] for an example of a paradigm-based inflectional morphology).

2.2 Encoding Finite Automata Within Typed Feature Formalisms

To specify an automaton as a typed feature structure, we introduce for every state $q \in Q$ a possibly recursive feature type with the same name as q. We will call such a type a *configuration*. Exactly the attributes `EDGE`, `NEXT`, and `INPUT` are appropriate for such a configuration, where `EDGE` encodes the *outgoing edges* of q, `NEXT` the *successor states* of q, and `INPUT` the symbols which remain on the *input list* when reaching q.[2] A configuration does thus not just model a state of the automaton, but an entire description of the FA at a given point in computation.[3] In order to formally define a configuration as a feature structure type, we first introduce the notion of a proto configuration that specifies the appropriate attributes and their values.

$$\textit{proto-configuration} \equiv \begin{bmatrix} \texttt{EDGE} & \textit{input-symbol} \vee \textit{undef} \\ \texttt{NEXT} & \textit{configuration} \vee \textit{undef} \\ \texttt{INPUT} & \textit{list(input-symbol)} \end{bmatrix} \tag{1}$$

We now define two natural subtypes of *proto-configuration*. The first one represents the *non-final states* $Q \setminus F$. Because we assume that exactly one input symbol is consumed every time an edge is traversed, we separate the input list into the first element and the rest list, structure-share the first element with `EDGE` (the consumed input symbol), and pass the rest of the list one level deeper to the next state.

$$\textit{non-final-configuration} \equiv \begin{bmatrix} \textit{proto-configuration} \\ \texttt{EDGE}\ \boxed{1} \\ \texttt{NEXT}|\texttt{INPUT}\ \boxed{2} \\ \texttt{INPUT}\ \langle \boxed{1} \cdot \boxed{2} \rangle \end{bmatrix} \tag{2}$$

The other subtype encodes the *final states* of F which possess no outgoing edges, therefore no successor states (and vice versa), or in our terminology: `EDGE`

[2] There might exist states in an FA with no outgoing edges and thus with no successor states. To cope with this fact, we introduce a special subtype of the most general type $\top$, called *undef*, which is *incompatible* with every other type (except with itself and $\top$).

[3] Note the similarity between a configuration and a closure in functional programming or a machine state in operational semantics—all notions exhaustively describe the corresponding computing device at a certain point in time.

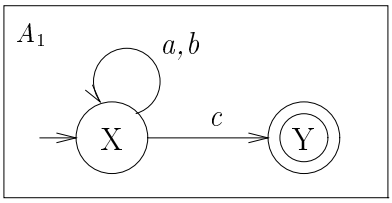
and NEXT are undefined (are of type *undef*). In addition, successfully reaching a final state with no outgoing edge implies that the input list is *empty*.

$$\textit{final-configuration} \equiv \begin{bmatrix} \textit{proto-configuration} \\ \texttt{EDGE}\ \ \textit{undef} \\ \texttt{NEXT}\ \ \textit{undef} \\ \texttt{INPUT}\ \ \langle\ \rangle \end{bmatrix} \tag{3}$$

Of course, there will also be final states *with* outgoing edges, but such states are subtypes of the following *disjunctive* type specification:

$$\textit{configuration} \equiv \textit{non-final-configuration} \vee \textit{final-configuration} \tag{4}$$

To make things more concrete, let us look at an example, viz., the FA A_1 which recognizes the language $\mathcal{L}(A_1) = (a+b)^\star c$.



A_1 consists of the two states X and Y; therefore, we have to define two types X and Y, where Y (given in (5)) is only an instantiation of a final configuration. Note that we make use of *distributed disjunctions* [5] (depicted by the disjunction name \$1) in the definition of X to express the covariation between edges and successor states: if a is processed, use type X (and vice versa), if b is processed, use again type X, but if c is chosen, choose type Y.

$$X \equiv \begin{bmatrix} \textit{non-final-configuration} \\ \texttt{EDGE}\ \ {}_{\$1}(\texttt{a} \vee \texttt{b} \vee \texttt{c}) \\ \texttt{NEXT}\ \ {}_{\$1}(X \vee X \vee Y) \end{bmatrix} \tag{5}$$

$$Y \equiv [\,\textit{final-configuration}\,]$$

Whether a FA A *accepts* a given input string or not is thus equivalent to the question of *feature term consistency/satisfiability*: if we want to know whether w (a list of input symbols) will be recognized by A, we must *expand the type* which is associated with the initial state q_0 of A and specify w as its INPUT. Speaking in Carpenter's terms [4], we thus require that

$$q_0 \wedge [\texttt{INPUT}\ \ w]$$

be *totally well-typable*, i.e., that there is at least one model that satisfies the input description.[4]

The processing of FA within TFF is thus achieved by type expansion of possibly recursive feature types. However, type expansion not only tests for the satisfiability of a description but also makes the idiosyncratic and inherited constraints of a type explicit (see below). In our case, type expansion *always terminates*, either with a *unification failure* (the FA does not accept w) or with a fully expanded feature structure, representing a successful recognition.

