A Proof-Theoretic Reconstruction of HPSG*

Stephan Raaijmakers

Institute for Language Technology and Artificial Intelligence ITK
P.O.Box 90153, 5000 LE Tilburg, The Netherlands
email: stephan@kub.nl

Abstract

A reinterpretation of Head-Driven Phrase Structure Grammar (HPSG) in a proof-theoretic context is presented. This approach yields a *decision procedure* which can be used to establish whether certain strings are generated by a given HPSG grammar. It is possible to view HPSG as a fragment of linear logic (Girard, 1987), subject to partiality and side conditions on inference rules. This relates HPSG to several categorial logics (Morrill, 1990). Specifically, HPSG signs are mapped onto quantified formulae, which can be interpreted as *second-order* types given the Curry-Howard isomorphism. The logic behind type inference will, aside from the usual quantifier introduction and elimination rules, consist of a partial logic for the undirected implication connective.

It will be shown how this logical perspective can be turned into a parsing perspective.

The enterprise takes the standard HPSG of Pollard — Sag (1987) as a starting point, since this version of HPSG is well-documented and has been around long enough to have displayed both merits and shortcomings; the approach is directly applicable to more recent versions of HPSG, however. In order to make the proof-theoretic recasting smooth, standard HPSG is reformulated in a binary format.

1 Introduction

The main concern of this paper lies in building a parser for HPSG. The result of the enterprise should meet the following desiderata:

- The parser should interpret the original grammatical theory, or as close a dialect as possible.
- The parser should separate grammatical theory from parsing issues.
- The parser should make an operationalisation of the grammatical theory explicit, as declaratively as possible.
- It should be easy to alter the grammatical theory.
- The parser should have reasonable time/space complexity.

Existing parsers for HPSG do not obey these demands; e.g., the Popowich/Vogel parser (Popowich — Vogel, 1990) violates the second, fourth and fifth demand; the LiLog STUF environment (Dörre — Raasch, 1991) violates the first, third, and fifth. For a full comparison, see Raaijmakers (forthcoming).

In the *parsing-as-deduction* field, several parsing routines have arisen from proof-theoretic investigations (Moortgat, 1988; König, 1989). While these routines are not all among the most efficient, once a proof-theoretic formulation of HPSG has been made, one can benefit from these results.

*This research was carried out within the framework of the research programme 'Human-Computer Communication using natural language' (MMC). The MMC programme is sponsored by Senter, Digital Equipment B.V., SUN Microsystems Nederland B.V. and AND Software.

Some terminological remarks: we refer to HPSG of Pollard — Sag (1987) with '(classical) HPSG', and to its type-theoretic (deductive) equivalent with '$\mathcal{D}$-HPSG'.

2 An overview of HPSG

HPSG is a lexicalist, feature-based formalism for syntactic and semantic analysis of natural language. HPSG puts all relevant linguistic information in the lexicon, and has general rules and principles governing the construction of phrases from subphrases.

As a syntactic formalism, HPSG divides the labour of tree construction into separate processes of *mobile construction* and *mobile ordering*. A mobile is a tree-like structure with unordered trees; actually, a mobile can be interpreted as a description of a set of trees.

Socalled *immediate dominance* (ID) rules build these mobiles, which are then turned into trees by *linear precedence* (LP) principles. HPSG is a feature-based formalism, employing various feature mechanisms transporting feature information through feature structures. In HPSG, lexemes are bundles of so-called attribute-value pairs

$$\begin{bmatrix} A_i & V_i \\ \vdots & \vdots \\ A_n & V_n \end{bmatrix}$$

where A_j is a certain linguistic (phonological/syntactic/semantic) property taking its specification from a set of values containing V_j. These bundles are called *signs*. The reader is referred to Pollard — Sag (1987,1992) for a full overview of the various attributes and their values. The generic structure of main signs in HPSG is

$$\begin{bmatrix} \texttt{phon} & \dots \\ \texttt{syn} & \dots \\ \texttt{sem} & \dots \\ \texttt{dtrs} & \dots \end{bmatrix}$$

where the `phon`, `syn`, `sem` and `dtrs` values describe respectively the phonological, syntactic, semantic and configurational properties of the sign.

Attributes take either atomic or complex values; an attribute like `person` ranges over the set {`first`, `second`, `third`}, whereas an attribute like `dtrs` (describing daughters of phrases) takes full signs as values. The notion of *head* is a central concept in HPSG. Basically, a head of a phrase is a subphrase determining the relevant combinatorial properties of the sign. Heads can be phrasal or lexical; lexical heads are simply signs having no daughters. For instance, the head of a VP *sees Mary* is the verb *sees*; The grammatical properties of *sees* determine the properties (viz. agreement) of the VP as a whole, and not those of the direct object *Mary*. The head of the sentence *John sees Mary* is the VP *sees Mary*.

HPSG heavily leans on the notion of *unification* (Shieber, 1986). Simplifying matters somewhat, two signs S_1 and S_2 are unifiable with each other, written $S_1 \sqcup S_2$, if for any attribute they are both specified for, they bear non-conflicting values. Further, any fully disjunct parts of two signs (consisting of different attribute-value pairs) of the two signs can be combined directly. So,

$$\begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{case} & \texttt{dative} \end{bmatrix} = \begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \\ \texttt{gender} & \texttt{fem} \\ \texttt{case} & \texttt{dative} \end{bmatrix}$$

Likewise,

$$\begin{bmatrix} \texttt{syn}\,|\,\texttt{loc}\,|\,\texttt{head}\,|\,\texttt{maj} & \texttt{n} \\ \texttt{agr} & \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{number} & \texttt{sg} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{person} & 2 \end{bmatrix} = \begin{bmatrix} \texttt{syn}\,|\,\texttt{loc}\,|\,\texttt{head}\,|\,\texttt{maj} & \texttt{n} \\ \texttt{agr} & \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{number} & \texttt{sg} \\ \texttt{person} & 2 \end{bmatrix} \end{bmatrix}$$

But of course the following fails:

$$\begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{number} & \texttt{plur} \end{bmatrix}$$

2.1 Immediate dominance (ID) rules

ID rules describe admissible dominance structures, which can be interpreted as mobiles: tree-like structures with unordered branches. The

rules themselves take the form of (partially specified) signs (just like HPSG's principles), applying as felicity constraints to signs to be combined.