Coming back to our example, let us ask whether `abc` belongs to the language $\mathcal{L}(A_1)$ accepted by A_1. By expanding type X with [INPUT ⟨a,b,c⟩], we can decide this question. This will lead to the following consistent feature structure, which represents the *complete recognition history* of `abc`, i.e., *all* its "solutions" in the FA (recall that because X is a subtype of *non-final-configuration* and *proto-configuration*, it will *inherit* all constraints of these types; similar for Y):

$$
\left[\begin{array}{ll}
X & \\
\text{EDGE } \boxed{1}\ \mathtt{a} & \\
\quad \left[\begin{array}{ll}
X & \\
\text{EDGE } \boxed{3}\ \mathtt{b} & \\
\quad \left[\begin{array}{ll}
X & \\
\text{EDGE } \boxed{5}\ \mathtt{c} & \\
\quad \left[\begin{array}{ll}
Y & \\
\text{EDGE } \textit{undef} & \\
\text{NEXT } \textit{undef} & \\
\text{INPUT } \boxed{6}\ \langle\ \rangle &
\end{array}\right] & \\
\text{INPUT } \boxed{4}\ \langle\ \boxed{5}\ .\ \boxed{6}\ \rangle &
\end{array}\right] & \\
\text{INPUT } \boxed{2}\ \langle\ \boxed{3}\ .\ \boxed{4}\ \rangle &
\end{array}\right] & \\
\text{INPUT } \langle\ \boxed{1}\ .\ \boxed{2}\ \rangle &
\end{array}\right]
\qquad \begin{array}{l} \text{NEXT} \\ \text{NEXT} \\ \text{NEXT} \end{array}
\tag{6}
$$

We now change our focus from DFA to arbitrary NFA. The first question we have to ask is whether *nondeterminism* in general makes the whole encoding method invalid. In fact, nondeterminism does not introduce any problems at all. There is no difference in our framework between a DFA and a NFA, neither from a descriptive nor from an expressive standpoint, because outgoing edges labelled with the same symbol (the NFA criterion) can be easily captured by distributed disjunctions, as is done in the DFA example above (cf. the description of type X given by (5) in FA A_1).[5]

In addition, changing from Σ- to $\Sigma^\star$-consuming edges leads only to minor modifications in the definition of *non-final-configuration* (2). Multiple-symbol

[4] Type expansion here is analogous to a top-down parsing method in syntactic analysis, viz., *recursive descent parsing*. Note that the satisfiability problem for recursive type descriptions is *in general* undecidable, although this is not the case for our encoding [14].

[5] From a *processing* standpoint, of course, a DFA differs from a NFA in our approach. We will come back to this later.

consuming edges are modelled through lists of symbols instead of declaring single symbols appropriate for EDGE: an ϵ-transition (Σ^0) is encoded as the empty list (7), a single input symbol (Σ^1) through a list over this symbol (8), two input symbols (Σ^2) are represented using a list of two symbols (9), and so on. Therefore, we substitute the definition of *non-final-configuration* by giving a family of specialized definitions, where the number of definitions depends on the length of the longest word associated with an edge in the FA.

$$\textit{non-final-configuration}_0 \equiv \left[\begin{array}{l} \textit{proto-configuration} \\ \texttt{EDGE}\ \langle\,\rangle \\ \texttt{NEXT|INPUT}\ \boxed{1} \\ \texttt{INPUT}\ \boxed{1} \end{array}\right] \tag{7}$$

$$\textit{non-final-configuration}_1 \equiv \left[\begin{array}{l} \textit{proto-configuration} \\ \texttt{EDGE}\ \langle\,\boxed{1}\,\rangle \\ \texttt{NEXT|INPUT}\ \boxed{2} \\ \texttt{INPUT}\ \langle\,\boxed{1}\ \texttt{.}\ \boxed{2}\,\rangle \end{array}\right] \tag{8}$$

$$\textit{non-final-configuration}_2 \equiv \left[\begin{array}{l} \textit{proto-configuration} \\ \texttt{EDGE}\ \langle\,\boxed{1}\,,\,\boxed{2}\,\rangle \\ \texttt{NEXT|INPUT}\ \boxed{3} \\ \texttt{INPUT}\ \langle\,\boxed{1}\,,\,\boxed{2}\ \texttt{.}\ \boxed{3}\,\rangle \end{array}\right] \tag{9}$$

Under these circumstances, *configuration* (4) must also be altered, since it now consists of multiple alternatives:

$$\textit{configuration} \equiv \textit{final-configuration} \vee \bigvee_{i=0}^{n} \textit{non-final-configuration}_i \tag{10}$$

It is worth to have a look at the *complexity* of our approach. We all know that in the case of DFA, input can be recognized in $O(n)$, whereas the time complexity for a NFA is $O(2^n)$ in the worst case, where n is given by the length of the input string. Because we employ disjunctions to describe the covariation between edges and successor states, one might assume that the complexity of our treatment is already exponential for the DFA case as a result of the fact that the satisfiability problem for disjunctive formulae is $\mathcal{NP}$-complete [9], thus a unification algorithm will have a non-polynomial complexity, assuming that $\mathcal{P} \neq \mathcal{NP}$. Recall that we are using unification as a means for testing equality.

However, when modelling DFA in our approach, the disjunctions under EDGE and NEXT will *collapse* into one element as a consequence of the fact that in a DFA *at most one* arc can be traversed at a time (the one whose label matches the input). We therefore have to expand only *one* type under NEXT and unification only operates on conjunctive descriptions. But if this is the case, our treatment has nearly the same complexity as in theory: there exist well-known *quasi-linear* unification algorithms for conjunctive formulae, for instance Aït-Kaci's unification algorithm employed in LOGIN [1], which is an extension of Huet's method

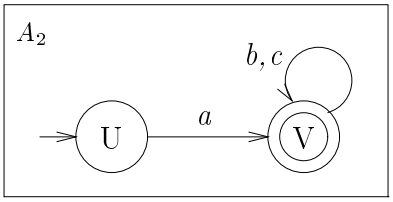
for fixed-arity, first-order terms. By encoding general NFA in our framework, we obtain the same theoretical result as is the case for a direct encoding, viz., exponential time complexity.

2.3 Intersection, Union, and Complementation of FA

As a nice by-product of our encoding technique, we can show that *unification, disjunction*, and *classical negation* in the underlying feature logic directly correspond to the *intersection, union*, and *complementation* of FA. The correspondences can be easily shown when assuming a sorted set-theoretical semantics for feature descriptions [24].