Rule 1 (**R1**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle\rangle \\ \texttt{dtrs}\ \left[\begin{array}{l}\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{lex}\ - \\ \texttt{compdtrs}\ \langle \texttt{\_d}\rangle\end{array}\right]\end{array}\right]$$

Rule 2 (**R2**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle \texttt{\_s}\rangle \\ \texttt{dtrs}\ \left[\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc}\left[\begin{array}{l}\texttt{head} \mid \texttt{inv}\ - \\ \texttt{lex}\ +\end{array}\right]\right]\end{array}\right]$$

Rule 3 (**R3**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle\rangle \\ \texttt{dtrs}\ \left[\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc}\left[\begin{array}{l}\texttt{head} \mid \texttt{inv}\ + \\ \texttt{lex}\ +\end{array}\right]\right]\end{array}\right]$$

Rule 1 licenses signs having a non-lexical (i.e. phrasal) head daughter and being fully saturated, that is, signs having a subcat(egorisation) list of length zero (written as $\langle\rangle$). There is one complement daughter (indicated with the variable '_d'); an example would be an S having a VP as head daughter and a subject as only complement daughter. The `loc` attribute is used to describe "local" properties of a sign, such as lexicality and subcategorisation demands; this contrasts with the `bind` attribute, describing anaphoric links over signs.

Rule 2 caters for instance for VP's, which have a lexical head daughter (the verb), and are one short of becoming saturated: they subcategorise for a subject.

Rule 3 admits of saturated signs with a lexical, inverted head daughter, like in *Is John sleeping?*, the head daughter of which is the finite auxiliary *Is*, which subcategorises for both an infinitival VP and a nominative NP.

2.2 Linear Precedence (LP) principles

LP principles turn mobiles into genuine trees by imposing order on sister nodes.

Constituent Order Principle

$$\left[\begin{array}{l}\texttt{phon}\ \texttt{order} - \texttt{constituents}(\boxed{1}) \\ \texttt{dtrs}\ \boxed{1}\end{array}\right]$$

Linear Precedence Constraint 1 (**LP1**)
`head[lex +] < [ ]`

Linear Precedence Constraint 2 (**LP2**)
`complement << complement[lex -]`

The operation `order-constituents` gives the disjunction of all permutations of the phonology of the daughters. At least one of these permutations will have to be consistent with the constraints of order expressed by the LP principles, which are specific for English (and related languages):

LP1 says that lexical heads precede all their sisters (the empty sign [] acts like a "wildcard" symbol here, unifying with every sign). LP2 says that less oblique complements precede more oblique phrasal sisters; << is precedence between oblique elements, where obliqueness corresponds inversely to degree of obligatoriness. Subjects, for instance, are in Germanic languages less oblique than direct objects, which means they are more obligatory: they cannot be omitted, in general.

John eats.

*eats an apple.

The degree of obliqueness is mirrored (in reverse) by the order of complements on the `subcat` list of signs: the less oblique elements follow the more oblique elements.

2.3 Feature transport principles

Various principles take care of the distribution of feature information in a feature structure, defining the paths along which information percolates upwards. The concept of *reentrancy* expresses the sharing of information by several attributes across a sign, using boxed integers to identify attributes.

Head Feature Principle

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{head}\ \boxed{1} \\ \texttt{dtrs} \mid \texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{head}\ \boxed{1}\end{array}\right]$$

Subcat(egorisation) Principle

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \boxed{2} \\ \texttt{dtrs}\ \left[\begin{array}{l}\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \boxed{1} + \boxed{2} \\ \texttt{compdtrs}\ \boxed{1}\end{array}\right]\end{array}\right]$$

(L1+L2 is the concatenation of the two lists L1 and L2.)

Semantics principle (simplified)

$$\left[\begin{array}{l}\textbf{sem}\ \left[\ \textbf{cont}\ s-c-s(\boxed{1},\boxed{2})\ \right] \\ \textbf{dtrs}\ \left[\begin{array}{l}\textbf{headdtr}\mid\textbf{sem}\mid\textbf{cont}\ \boxed{1} \\ \textbf{compdtrs}\mid\textbf{sem}\mid\textbf{cont}\ \boxed{2}\end{array}\right]\end{array}\right]$$

The HFP enforces identity between the head features of the head daughter and the mother sign.

The subcat principle decomposes the subcat list of a head daughter in two parts of arbitrary (non-empty) length: the first part should correspond to the value of `compdtrs`, the second part becomes the value of the `subcat` attribute of the mother sign.

The semantics principle states that the semantics of a mother sign must consist of the combination of the semantics of the head daughter and the complement daughters (the operation `successively-combine-semantics`, abbreviated as `s-c-s`, does just this.)

2.4 Sample derivation

The derivation in figure 1 shows the operation of the various principles and rules. Notice that HPSG as presented in Pollard — Sag (1987) assumes that nouns are heads selecting for determiners. Also, Pollard and Sag assume that heads themselves participate in the obliqueness hierarchy: they are more oblique than all their complements. Thus, LP2, once formulated as

`complement << complement[lex -]`

Pollard — Sag (1987:176) orders phrasal heads after their complements (for instance VP's after subjects). In order to derive the order 'the cat' rather than 'cat the', Pollard and Sag assume that the head noun 'cat' becomes phrasal by the fact that Rule 2 is applicable to it as a lexical sign (cf. op.cit. p.153); so, rule application turns `[lex +]` into `[lex -]`.

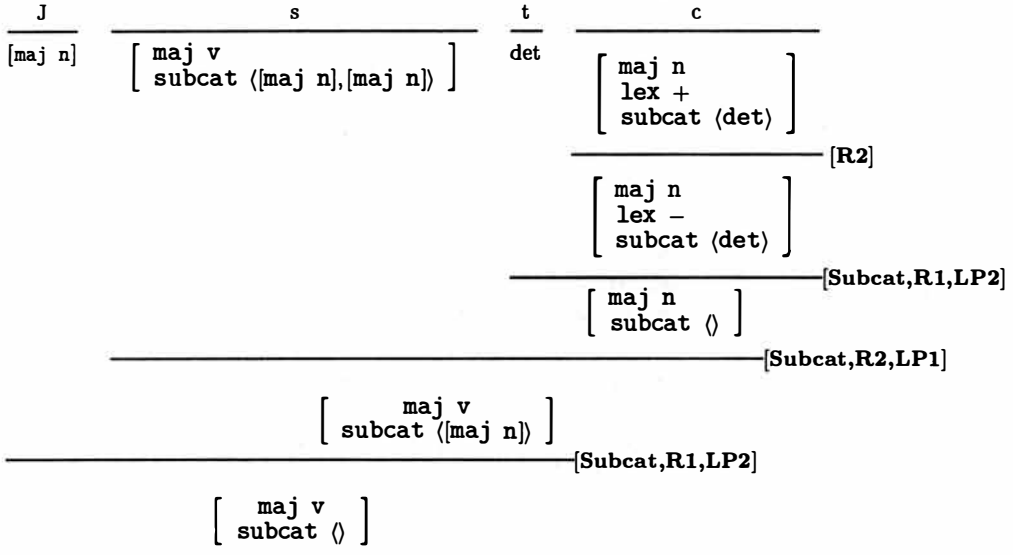


Figure 1: John sees the cat.