Take, for instance, the intersection of two arbitrary FA, A_1 and A_2. Intersecting A_1 and A_2 means construction of an FA A which recognizes the intersection of $\mathcal{L}(A_1)$ and $\mathcal{L}(A_2)$. But exactly this is achieved through unification: constructing A is equivalent to unifying the types associated with the start states of A_1 and A_2, q_0 and q_0'; the denotation of $q_0 \wedge q_0'$ is then given by the intersection of the objects denoted by q_0 and q_0'. The same argumentation holds for union and complementation of FA.

To see how this is accomplished, consider A_1 (as before) and A_2, which recognizes the language $\mathcal{L}(A_2) = a(b + c)^\star$.



To model A_1 and A_2, we refer to the types X and Y of (5) and to U and V, which are defined in (11).

$$U \equiv \begin{bmatrix} \textit{non-final-configuration} \\ \texttt{EDGE a} \\ \texttt{NEXT } V \end{bmatrix}$$
$$V \equiv \begin{bmatrix} \textit{configuration} \\ \texttt{EDGE } \$1(\texttt{b} \vee \texttt{c} \vee \textit{undef}) \\ \texttt{NEXT } \$1(V \vee V \vee \textit{undef}) \end{bmatrix} \tag{11}$$

The intersection of A_1 and A_2 then corresponds to the unification of X and U, which leads to the following structure (assuming that our logic is based on an open-world semantics [17]):

$$\begin{bmatrix} X \\ \texttt{EDGE } \$1(\texttt{a} \vee \texttt{b} \vee \texttt{c}) \\ \texttt{NEXT } \$1(X \vee X \vee Y) \end{bmatrix} \wedge \begin{bmatrix} U \\ \texttt{EDGE a} \\ \texttt{NEXT } V \end{bmatrix} = \begin{bmatrix} X \wedge U \\ \texttt{EDGE a} \\ \texttt{NEXT } X \wedge V \end{bmatrix} \tag{12}$$

Testing whether a given string w belongs to $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is equivalent to testing for the satisfiability of $q_0 \wedge q_0' \wedge [\texttt{INPUT}\ \ w]$. Again, type expansion decides the consistency of the given input description; see (13). Note that the unification of q_0 and q_0' has the same effect as running A_1 and A_2 in "parallel" which is equivalent to the intersection of A_1 and A_2, exactly what we want to achieve. Again, a similar argumentation holds for the union and complementation of FA; see (14) and (15).

$$w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \Longleftrightarrow q_0 \wedge q_0' \wedge [\texttt{INPUT}\ \ w] \neq \bot, \tag{13}$$

$$w \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \Longleftrightarrow (q_0 \vee q_0') \wedge [\texttt{INPUT}\ \ w] \neq \bot \tag{14}$$

$$w \in \overline{\mathcal{L}(A_1)} \Longleftrightarrow \neg q_0 \wedge \textit{configuration} \wedge [\texttt{INPUT}\ \ w] \neq \bot \tag{15}$$

Because we are working in the domain of FA (although they are encoded via feature structures), complementing an FA means to complement the language it accepts with respect to $\Sigma^\star$ and not to complement the set of objects denoted by q_0 with respect to the domain of feature descriptions, i.e., the whole universe (which represents a much larger set). We, therefore, have to intersect/unify $\neg q_0$ with *configuration* in (15) in order to restrict ourselves to the domain of feature structures which *model* FA.

To see how the proposed mechanism works, let us look at the FA A_1 and A_2 again and let us ask whether $abc \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$? Deciding this question means to expand $\mathrm{X} \wedge \mathrm{U} \wedge [\texttt{INPUT}\ \langle \texttt{a,b,c} \rangle]$ which results in (16).

$$\left[\begin{array}{l} X \wedge U \\ \texttt{EDGE}\ \boxed{1}\ \texttt{a} \\ \texttt{NEXT}\ \left[\begin{array}{l} X \wedge V \\ \texttt{EDGE}\ \boxed{3}\ \texttt{b} \\ \texttt{NEXT}\ \left[\begin{array}{l} X \wedge V \\ \texttt{EDGE}\ \boxed{5}\ \texttt{c} \\ \texttt{NEXT}\ \left[\begin{array}{l} Y \wedge V \\ \texttt{EDGE}\ \textit{undef} \\ \texttt{NEXT}\ \textit{undef} \\ \texttt{INPUT}\ \boxed{6}\ \langle\ \rangle \end{array}\right] \\ \texttt{INPUT}\ \boxed{4}\ \langle\ \boxed{5}\ .\ \boxed{6}\ \rangle \end{array}\right] \\ \texttt{INPUT}\ \boxed{2}\ \langle\ \boxed{3}\ .\ \boxed{4}\ \rangle \end{array}\right] \\ \texttt{INPUT}\ \langle\ \boxed{1}\ .\ \boxed{2}\ \rangle \end{array}\right] \tag{16}$$

It has to be noted that the intersection of FA via unification does not work in general for FA with ϵ-moves. This problem is inherent and well-known but is no restriction w.r.t. expressivity (see [14] for more details and related aspects).

2.4 Concatenation and Kleene Closure

Let us now focus on the *concatenation* and *Kleene closure* of regular expressions/FA. It turns out that the feature logic on which our approach is based together with a weak form of *functional uncertainty* [8] allows for a characterization of these operations [14]. Let $A_1 = \langle Q_1, \Sigma_1, \delta_1, q_0, F_1 \rangle$ and $A_2 = \langle Q_2, \Sigma_2, \delta_2, q'_0, F_2 \rangle$ be two arbitrary FA. The concatenation of A_1 and A_2 is given by

$$A_1 \cdot A_2 \equiv q_0 \wedge [(\texttt{NEXT})^* \;\; q'_0 \wedge \bigvee_i f_i] \tag{17}$$

where the f_i must be subtypes of *non-final-configuration*, although on the FA level, they belong to the set of final states. While $A_1 \cdot A_2$ would usually be constructed by introducing an ϵ-move between A_1 and A_2 [6, p. 31], we account for concatenation by connecting every final state $f_i \in F_1$ with the start state q'_0 of A_2; thus, we have to write $(\vee_i f_i) \wedge q'_0$. Connection here does *not* mean introducing an ϵ-move but to *unify* every f_i with q'_0, which requires us to turn the final states of A_1 into non-final ones to allow for successful unifications; this is why f_i must be a subtype of *non-final-configuration*.

At this point, functional uncertainty comes into play because we do not know for a concrete input $w = w_1 \cdot w_2$ how many iterations of NEXT are necessary in A_1 to successfully recognize w_1, so that w_2 can be further processed by A_2. Note that the functional uncertainty constraint in (17) can be restated by using the following recursive type definition—thus there is no need for a richer logic:

$$[(\texttt{NEXT})^* \;\; \sigma] \quad \rightsquigarrow \quad \Phi \equiv \sigma \vee [\texttt{NEXT} \;\; \Phi] \tag{18}$$

The iteration or Kleene closure of A_1 is constructed in a similar way: the final states $f_i \in F_1$ are *unified* with the start state q_0 (to be more precise, with the types associated with these states). The construction of A_1^* then looks as follows:

$$A_1^* \equiv A^0 \vee A^+ \tag{19}$$

where A^0 is an instantiation of *final-configuration* (the empty string case) and $A^+ \equiv q_0 \wedge \left[(\texttt{NEXT})^* \;\vee_i f_i\right]$. However, f_i must be a subtype of the disjunctive type *configuration* (4) because the f_i serve as final states as well as non-final states in this construction, which is in accordance with the definition of *configuration*.

Although concatenation and Kleene closure are directly encodable in our logic, we recommend against using the above technique for reasons of efficiency. In this regard, it is better to construct the composite automaton first by hand—which is fairly straightforward—and then apply the encoding mechanism for non-complex FA.

3 Logical Form Simplification Within HPSG

Typed feature formalisms in general, and HPSG in particular, serve as a basis for many NLP/MT systems [27, 28, 10]. Even though most of these systems represent the semantic content of an utterance as a feature structure, they do *not* use a parser (or generator) or a uniform deduction component to simplify logical form or to draw domain-specific inferences within the calculus of HPSG in order to derive legal, simpler expressions represented as a feature structure again (cf. [2] to get an impression of simplifying/resolving (quasi) logical form within the core language engine of SRI).

Instead, all systems either translate the semantic representation directly into an application language (e.g., a database language), which means that semantic inferences are not seen as essential in the front-end, or transform feature structures into a term of a semantic representation logic (for instance the language $\mathcal{NLL}$ [18]), on which a deduction component operates to yield another, denotation-preserving expression. Given such an intermediate language, the method of processing the semantics of a sentence is as follows:

1. incrementally construct a feature structure f representing the semantics of a given sentence,
2. transform the content of f into a term t of the intermediate language,
3. apply simplification schemata iteratively to t, yielding a simpler term t',
4. translate t' into an application language expression e,
5. interpret e with the inference machinery of the application language.

We will argue in this paper that semantic inferences can be carried out locally as part of the parsing (generation) process so that step (2.) and (3.) are in fact *not* needed and that f can be directly translated into e. Doing away with an intermediate level of semantic representation has many advantages:

- PROCESSING: semantic inferences can be carried out locally during the parsing process (if needed); since inconsistencies can thus be detected at an early stage of analysis, processing efforts can be reduced
- ARCHITECTURE: semantic inferences are integrated into the parser—which leads to a simpler architecture of the whole NLP system
- EFFICIENCY: there is no need to transform a feature structure into an expression of the intermediate language—which saves time and space
- UNIFORMITY: it is theoretically appealing to provide a coherent framework in which all levels of linguistic description are represented and in which artificial interface problems are thus avoided

Because HPSG in general allows for higher order expressivity through unrestricted relations and recursive types, the notion of *logical equivalence* of descriptions is undecidable, and moreover, not even recursively enumerable. Hence the subject of this paper will not be a restricted decision procedure for testing the equivalence of two descriptions, but, rather, a limited method of logical form simplification. This is achieved by enriching the feature logic underlying HPSG—however, without sticking to external relational constraints.

3.1 Encoding Logical Form Simplification

In the following, we refer to Pollard and Sag's first volume of HPSG [20]. Even though the examples given throughout this section are simplified in that the structure of SEM is *flat*, i.e., only consists of top level attributes like OP (operator), SC (scope), CONN (connective), etc., the idea developed here can be easily adapted to more complex forms of HPSG and other constraint-based grammar formalisms which have similar notions of what English (or any natural language) is [20, p. 147]:

$$\mathit{English} = P_1 \wedge \cdots \wedge P_{n+m} \wedge (L_1 \vee \cdots \vee L_p \vee R_1 \vee \cdots \vee R_q) \tag{20}$$

In the introductory section, we said that during parsing the primary reason for using feature structures is the need for storing information obtained so far (e.g., semantic content). A parser, however, will, for instance, *not* simplify nested occurrences of an operator like a *semantic not* $\neg$.[6] There's a notable exception to what we said about the lack of semantic inferences in HPSG: most of the effects of *β-reduction*, used by many semanticists growing out of the Montagovian tradition, can be easily captured by *unification* (see for instance [19]).

In this section, we intend to present the necessary inventory for *logical form simplification* within HPSG. What we need is

1. an immediate dominance (rule) schema R_{proj} formulated as (Project) in (22) to record semantic inferences, and
2. for each simplification schema exactly one *extralinguistic/metalogical principle* P_{meta_i} $(1 \leq i \leq k)$ realized as (a special form of) an implication.