2.5 Tree arity

Classical HPSG is not strictly binary-branching: the Subcat principle allows for the combination of an *n*-ary functor with *n-1* arguments at once; e.g. the combination of a ditransitive verb taking three NPs (two complements and one subject) with two of its complements.

Classical HPSG is *non-monotonous*: as mentioned in the previous section, Rule 2 is able to change the lexicality of a sign by vacuously applying to that sign. This underdocumented feature of HPSG has several drawbacks, most significantly, the fact that the operation is not structure preserving: usually, signs evolve from subsigns under unification; here, a + value becomes a − value, which unification cannot possibly account

for. Conceptually, the lexicality feature seems to be *derivable*, since, only those signs are non-lexical (phrasal) which carry at least one daughter.

The HPSG theory as originally put forward by Pollard — Sag (1987) does not lend itself directly to a proof-theoretic reconstruction. The theory, being declarative in a strong sense, has obscure operational aspects. Also, mainly for practical (but possibly also for theoretical) reasons, it appears to be desirable to have a version of HPSG building binary branching syntax trees. So, as a first step we present a binary version of HPSG.

3 Binary-Branching HPSG

We start off by presenting a binary version of HPSG which removes some of the unattractive features of classical HPSG. Most significantly, this version makes no use of vacuous application of rules to signs, and thus allows for signs to monotonously evolve from lexical to non-lexical status. The theory remains very close to classical HPSG in all other aspects. The binarity is mainly motivated from practical reasons; it facilitates the linking of HPSG to a logical type calculus. Binarity is by no means a strong commitment, however. Focus is on the desire to analyse a fragment of Dutch declarative main clauses, although some examples illustrate the applicability of the binary apparatus on fragments of English as well.

First, we define lexicality in terms of daughters, using common predicate notation.

- `lexical`(Sign) if `dtrs`(Sign)= $\langle\rangle$, parafrased as: Sign is lexical if Sign has zero daughters.

We then define:

- $\texttt{args}_n$(Sign) if `length`(`subcat`(Sign))= $n, n \geq 1$, parafrased as: Sign wants n arguments if the subcat list of Sign has length n (an empty list has length zero).

We refer to a functor $\mathcal{F}$ with $\texttt{args}_n$ as $\mathcal{F}_n$.

The crucial observation for languages like Dutch and English is that the amount of saturation together with the lexicality of a functor (a sign with non-empty subcat list) determines the position of the functor with respect to its argument. A post-modifier like *with pictures*, modifying a noun like *book*, follows the noun: it is non-lexical, and has $\texttt{args}_1$. Similarly, intransitive verbs — assuming they are lexicalised as VP's, i.e. non-lexical, verbal $\texttt{args}_1$ functors — follow their subjects. Semi-saturated verbal functors like *gives John* precede their objects: they are $\texttt{args}_2$ functors. We can capture the order determiner-noun by assuming that determiners subcategorise for non-maximal noun projections (like *book*, *little book with black cover*), so they are $\texttt{args}_1$; they are lexical, and precede their argument. This contrasts with the view of Pollard — Sag (1987), which analyses nouns as subcategorising for determiners.[1] So, the generalisation seems to be that:

1. Ordering effects triggered by the lexicality of functors come into play only for $\mathcal{F}_1$ functors: a lexical $\mathcal{F}_1$ is ordered before its argument; a non-lexical $\mathcal{F}_1$ is ordered after its argument.
2. A functor $\mathcal{F}_n$ where $n > 1$ is ordered before its argument.

The following LP principles make this precise:[2]

(BLP1)	$\left[\begin{array}{ll}\texttt{lex} & +\\ \texttt{args}_1 & \end{array}\right] < \alpha$
(BLP2)	$\alpha < \left[\begin{array}{ll}\texttt{lex} & -\\ \texttt{args}_1 & \end{array}\right]$
(BLP3)	$[\ \texttt{args}_n\ ] < \alpha$

To see how these principles work, consider the derivation in figure 2.

[1] We shall neglect the question on how to encode (non)-maximality of phrases here; a bar-level along the lines Cooper (1990) suggests may be necessary here.

[2] The signs in these principles are only partly specified.

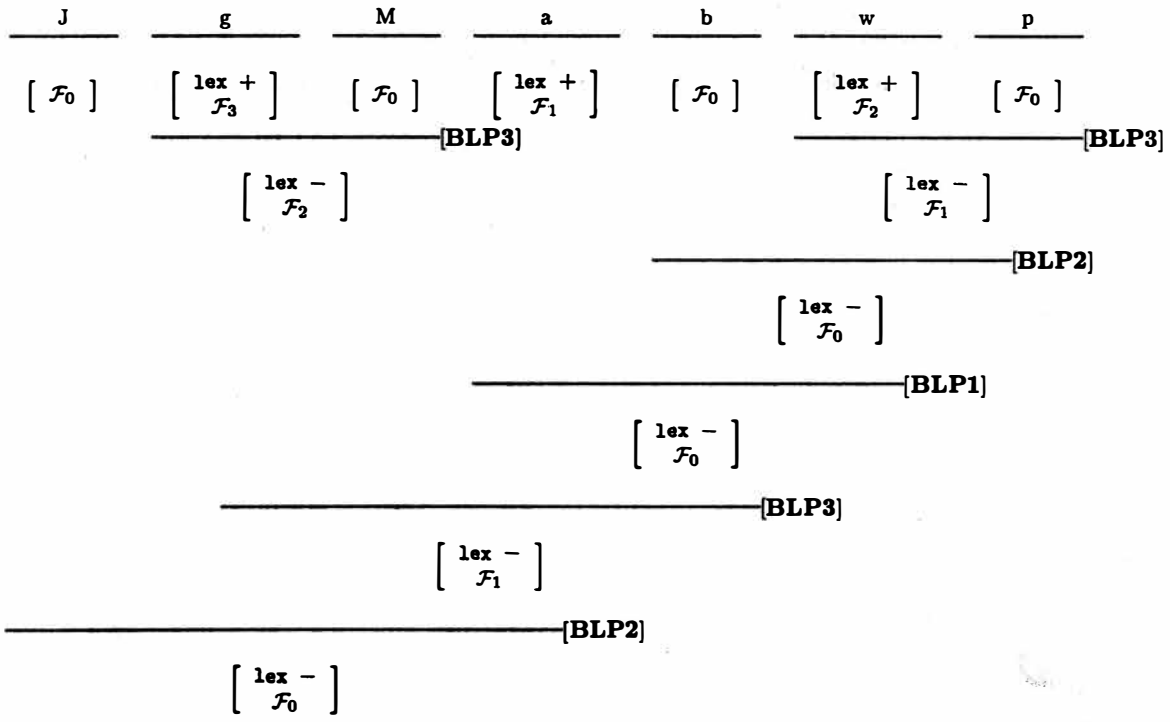


Figure 2: Sample derivation for 'John gives Mary a book with pictures'.

We also need the regular LP principle for inverted phrases:

(BLP4)	[inv +] < α

This all works fine for concatenative phenomena, i.e. the combination of two phrases under adjacency. Certain adjuncts appear to be non-concatenative, however. In Dutch, one observes:

- Jan geeft met plezier Marie een boek.
  John gives eagerly Mary a book.
- Jan geeft Marie met plezier een boek.
- Jan geeft Marie een boek met plezier.

This suggests that the phonological operation associated with certain adverbial modifiers should not be concatenation but *infixation*. Reape (1990) has made similar remarks concerning semi-free word order phenomena. We then arrive at the following LP principle

(BLP5)	[ADVMOD] ⇓ α

where $A \Downarrow B$ says that (the phonology of) A is infixed into (the phonology of) B. The non-concatenative connective $\downarrow$ was introduced in categorial grammar by Moortgat (Moortgat, 1988) for similar purposes; an expression of type $A \downarrow B$ infixes into expressions of type B to form an expression of type A (see section 4.4.2 below). ADVMOD describes the sign for a VP-level adverbial modifier, which is a sign subcategorising for a VP to yield a VP: it inherits the NP argument (the subject) its argument VP is still incomplete for. There is a little snag here: mere infixation of the adverbial phonology into the VP phonology would result in ill-formed strings where the adverb penetrates into one of the verbal arguments. For instance,

*Jan geeft de graag man een boek
John gives the with-pleasure man a book

This problem cannot be fixed by letting

phonology-values be nested lists (lists of lists) rather than *flat* lists, for instance

[Jan,[[geeft,[de,man]],[een,boek]]]

The infixation of *graag* into the VP phonology [[geeft,[de,man]],[een,boek]] will be possible only for

- graag geeft de man een boek
- geeft de man graag een boek
- geeft de man een boek graag

Deriving the well-formed

- Jan geeft graag de man een boek

now becomes hard: a rebracketing of

[[geeft,[de,man]],[een,boek]]

to

[[geeft],[[de,man],[een,boek]]]

will be necessary. So, it is not entirely clear whether the phonological operation of adverbial modifiers is not beyond simple infixation. For the moment, we leave the topic.

Complement order needs no longer be stipulated as a separate LP principle: functors now combine with one argument at a time, and the order of arguments is expressed by the order on the Subcat list.

The ID rules of original HPSG must be adapted as well; while Rules 1 and 3 can be kept, Rule 2 must now be altered to cater for generalised incompleteness: a sign having more than one item on its subcategorisation list is a well-formed sign as well.

4 Deduction for HPSG

With the binary version of HPSG we are set to give HPSG a deductive basis. First, we show that it is possible to reinterpret signs as types. Then we introduce a deductive apparatus performing type-deduction with these derived types. This calculus builds binary proof trees (*proof terms*), which are orthogonal to (binary-) HPSG derivation trees.

4.1 Signs as Formulae

We propose to view signs as *types*, or, with the Curry-Howard isomorphism in mind, as *formulae* of a certain logic. Ideas in this spirit can already be found in work of Blackburn (interpreting signs as modal formulae), Morrill and others. The concept of types has many interpretations, but one particularly apt for linguistics is that a type is a set of expressions, or, in more traditional terms, a category. Together with a set of combinatorial principles, types form an algebra of expressions over a certain domain: a type system. Essentially, these combinatorial principles constitute a *derivability relation* between sequences of types '$\rightarrow$': $A \rightarrow B$ saying that from the type sequence A the type sequence B can be derived. An example of a type system would be any syntactic algebra consisting of a set of type formation rules (e.g. the production rules in a rewrite system) and a set of syntactic categories (types) containing expressions over some alphabet of strings. More fine-grained type systems make a distinction between atomic and complex types: atomic types being monadic objects and complex types being made up from (atomic or complex) subtypes with the use of so-called type-forming connectives which serve to express combinatorial properties. Type-forming connectives are relations over the set of type symbols; a familiar example are the slashes from categorial grammar $/, \backslash$: a *functor* type X/Y combines with a type Y to its *right* to form a type X; a functor type $Y\backslash X$ combines with a type Y to its *left* to form an X. There is a nice interpretation of linguistic types as propositional formulae in a logic: atomic types T correspond to formulae T; complex types like $A\backslash B$ correspond to $A \Rightarrow_l B$, with $\Rightarrow_l$ a left-oriented version of the implication arrow $\Rightarrow$ of propositional logic. The combination of a type A with a type $A \Rightarrow B$ to a type B then becomes an instance of Modus Ponens, of which we now have two versions: $A, A \Rightarrow_l B \rightarrow B$ and $A \Rightarrow_r B, A \rightarrow B$. This, in fact, is an operationalisation of the slogan *parsing as deduction*, and is basically the central theme of categorial deduction as in Lambek calculus (Moortgat, 1988).

The intuition that signs can be interpreted as types arises from the functionality expressed by the `subcat` feature: essentially, this feature

expresses that a certain sign is *functionally (in-)complete* for one or more other signs. This immediately suggests a functional type equivalent (a functor) for these signs. Saturated signs then can be interpreted to correspond to saturated functors, or atomic types, i.e. types not being made up from a type-forming connective and one or more subtypes. HPSG's Subcat principle, which allows for the combination of a non-saturated sign with a subset of the signs it subcategorises for should then correspond to a combinatorial rule of type formation, i.e. an inference rule in a type calculus.

When we want to make a correspondence between the signs of HPSG and types of a certain kind, we immediately notice that HPSG signs encode much more information than the monadic categories of simple type systems like production grammars. A category like S, for instance, is represented in HPSG as a fine-grained specification of a verbal projection having various properties among which is an empty subcategorisation frame. Clearly, we need a more sophisticated type language than can be offered by monadic categories alone. Suppose then we switch from monadic types to types with internal structure: *predicational* types in stead of propositional types. The value of the category-determining `maj(or)` attribute should become the top-level predicate constant. As in predicate logic, types (propositions) are made up from such a predicate constant and terms as arguments of the predicate. Signs having variable values, i.e. being *underspecified* for certain attributes, correspond to (universally) quantified formulae. E.g., a partial sign like

$$\begin{bmatrix} \texttt{major} & \texttt{n} \\ \texttt{gender} & \texttt{neuter} \\ \texttt{person} & \_x \end{bmatrix}$$

with $\_x$ a variable should correspond to the type

$$\forall(\_x).[n(gender(neuter), person(\_x))].$$

The choice between universal and existential quantification is mainly motivated from considerations regarding the proof terms for quantified formulae, which will be discussed in the next section. A related motivaton is the fact that universally quantified types have a straightforward connection with Prolog literals, facilitating implementation. It is important to notice that there is no deep, 'predicate-like' meaning behind such a formula: it is just a description of a certain kind of category, in the case above having a variable spot for the person value. Sign-valued attributes, i.e. attributes taking a full sign as value, or a list of signs, are treated the same: whenever such an attribute takes a variable sign as value, universal quantification over this variable occurs. This is responsible for the second-order nature of the type language we use.