Therefore, we must redefine (20) by adding the rule schema and the principles. This results in the following definition of *English*:

$$P_1 \wedge \cdots \wedge P_{n+m} \wedge P_{\mathrm{meta}_1} \cdots \wedge P_{\mathrm{meta}_k} \wedge (L_1 \vee \cdots \vee L_p \vee R_1 \vee \cdots \vee R_q \vee R_{\mathrm{proj}}) \tag{21}$$

The rule schema R_{proj} serves to represent both sides of an inference step by projecting the simplified semantics to the top level SEM and storing the non-simplified representation under DTRS; see (22). Note the similarity between R_{proj} and an R_i: R_i serves as an instruction to build up phrase structure. However, the number of branches in such a derivation tree is in general greater than one—this is in contrast to the single daughter of R_{proj}. The idea now is to postulate a similar structure which allows us to construct a *proof tree*. Topologically speaking, such a proof tree corresponds to a linear chain. Because we are interested in the value of the SEM attribute, we structure-share PHON and SYN on the top level with the same attributes of the single daughter under the path DTRS|NON-SIMPL-DTR. This is necessary for a parser to continue (syntactic parsing) properly.

[6] For example, an expression like [SEM|CONT [OP $\neg$, SC [OP $\neg$, SC ψ]]] should be simplified in many cases to [SEM|CONT ψ].

(Project)

$$\left[\begin{array}{l} \textit{phrasal-sign} \\ \texttt{PHON}\ \boxed{1} \\ \texttt{SYN}\ \boxed{2} \\ \texttt{SEM}\ \boxed{3} \\ \texttt{DTRS} \left[\begin{array}{l} \textit{non-simpl-dtr-struct} \\ \texttt{NON-SIMPL-DTR} \left[\begin{array}{l} \textit{sign} \\ \texttt{PHON}\ \boxed{1} \\ \texttt{SYN}\ \boxed{2} \\ \texttt{SEM}\ \boxed{4} \end{array}\right] \end{array}\right] \\ \boxed{3} \neq \boxed{4} \end{array}\right] \tag{22}$$

Note that it is always possible to instantiate such a structure, if needed, *during* the construction of syntax and semantics in order to simplify the value of SEM (cf. the examples in Section 3.2). After a successful simplification step, we may then continue with syntactic analysis and possibly perform some more simplification steps again later.

To avoid *interferences* between linguistic principles and extralinguistic ones, we assume DTRS to be of type *non-simpl-dtr-struct*; see (22). Thus, we exclude the application of principles like the Head Feature Principle, the Semantics Principle, or the Subcategorization Principle. Because those principles are of the form

$$[\texttt{DTRS}\ \ [\textit{headed-structure}]\,] \implies [\ \cdots\ ] \tag{23}$$

they cannot be applied to structures which are licensed by the projection rule schema (22). The same argument also holds for the opposite case: structures admitted by the four rule schemata of HPSG-I, cannot be constrained by our extralinguistic principles, because the antecedents of such principles assume a single daughter of type *non-simpl-dtr-struct*, which would cause the principles to fail.

We now present two well-known simplification schemata and show how to represent them in terms of feature structure implications—actually, we only represent *one* direction of the biconditional (otherwise we would have to state two implications). We start with the simplification schema for *double negation*, i.e.,

$$\frac{\neg\neg\psi}{\psi} \tag{24}$$

or as an implication

$$\textbf{(2Neg)}\quad \begin{bmatrix} \textit{phrasal-sign} \\ \texttt{DTRS} \begin{bmatrix} \textit{non-simpl-dtr-struct} \\ \texttt{NON-SIMPL-DTR|SEM} \begin{bmatrix} \textit{op-sc-struct} \\ \texttt{OP}\ \neg \\ \texttt{SC|OP}\ \neg \end{bmatrix} \end{bmatrix} \end{bmatrix} \quad (25)$$

$$\Longrightarrow \begin{bmatrix} \textit{phrasal-sign} \\ \texttt{SEM}\ \boxed{1} \\ \texttt{DTRS|NON-SIMPL-DTR|SEM|SC|SC}\ \boxed{1} \end{bmatrix}$$

Note the special form of the left-hand side: (25) can only be applied to structures which contain a single daughter of type *non-simpl-dtr-struct*, where the daughter's semantics represents a doubly negated formula. If this is the case, the right-hand side of (25) percolates the matrix of this nested formula to the top level.

It is worth noting that our feature structure implications can *not* be interpreted as *rewrite rules* in the sense of term rewriting systems; however, they *encode* a rewrite rule through phrase structure trees. Real rewriting, instead, would violate the main assumption of the unification-based grammar paradigm, viz., *monotonicity*.

Our next example concerns one of *De Morgan's rules*, i.e.,

$$\frac{\neg(\phi \wedge \psi)}{\neg\phi \vee \neg\psi} \quad (26)$$

which corresponds to the following implication:

$$\begin{bmatrix} \textit{phrasal-sign} \\ \texttt{DTRS} \begin{bmatrix} \textit{non-simpl-dtr-struct} \\ \texttt{NON-SIMPL-DTR|SEM} \begin{bmatrix} \textit{op-sc-struct} \\ \texttt{OP}\ \neg \\ \texttt{SC|CONN}\ \wedge \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\textbf{(DeMorgan)}\quad \Longrightarrow \begin{bmatrix} \textit{phrasal-sign} \\ \texttt{SEM} \begin{bmatrix} \textit{conn-args-struct} \\ \texttt{CONN}\ \vee \\ \texttt{ARG1} \begin{bmatrix} \textit{op-sc-struct} \\ \texttt{OP}\ \neg \\ \texttt{SC}\ \boxed{1} \end{bmatrix} \\ \texttt{ARG2} \begin{bmatrix} \textit{op-sc-struct} \\ \texttt{OP}\ \neg \\ \texttt{SC}\ \boxed{2} \end{bmatrix} \end{bmatrix} \\ \texttt{DTRS|NON-SIMPL-DTR|SEM|SC} \begin{bmatrix} \texttt{ARG1}\ \boxed{1} \\ \texttt{ARG2}\ \boxed{2} \end{bmatrix} \end{bmatrix} \quad (27)$$

3.2 An Improved Version

The proposal presented so far has one significant disadvantage: extralinguistic principles can only be applied to top level forms which are licensed by the projection rule but can *not* be taken into consideration in the case of embedded structures, unless deeper reaching principles have been provided. While from a practical point of view, this may not be considered a severe drawback, it is unacceptable from the viewpoint of expressiveness.

Let us illustrate this claim with an example. Consider, for instance, the following derivation tree.

$$\begin{array}{ccc}
\boxed{2}\ [\texttt{SEM}\ (\neg\neg\psi) \wedge (\neg\neg\phi)] & & \\
\textbf{(DeMorgan)} \quad \Big| & & \\
\boxed{1}\ [\texttt{SEM}\ \neg(\neg\psi \vee \neg\phi)] & & \\
\Big| & & [\texttt{SEM}\ \neg\psi \vee \neg\phi] \\
 & \textbf{(DeMorgan)} & \Big| \\
[\,\ldots\,] & & [\texttt{SEM}\ \neg(\psi \wedge \phi)]
\end{array}$$

This example shows that everything works fine until De Morgan's rule is applied a second time. Given the structure of $\boxed{1}$,

$$\left[\begin{array}{l}\texttt{SEM}\ \left[\begin{array}{l}\textit{op-sc-struct}\\ \texttt{OP}\ \neg\\ \texttt{SC}\ \left[\begin{array}{l}\texttt{CONN}\ \vee\\ \texttt{ARG1}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \psi\end{array}\right]\\ \texttt{ARG2}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \phi\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right] \tag{28}$$

we can successfully apply (27), thus producing the following simplified semantics for $\boxed{2}$:

$$\left[\begin{array}{l}\texttt{SEM}\ \left[\begin{array}{l}\textit{conn-args-struct}\\ \texttt{CONN}\ \wedge\\ \texttt{ARG1}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \psi\end{array}\right]\end{array}\right]\\ \texttt{ARG2}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \left[\begin{array}{l}\texttt{OP}\ \neg\\ \texttt{SC}\ \phi\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right] \tag{29}$$

The problem now is that the schema for double negation stated in (25) cannot be applied to (29) because the structure under DTRS|NON-SIMPL-DTR|SEM would

be of type *conn-args-struct* after the application of (Project) but not of type *op-sc-struct*. Although the arguments of the connective $\wedge$ fulfill the antecedent of (25), the metalogical principle cannot fire. Note that this problem is not restricted to top level parts of the semantics of the immediate daughter but can arise at an arbitrary depth.

To overcome this shortcoming, we need the ability to *iterate* certain attributes/paths in the antecedent of an implication. The relevant attributes in example (29) are the arguments of the connective, `ARG1` and `ARG2`. Here however, the iteration is only of depth 1. If the feature logic allows us to specify *regular path expressions*, we are able to restate the antecedent of the principle for double negation in such a way that we can characterize doubly negated formulae at deeper levels; see (30). There exists a mechanism used primarily in the LFG community which fulfills exactly our needs: *functional uncertainty* [8] (note that Section 2.4 also makes use of this device). Functional uncertainty is a mechanism for dealing elegantly with linguistic phenomena like long distance dependencies or constituent coordination. With functional uncertainty, we can characterize a nested doubly negated formula at an arbitrary depth by the antecedent of (30). Because such a formula might occur under `ARG1` as well as under `ARG2`, the Kleene star * is applied to a disjunction `+` of these attributes; see (30).

Advocates of rewrite systems may question whether functional uncertainty is really called for here. They might propose simplification rules that can be applied anywhere within a feature structure as is known from rewrite systems. This, however, would assume a different semantics for feature structure implications—in order to *encode* the universal applicability of rewrite rules in term rewriting systems, functional uncertainty seems to be the only viable solution. The seeming disadvantage of specifying exactly the path where a matching structure must be located turns out to be a benefit: in our case, the specified path *guides the search* of an inference engine that, for a given principle, tests for the applicability of its antecedent. In the case of general rewrite systems, this search is not guided, i.e., the rewrite system is "blind" or must rely on heuristics.

Unfortunately, functional uncertainty is *not* sufficient to cope with structures embedded at deeper levels. This is because we must extract certain substructures under `DTRS`, which, however, should *not* be percolated entirely. Moreover, these structures might be specified by a *regular path*, since we do *not* know how deep they are located. Take, for instance, our example of double negation. What we would like to state is that the (top level) value of `SEM` is identical to the value under `DTRS|NON-SIMPL-DTR|SEM` with one important *exception*: the value under `DTRS|NON-SIMPL-DTR|SEM|(ARG1+ARG2)`* (the doubly negated formula) has to be *substituted* with `DTRS|NON-SIMPL-DTR|SEM|(ARG1+ARG2)`*`|SC|SC` (the matrix of the formula). This requires a special form of *monotonic substitution*. Since our notion of substitution is similar to the one used in the λ-calculus, we write $X_{\{Y \setminus Z\}}$ meaning:

Substitute in a copy of X every Y' with Z, where Y' is subsumed by Y.

The notion of a copy is defined as follows: X is a *copy* of X' iff $X \Rightarrow X'$ and $X' \Rightarrow X$, such that $X \neq X'$.