Under the logical interpretation of types as formulae, types have *proof terms* associated with them; these proof terms are the *justification* for assuming the formula is true: they correspond to proofs for the propositions the types express. These proofs are constructed in a *calculus* of inference rules, the inference rules constituting a derivability relation over type sequences (like the combinatorial rules of production systems), where this derivability relation now gets a logical interpretation as well. An alternative, quite common point of view is that proof terms are a kind of procedures (or programs) and types are the specification of what these programs do. For instance, the formula

$$\forall(\_x).[n(gender(masc), number(\_x))]$$

would be a specification of the program recognising singular and plural masculine noun phrases (this basically is what parsing is about).

A concept like *reentrancy* can easily be encoded by means of variable sharing, for example

$$\forall(\_x).[P_1(P_i(\_x), \ldots, P_n(\_x))]$$

where each P_j is a predicate symbol.

We now turn to the translation from signs to types, where we let $\dagger(S)$ yield the formula (type) equivalent of the sign S. A few words on notation: $\vec{Q}$ denotes a sequence $\forall(\_x_i)\ldots\forall(\_x_n)$ of quantifiers. The empty quantifier sequence is written as $\mathcal{Q}_0$; $\mathcal{Q}_0.\mathcal{F} = \mathcal{F}$. Further, $\vec{Q}\mathcal{Q}_0 = \mathcal{Q}_0\vec{Q} = \vec{Q}$. We use the notation

$$\Sigma \begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

to refer to some sign Σ with the sign

$$\begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

as a subsign. Likewise,

$$\Sigma - \begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

refers to some sign Σ with the sign

$$\begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

deleted from it. Furthermore, $\mathtt{var}(X)$, $\mathtt{atom}(X)$, $\mathtt{number}(X)$ express respectively that X is a variable, an atom or a number.

- $\dagger(X) := \mathcal{Q}_0.X$
  if $\mathtt{var}(X)$ **or** $\mathtt{atom}(X)$ **or** $\mathtt{number}(X)$ **or** $X = \langle\rangle$
- $\dagger(\Sigma \begin{bmatrix} \mathtt{subcat}\ \langle\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}.M(F)$
  if $\dagger(\Sigma - \begin{bmatrix} \mathtt{subcat}\ \langle\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}.F$
- $\dagger(\Sigma \begin{bmatrix} \mathtt{subcat}\ \langle X_1, ..., X_n\rangle \\ \mathtt{maj}\ M \end{bmatrix}) :=$
  $\vec{\mathcal{Q}}_1\vec{\mathcal{Q}}_2.A_1 \Rightarrow \ldots A_n \Rightarrow M(F)$
  if $\dagger(\Sigma - \begin{bmatrix} \mathtt{subcat}\ \langle X_1, ..., X_n\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}_1.F$
  and $\dagger(\langle X_1, ..., X_n\rangle) := \vec{\mathcal{Q}}_2.\langle A_1, \ldots, A_n\rangle$
- $\dagger([\ A\ V\ ]) := \forall(V).A(V)$ **if** $\mathtt{var}(V)$
- $\dagger([\ A\ V\ ]) := \vec{\mathcal{Q}}.A(X_1, \ldots, X_n)$
  if $\dagger(V) := \vec{\mathcal{Q}}.\langle X_1, \ldots, X_n\rangle$
- $\dagger(\langle X_1, \ldots, X_n\rangle) := \vec{\mathcal{Q}}_1\vec{\mathcal{Q}}_2.\langle F_1, F_2, \ldots, F_n\rangle$
  if $\dagger(X_1) := \vec{\mathcal{Q}}_1.F1$ **and**
  $\dagger(\langle X_2, \ldots, X_n\rangle) := \vec{\mathcal{Q}}_2.\langle F_2, \ldots, F_n\rangle$

The crucial thing to note is that the `subcat` information of a sign is reformulated as the functional demands of a functor type: a `subcat` list of length n yields a functor with functional degree n, where n now indicates the number of arguments the functor is incomplete for.

The following example illustrates the mapping from signs to formulae. Variables are prefixed with a *don't care* '_'.

```
[syn,[[loc,[[head,[[maj,n],
                  [case,_c],
                  [nform,_n],
                  [aux,nil],
                  [inv,nil],
                  [prd,nil]]],
           [subcat,[]],
           [lex,1]]],
     [bind,_b]]]
```

then becomes

$$\forall(\_c)\forall(\_n)\forall(\_b).$$
$$[n(syn(loc(head(case(\_c), nform(\_n), aux(nil),$$
$$inv(nil), prd(nil)), lex(1)), bind(\_b)))]$$

4.2 Type deduction

Now that we have types, the question arises: what do we do with these types? In this section we show how we can interpret the HPSG apparatus of ID rules and various principles as an inference mechanism for type deduction. Before we do so, a few words on type deduction are necessary.

As mentioned in section 4.1, types have a truth-conditional interpretation: they correspond to *propositions* (formulae). This logical point of view makes it possible to identify type derivability relations with *logical* derivability relations from proof theory. A statement $A \rightarrow B$ expressing the derivability of type sequence B from type sequence A is then called a *sequent* (Gallier, 1986). A sequent $A_1, ..., A_n \rightarrow B$ can be interpreted as: the validity of the formulae $A_1, ..., A_n$ implies the validity of B; i.e., there is no model for the formulae $A_1, ..., A_n$ that is not also a model for B. The sequence $A_1, ..., A_n$ is called the *antecedent* of the sequent; the sequence B (in the present case of length 1) is called the *succedent* of the sequent. The configuration $\frac{P_1 \quad \ldots \quad P_n}{C}$ is read as: the *conclusion sequent* C is valid iff the *premise sequents* $P_1, \ldots, P_n$ are valid. As an example, here is a fragment of so-called *linear non-commutative propositional logic*. 'Linear' (Girard, 1987) means here that this logic forces 'honest' bookkeeping: we are not allowed to duplicate nor delete types during derivation. From a linguistic point of view, linearity can be used to express the fact that

the meaning of an utterance depends on the linear order of its words. Every Γ_i is a (possibly empty) type sequence; Λ is a non-empty type sequence, and X, Y, Δ are types. The comma ',' denotes non-commutative concatenation: Γ_1, Γ_2 is the concatenation of the type sequences Γ_1 and Γ_2. This entails that antecedents are essentially lists of types.