Functional uncertainty together with monotonic substitution now allows us to state an improved version of the principle for double negation, which subsumes (25).

$$\left[\begin{array}{l} \textit{phrasal-sign} \\ \texttt{DTRS} \left[\begin{array}{l} \textit{non-simpl-dtr-struct} \\ \texttt{NON-SIMPL-DTR|SEM } \boxed{1} \left[\begin{array}{l} \textit{conn-args-struct} \\ \texttt{(ARG1+ARG2)}^* \ \boxed{2} \left[\begin{array}{l} \textit{op-sc-struct} \\ \texttt{OP } \neg \\ \texttt{SC } \left[\begin{array}{l} \texttt{OP } \neg \\ \texttt{SC } \boxed{3} \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

$$\Longrightarrow \left[\begin{array}{l} \textit{phrasal-sign} \\ \texttt{SEM } \boxed{1}_{\{\boxed{2} \setminus \boxed{3}\}} \end{array}\right] \tag{30}$$

Coming back to our example, we are now able to simplify the value of SEM after the application of De Morgan's rule by using the improved principle for double negation. Note that (30) is applied to *both* arguments of the connective $\wedge$ in $\boxed{1}$. The derivation tree then looks as follows:

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \boxed{7}\ [\texttt{SEM } \psi \wedge \phi] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Big| \\ 2 \times \textbf{(2Neg)} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \boxed{1}\ [\texttt{SEM } (\neg\neg\psi) \wedge (\neg\neg\phi)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Big| \\ \textbf{(DeMorgan)} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \boxed{4}\ [\texttt{SEM } \neg(\neg\psi \vee \neg\phi)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Big| \\ \Big| \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\texttt{SEM } \neg\psi \vee \neg\phi] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{(DeMorgan)} \qquad \Big| \\ [\ldots] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\texttt{SEM } \neg(\psi \wedge \phi)] \end{array}$$

where

$$\boxed{1} = \begin{bmatrix} \textit{phrasal-sign} \\ \texttt{PHON}\ \boxed{2} \\ \texttt{SYN}\ \boxed{3} \\ \texttt{SEM} \begin{bmatrix} \textit{conn-args-struct} \\ \texttt{CONN}\ \boxed{8}\ \wedge \\ \texttt{ARG1} \begin{bmatrix} \texttt{OP}\ \neg \\ \texttt{SC} \begin{bmatrix} \texttt{OP}\ \neg \\ \texttt{SC}\ \boxed{5}\ \psi \end{bmatrix} \end{bmatrix} \\ \qquad\quad \texttt{ARG2} \begin{bmatrix} \texttt{OP}\ \neg \\ \texttt{SC} \begin{bmatrix} \texttt{OP}\ \neg \\ \texttt{SC}\ \boxed{6}\ \phi \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \texttt{DTRS|NON-SIMPL-DTR}\ \boxed{4} \begin{bmatrix} \texttt{PHON}\ \boxed{2} \\ \texttt{SYN}\ \boxed{3} \\ \texttt{SEM|SC} \begin{bmatrix} \texttt{ARG1}\ \boxed{5} \\ \texttt{ARG2}\ \boxed{6} \end{bmatrix} \end{bmatrix} \end{bmatrix} \tag{31}$$

and

$$\boxed{7} = \begin{bmatrix} \textit{phrasal-sign} \\ \texttt{PHON}\ \boxed{2} \\ \texttt{SYN}\ \boxed{3} \\ \texttt{SEM} \begin{bmatrix} \texttt{CONN}\ \boxed{8} \\ \texttt{ARG1}\ \boxed{5} \\ \texttt{ARG2}\ \boxed{6} \end{bmatrix} \\ \texttt{DTRS|NON-SIMPL-DTR}\ \boxed{1} \begin{bmatrix} \texttt{PHON}\ \boxed{2} \\ \texttt{SYN}\ \boxed{3} \\ \texttt{SEM} \begin{bmatrix} \texttt{CONN}\ \boxed{8} \\ \texttt{ARG1|SC|SC}\ \boxed{5} \\ \texttt{ARG2|SC|SC}\ \boxed{6} \end{bmatrix} \\ \texttt{DTRS|NON-SIMPL-DTR}\ \boxed{4} \end{bmatrix} \end{bmatrix} \tag{32}$$

If the principle of double negation should also be able to handle other cases of embedded constructions (quantifier within the scope of $\neg$, etc.), we must specify this as is the case for rewrite schemata in term rewriting systems. This can be achieved either by adding new principles for each case or, more generally, by making the improved version of (2Neg) sensitive to these special situations (cf. [13] for more details).

Our last extension concerns the introduction of *set values*. A truly robust, HPSG-inspired approach to logical form simplification must be able to unify the following two structures:

$$\begin{bmatrix} \textit{conn-args-struct} \\ \texttt{CONN}\ \wedge \\ \texttt{ARG1}\ \phi \\ \texttt{ARG2}\ \psi \end{bmatrix} \qquad \begin{bmatrix} \textit{conn-args-struct} \\ \texttt{CONN}\ \wedge \\ \texttt{ARG1}\ \psi \\ \texttt{ARG2}\ \phi \end{bmatrix} \tag{33}$$

Although $\phi \wedge \psi$ and $\psi \wedge \phi$ are equal in a model-theoretic sense (that is, the extensions are equal, i.e., denote the same set of objects), standard unification would fail. We, therefore, suggest to replace the keyword approach ARGn by a set-valued treatment as shown in (34). Moreover, this has the advantage of allowing more than two arguments for connectives like $\wedge$ or $\vee$ (see [19] for a similar proposal). In addition, there is no longer a need for specifying commutativity via a principle/schema; instead, commutativity is now handled internally through set unification.

$$\begin{bmatrix} \textit{conn-args-struct} \\ \texttt{CONN}\ \wedge \\ \texttt{ARGS}\ \{\phi, \psi, \ldots\} \end{bmatrix} \tag{34}$$

However, the question still remains which form of set values and set unification is really needed in our case (see for instance [22]). And perhaps, more important, what is the price we have to pay when using set-values. However, an examination of these aspects would exceed the scope of this paper.