$$\mathcal{I}\frac{}{X \to X}$$

$$\mathcal{R}\Rightarrow\frac{\Gamma, X \to Y}{\Gamma \to X \Rightarrow Y}$$

$$\mathcal{R}\Rightarrow\frac{X, \Gamma \to Y}{\Gamma \to X \Rightarrow Y}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to X \quad \Gamma_1, Y, \Gamma_2 \to \Delta}{\Gamma_1, X \Rightarrow Y, \Lambda, \Gamma_2 \to \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to X \quad \Gamma_1, Y, \Gamma_2 \to \Delta}{\Gamma_1, \Lambda, X \Rightarrow Y, \Gamma_2 \to \Delta}$$

The $\mathcal{L}$ rules are referred to as the *left rules*; the $\mathcal{R}$ rules as the *right* rules of the calculus. Here is a proof of the theorem $A \to (A \Rightarrow B) \Rightarrow B$.

$$\mathcal{R}\Rightarrow\frac{\mathcal{L}\Rightarrow\dfrac{\mathcal{I}\dfrac{}{A \to A} \quad \mathcal{I}\dfrac{}{B \to B}}{A, A \Rightarrow B \to B}}{A \to (A \Rightarrow B) \Rightarrow B}$$

As we alluded to in section 4.1, it is possible to associate with deductions *proof terms* encoding the proofs performed; these terms are λ-terms made up from the terms associated with the types in the sequents. The λ-terms come in various kinds; the ones we discuss are either *application terms* $t(t')$, saying that the functional term t is applied to the term t'; or *abstraction terms* $\lambda v.t$, a functional term taking a term v to a term t. Terms are in either *normal form* or *non-normal form*; in the latter case, terms contain subterms $(\lambda v.t)(t')$, so-called *redexes*. The relation called β-reduction allows the simplification of such a redex to $t[t'/v]$, which means that in term t, every occurrence of v is replaced by t'. The λ-terms for these deductions have the so-called *single-bind* property: every λ-bound variable v such that $\lambda v.t$ occurs exactly once in t; so we do not have terms $\lambda v.w$ where v does not occur in w, nor terms like $\lambda v.(t(v)(v))$. We then end up with the following rules:

$$\mathcal{I}\frac{}{t : X \to t : X}$$

$$\mathcal{R}\Rightarrow\frac{\Gamma, v : X \to t : Y}{\Gamma \to \lambda v.t : X \Rightarrow Y}$$

$$\mathcal{R}\Rightarrow\frac{v : X, \Gamma \to t : Y}{\Gamma \to \lambda v.t : X \Rightarrow Y}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to t' : X \quad \Gamma_1, t(t') : Y, \Gamma_2 \to \Delta}{\Gamma_1, t : X \Rightarrow Y, \Lambda, \Gamma_2 \to \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to t' : X \quad \Gamma_1, t(t') : Y, \Gamma_2 \to \Delta}{\Gamma_1, \Lambda, t : X \Rightarrow Y, \Gamma_2 \to \Delta}$$

The term for the proof above would be $\lambda P.P(t)$ giving the term t as a proof for A. Once one adds the so-called Cut rule to the calculus:

$$\text{Cut}\frac{\Gamma_2 \to \Lambda \quad \Gamma_1, \Lambda, \Gamma_3 \to \Delta}{\Gamma_1, \Gamma_2, \Gamma_3 \to \Delta}$$

λ-terms in non-normal form occur as proof terms. The Cut rule expresses the transitivity of the derivability relation $\to$.

Cut-free sequent calculus for the linear fragment of propositional logic has the so-called *subformula property*: premise sequents contain all and only subformulae of the conclusion sequent. Premise sequents have lower degree in terms of type-forming connectives: they contain one connective less than the conclusion sequents. From a top-down theorem proving regime, this means a steady reduction of complexity during deduction: one starts with a 'complex' sequent containing a lot of connectives, breaking this sequent down into sequents of smaller degree, until one reaches the axiom sequents of type $A \to A$, thus settling the conjecture of the conclusion sequent. In calculi with Cut, the subformula property no longer holds, since Λ can be any type, possibly increasing the degree of the premise sequent $\Gamma_1, \Lambda, \Gamma_3 \to \Delta$. Fortunately, the Cut elimination theorem (Gentzen's **Hauptsatz** (Gentzen, 1934)) says that Cut is a derivable rule: every proof with Cut can be transformed into a Cut-free proof. Cut-elimination leads to normal-form proof terms.

Here are the sequent rules for second-order quantifier types (Morrill, 1990).

$$\mathcal{L}\forall\frac{\Gamma_1, t(t') : \Lambda[t'/\_x], \Gamma_2 \to \alpha : X}{\Gamma_1, t : \forall(\_x).\Lambda, \Gamma_2 \to \alpha : X}$$

$$\mathcal{L}\exists\frac{\Gamma_1, \pi_2(t) : \Lambda[\pi_1(t)/\_x], \Gamma_2 \to \alpha : X}{\Gamma_1, t : \exists(\_x).\Lambda, \Gamma_2 \to \alpha : X}$$

$$\mathcal{R}\forall\frac{\Gamma \to t : \Lambda}{\Gamma \to \lambda\_x.t : \forall(\_x).\Lambda}$$

$$\mathcal{R}\exists\frac{\Gamma \to t_2 : \Lambda[t_1/\_x]}{\Gamma \to \langle t_1, t_2\rangle : \exists(\_x).\Lambda}$$

For $\{\mathcal{R}\forall, \mathcal{L}\exists\}$, the condition is that $\_x$ is not free in $\Gamma, \Gamma_1, \Gamma_2$; $t[\alpha/\beta]$ is the substitution of α for β in t and $\pi_i(t)$ is the i-th projection of the pair-term t: $\pi_1(\langle\alpha, \beta\rangle) = \alpha; \pi_2(\langle\alpha, \beta\rangle) = \beta$. The functional proof terms for $\forall$-types reflect the intuitionistic idea that a proof for a proposition $\forall(\_x).\Lambda$ consists of a method for proving the proposition expressed by Λ. This gives a more plausible interpretation of proofs for universal quantification once this quantification ranges over infinite domains: a mere truth-value then seems impossible to arrive at. The pair terms for $\exists$-types say that a proof for such a formula consists of an individual (a *witness*) and a term in which this individual is substituted for the bound variable. As noted earlier. the intuitionistic quantifier terms have a nice interpretation in our syntactic type calculus: a type $\forall(\_x).\Pi$ then becomes a specification of a method (proof) recognising all expressions of type Π on the basis of any (instantiation of) $\_x$.

We shall be silent about proof terms from now on, as they do not play an evident role in parsing HPSG. They could be of use in proving meta-results about HPSG parsing, however.

Given the calculus presented above, let us establish the fragment needed to perform deduction for HPSG.

4.3 ID rules as axiom schemata.

HPSG rules describe admissible, i.e. well-formed signs. In a type-theoretic setting, they can be interpreted as type definitions, since here, signs become types. A simple way to implement these definitions, is to formulate them as *axiom schemata* in a type calculus. That is, every rule R defining the sign Σ becomes an axiom scheme:

$$R \quad \dagger(\Sigma) \to \dagger(\Sigma)$$

Every axiom sequent thus becomes an instance of an ID rule. This assures that, whenever an axiom schema is used during deduction, the type check is effective.

For binary HPSG, this results in the following axiom schemata, where $\vec{\_x}$ schematises over types

$$\vec{Q}\_x_i \Rightarrow \ldots \Rightarrow \_x_n, 1 \leq i \leq n.$$

$$\begin{array}{l}\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)] \to\\ \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)]\end{array}$$

$$\begin{array}{l}\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\ \vec{\_x}\ .\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)] \to\\ \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\ \vec{\_x}\ .\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)]\end{array}$$

It is not too hard to recognise equivalents of the HPSG rules **R1** and **R2** in these axiom schemata, once one remembers that functors now combine with their arguments one at a time: in classical HPSG, there were only two kinds of functional configurations: a functor having consumed all of its arguments (treated by **R1**) and a functor having consumed all of its arguments but one (**R2**). In $\mathcal{D}$-HPSG, many more configurations arise, generally speaking: $n - 1$ for any n-placed functor. So, where the first axiom schema restores **R1**, the second can be seen to be a generalisation of **R2** to cover *any* kind of functional incompleteness. The lexicality demand on the head daughter Rule 2 makes vanishes here; functors consume one argument at a time, and once they have consumed one, they are no longer lexical.[3]

There is another option here: the ID-rules could be compiled away by ensuring they are consequently applied to every sign and its phrasal subsigns when the lexicon is created. Although this idea entirely hides the important concept of ID rules in the process of lexicon creation, it allows for using the regular axiom scheme

[3]The demand that Rule 2 makes on the non-invertedness of the head daughter is left unexpressed here.

$$\mathcal{I}\frac{}{t : X \rightarrow t : X}$$

4.4 Principles as inference rules and conditions

The various principles of HPSG appear to be easily reconciliable with the logical setting proposed. In HPSG, they do not form a homogenous class; some principles govern the flow of information in a feature structure, others create new information (like the LP principles). This is reflected in their proof-theoretic reconstruction.

4.4.1 Head Feature Principle

The Head Feature Principle of HPSG instantiates the head features of a fresh 'mother sign' to the head features of the head daughter. The necessity for doing so vanishes in the type-theoretic HPSG equivalent. To see this, notice that in the latter, all necessary feature transport is encoded by means of variable sharing in the *type assignments* for the lexical entries. Where HPSG uses the Subcat Principle to create new sign projections, with new `compdtrs` and `subcat` values, $\mathcal{D}$-HPSG never creates new sign projections during analysis: types only become gradually more developed in the sense that more and more variable subtypes become instantiated. Therefore, the Head Feature Principle becomes totally redundant: the head features of a functor (a verb, or whatever) are preserved and developed all the way. This makes $\mathcal{D}$-HPSG in a way more lexical than original HPSG. The distinction among head daughters and their superordinating signs vanishes as well; one reasonable thing to say is that the head feature principle is 'compiled away' in the lexicon, making this distinction irrelevant. So, we can suppress the `headdtr` attribute in our signs. Another option is to keep the attribute, letting it have as value a sign which has a nil value for `headdtr`.

4.4.2 Subcat Principle

The Subcat Principle is the motor behind syntactic combination in HPSG. Basically, what it does in original HPSG is to decompose the subcat list of a non-saturated sign, transferring a (non-fixed) number of entries on the list to the `compdtrs` attribute of a fresh mother sign, thus allowing the combination of the sign with suitable complements matching the `compdtrs` value. Unification takes care of making this match by recursively descending into the mother sign.

In $\mathcal{D}$-HPSG, the Subcat Principle has a binary shape: it secures the combination of a functor $A \Rightarrow B$ and a (single) argument expression A to a result type B. As $A \Rightarrow B$ is an undirected functor, combining with an A either to its left or right to form a B, we will need two versions (left and right) of the Subcat Principle. These can be interpreted as *inference rules*, i.e. the *left rules* for the propositional connective $\Rightarrow$ for undirected implication we saw earlier. The (type) variable sharing in these rules must now be understood as demands for unification on type level:

$$\mathcal{L}\Rightarrow\frac{\Lambda \rightarrow X \quad \Gamma_1, Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, X \Rightarrow Y, \Lambda, \Gamma_2 \rightarrow \Delta}$$

The Subcat principle covers *concatenative* functors only, i.e. functors which either follow or precede their arguments. For non-concatenative functors, such as the adverbial modifiers of section 3, we cannot use the concatenative connective $\Rightarrow$.

Borrowing the connective $\downarrow$ from categorial grammar (Moortgat, 1988), then, $A \downarrow B$ is a expression wanting to penetrate in an expression of type B to form an A. The adverbial adjuncts are typed $vp \downarrow vp$, where vp is an abbreviation of a formula $n(\ldots) \Rightarrow v(\ldots)$. It turns out to be technically impossible to establish a full logic for this connective under the perspective of antecedents as lists; only the rule $\mathcal{L}\downarrow$ can be formulated.[4] See Moortgat (1988, 1990) for discussion resp. a solution. For our purposes, this is enough, however: HPSG displays partial logics (left rules only) for functional connectives. The rule becomes:

$$\mathcal{L}\downarrow\frac{\Gamma_2, \Gamma_3 \rightarrow t' : B \quad \Gamma_1, t(t') : A, \Gamma_4 \rightarrow \Lambda}{\Gamma_1, \Gamma_2, t : A \downarrow B, \Gamma_3, \Gamma_4 \rightarrow \Delta}$$

[4] A full logic for this connective would make the structural rule of Permutation: $\frac{\Gamma' \rightarrow \Delta}{\Gamma \rightarrow \Delta}\,\mathtt{permutation}(\Gamma) = \Gamma'$ derivable. This means that antecedents now become treated as multisets (sets with repetition) rather than lists, which is not desirable for linguistic purposes.

9 References

Cooper, R. (1990): "Specifiers, Complements and Adjuncts in HPSG". Unpubl. ms.

Dörre, J., I. Raasch (1991): *The Stuttgart Type Unification Formalism- User Manual.* IBM, Stuttgart.

Duffy, D.A. (1991): *Principles of automated theorem proving.* Wiley, Chichester.