4 Summary and Conclusions

In this paper, I have shown how FA can be neatly integrated and processed within TFF. The encoding method assumes that the logic makes recursive type definitions available. Some examples of German inflectional morphology [16] have been implemented in the typed feature formalism $\mathcal{TDL}$ [17].

The second area addressed in this paper concerns a proposal for logical form simplification within TFF/HPSG. The approach makes strong assumptions about the expressivity of the feature calculus (set values, functional uncertainty/recursive types and monotonic substitution).

Both approaches extend the domain of "ordinary" constraint-based grammars beyond the construction of syntax and semantics, thus avoiding artificial interface problems between different components in that everything is represented within the same formalism. This integration need *not* lead to a heavy decrease of efficiency as explained in Section 2 and 3, so that the advantages of these proposals prevail against non-integrated, multi-component oriented systems.

References

1. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
2. Hiyan Alshawi, editor. *The Core Language Engine*. ACL-MIT Press Series in Natural Language Processing. MIT Press, 1992.
3. Steven Bird. Finite-state phonology in HPSG. In *Proceedings of the 14th International Conference on Computational Linguistics, COLING-92*, pages 74–80, 1992.
4. Bob Carpenter. *The Logic of Typed Feature Structures*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

5. Jochen Dörre and Andreas Eisele. Determining consistency of feature terms with distributed disjunctions. In Dieter Metzing, editor, *Proceedings of 13th German Workshop on Artificial Intelligence, GWAI-89*, pages 270–279, Berlin, 1989. Springer.
6. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.
7. Mark Johnson. *Attribute Value Logic and the Theory of Grammar*. CSLI Lecture Notes, Number 16. Center for the Study of Language and Information, Stanford, 1988.
8. Ronald M. Kaplan and John T. Maxwell III. An algorithm for functional uncertainty. In *Proceedings of the 12th International Conference on Computational Linguistics, COLING-88*, pages 297–302, 1988.
9. Robert T. Kasper and William C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 257–266, 1986.
10. Martin Kay, Jean Mark Gawron, and Peter Norvig. *Verbmobil: A Translation System for Face-to-Face Dialog*. CSLI Lecture Notes, Number 33. Center for the Study of Language and Information, Stanford, 1994.
11. Kimmo Koskenniemi. Two-level model for morphological analysis. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 683–685, 1983.
12. Hans-Ulrich Krieger. Derivation without lexical rules. In C.J. Rupp, M.A. Rosner, and R.L. Johnson, editors, *Constraints, Language and Computation*, pages 277–313. Academic Press, 1994. A version of this paper is available as DFKI Research Report RR-93-27. Also published in IDSIA Working Paper No. 5, Lugano, November 1991.
13. Hans-Ulrich Krieger. Logical form simplification within HPSG. Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1995. Forthcoming.
14. Hans-Ulrich Krieger. Representing and processing finite automata within typed feature formalisms. Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany, 1995. Forthcoming.
15. Hans-Ulrich Krieger and John Nerbonne. Feature-based inheritance networks for computational lexicons. In Ted Briscoe, Valeria de Paiva, and Ann Copestake, editors, *Inheritance, Defaults, and the Lexicon*, pages 90–136. Cambridge University Press, New York, 1993. A version of this paper is available as DFKI Research Report RR-91-31. Also published in Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon, Technical Report No. 238, University of Cambridge, Computer Laboratory, October 1991.
16. Hans-Ulrich Krieger, John Nerbonne, and Hannes Pirker. Feature-based allomorphy. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pages 140–147, 1993. A version of this paper is available as DFKI Research Report RR-93-28.
17. Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94, Kyoto, Japan*, pages 893–899, 1994. An extended version of this paper is available as DFKI Research Report RR-94-37.
18. Joachim Laubsch and John Nerbonne. An overview of $\mathcal{NLL}$. Technical report, Hewlett-Packard, 1991.

19. John Nerbonne. A feature-based syntax/semantics interface. In Alexis Manaster-Ramer and Wlodek Zadrozny, editors, *Mathematics of Language, Vol. 2*. Annals of Artificial Intelligence and Mathematics, 1992. Also available as DFKI Research Report RR-92-42.
20. Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol. I: Fundamentals*. CSLI Lecture Notes, Number 13. Center for the Study of Language and Information, Stanford, 1987.
21. Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press, Chicago, 1994.
22. William C. Rounds. Set values for unification-based grammar formalisms and logic programming. Technical Report CSLI-88-129, Center for the Study of Language and Information, 1988.
23. Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes, Number 4. Center for the Study of Language and Information, Stanford, 1986.
24. Gert Smolka. A feature logic with subsorts. LILOG Report 33, WT LILOG–IBM Germany, Stuttgart, May 1988. Also in J. Wedekind and C. Rohrer (eds.), Unification in Grammar, MIT Press, 1991.
25. Hans Uszkoreit. From feature bundles to abstract data types: New directions in the representation and processing of linguistic knowledge. In A. Blaser, editor, *Natural Language at the Computer—Contributions to Syntax and Semantics for Text Processing and Man-Machine Translation*, pages 31–64. Springer, Berlin, 1988.
26. Hans Uszkoreit. Linear precedence in head domains. Paper presented at the *HPSG in German* workshop, 1992.
27. Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO—an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of COLING-94, Kyoto, Japan*, 1994. A version of this paper is available as DFKI Research Report RR-94-38.
28. Rémi Zajac. A transfer model using a typed feature structure rewriting system with inheritance. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 1–6, 1989.

This article was processed using the LaTeX macro package with LLNCS style