Gabbay, D. (1991): *Labelled Deductive Systems*, Oxford University Press, to appear.

Gallier, J. (1986): *Logic for Computer Science*, Foundations of Automatic Theorem Proving. Harper and Row, New York.

Gentzen, G. (1934): "Untersuchungen über das logische Schliessen". In: *Math. Z.*,39:176-210, 405-431.

Girard, J.-Y. (1987): "Linear Logic". In: *Theoretical Computer Science*,50:1-102.

König, E. (1989): "Parsing as natural deduction". In: Proc. ACL, Vancouver.

Moortgat, M. (1988): *Categorial Investigations*, logic and linguistic aspects of the Lambek calculus. Foris, Dordrecht.

Moortgat, M. (1990): "Discontinuous Type Constructors". Paper presented at the workshop Categorial Grammar and Linear Logic, 2nd European Summerschool on Language, Logic and Information.

Morrill, G. (1990): "Grammar and logical types". Unpubl. ms., Edingburgh.

Pollard, C, I.Sag (1987): *Information-based syntax and semantics*, vol.1, CSLI Lecture Notes 13, Stanford.

Pollard, C, I.Sag (1992): *Information-based syntax and semantics*, vol.2, CSLI Lecture Notes, Stanford (forthcoming)

Popowich, F., C.Vogel (1990): '"A Logic-Based Implementation of Head-Driven Phrase Structure Grammar". In: *Proc. of the Third International Workshop on Natural Language Understanding and Logic Programming.* Lidinogo,Stockholm.

Raaijmakers, S. (forthcoming): *Parsing HPSG. An evaluation of several parsing strategies.. Ms.*, ITK.

Reape, M. (1990): "Getting things in order". Paper presented at the Symposium on Discontinuous Constituency, Tilburg University, January 25-27th 1990.

Roorda, D. (1991): *Resource Logics.* Dissertation, University of Amsterdam.

Shieber, S. (1986): *An introduction to unification-based approaches to grammar*, CSLI Lecture Notes 4, Stanford.

van Benthem, J. (1991): *Language in action*, Categories, Lambdas, and Dynamic Logic. Studies in logic and the foundations of mathematics, vol. 130. North-Holland, Amsterdam.

Wallen, L.A. (1990): *Automated proof-search in non-classical logics.* MIT Press, Cambridge, Mass.

where Γ_i is a type sequence of length ≥ 0, with the exception that at least one of Γ_2, Γ_3 is non-empty. Notice that this rule generalizes over incompleteness in the following way: if Γ_2 is empty, $\downarrow$ is an instance of /; if Γ_3 is empty, $\downarrow$ is an instance of \.

4.4.3 Semantics Principle

The Semantics Principle can be 'compiled away' as well, by putting in the lexicon the semantics of a sign as a product of the semantics of its daughter signs. This makes it possible to incorporate various kinds of semantics into lexical signs, for instance, a simple application semantics:

```
[ [phon,...]
  [syn,...
     [subcat,[[...[sem,X]]]]
   ...]
  [sem,f(X)],
  [dtrs,...]]
```

4.4.4 LP principles

Once a functor combines with one of its arguments to form a mobile the LP principles apply to order the functor and argument branches by ordering the respective phon values to arrive at the phon value of the mother node. LP principles can address both aspects of argument and functor, so they must be functions of a pair of types $\mathcal{T}$ to sets of types:

$$\mathcal{T} \times \mathcal{T} \longrightarrow POW(\mathcal{T})$$

In case a concatenative functor combines with its arguments, the string ordering functions yield a singleton set of result types; for non-concatenative functors, this result set often has an arity greater than one, since there is generally more than one string position for a non-concatenative functor, and each separate string position determines a new sign.

The operationalisation of LP principles in $\mathcal{D}$-HPSG is as follows. Once a functor has applied to its arguments, both functor and argument types are fed to the LP principles, which figure out the phon value of the range subtype of the functor. This entails that LP principles in $\mathcal{D}$-HPSG should operate as *side-conditions* on inference rules:[5]

$$\frac{P_1 \quad \cdots \quad P_n}{C} \textbf{ if } \mathbf{LP}_i \vee \ldots \vee \mathbf{LP}_n$$

LP principles operate on an argument type and a functor type. Here are the type-theoretic equivalents of the LP principles of binary HPSG. The notation $(\mathbf{BLP}_n)A \otimes B = C$ says that the result of applying the LP principle n to argument A and functor B is C. As before, $\overset{\Rightarrow}{\overset{X}{n}}Y$ schematises over types

$$X_i \Rightarrow \ldots X_n \Rightarrow Y,\ 0 \leq i \leq n$$

and

$$\overset{\Rightarrow}{\overset{X}{1}}Z = X \Rightarrow Z.$$

Further, inverted(X) says that X is inverted, ADVMOD(X) that X is an adverbial modifier, and $\texttt{infix}(S2, S1) = S3$ that $S3$ is the infixation of $S2$ into $S1$. Uninteresting variables are suppressed with an underscore, and quantifiers are omitted. Anticipating on the implementation, we use (Prolog) *difference list* notation for list construction: the difference list $[a, b, c] - [c]$ is equivalent to the list $[a, b]$. This is done to optimise the expression of list construction: concatenation of two lists can now be expressed via variable sharing with one unit clause:

```
conc_dl(A-B,B-C,A-C).
```

For example,

$$\texttt{conc\_dl}([a, b|C] - C, [c] - [], [a, b, c] - []).$$

(BLP1)

$$\overset{\Rightarrow}{\overset{X}{n}}.Y(\texttt{phon(S2 - S3)}, \_, \_, \_)$$
$$\otimes$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S1 - S2)}, \texttt{syn(loc(\_h, lex(+)), \_b)}, \texttt{\_u}, \texttt{\_w})$$
$$=$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S1 - S3)}, \texttt{syn(loc(\_h, lex(+)), \_b)}, \texttt{\_u}, \texttt{\_w})$$

(BLP2)

$$\overset{\Rightarrow}{\overset{X}{n}}.Y(\texttt{phon(S1 - S2)}, \_, \_, \_)$$
$$\otimes$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S2 - S3)}, \texttt{syn(loc(\_h, lex(-)), \_b)}, \texttt{\_u}, \texttt{\_w})$$

[5] This relates the current enterprise to Gabbay's (Gabbay, 1991) *labelled deductive systems*, where side-conditions on inference rules occur as well.

$$= \overset{\Rightarrow}{X}1.Z(\texttt{phon(S1-S3)},\texttt{syn(loc(\_h,lex(-)),\_b)},\texttt{\_u},\texttt{\_w})$$

(BLP3)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S2-S3)},\_,\_,\_) \otimes 2 \overset{\Rightarrow}{\leq} X n .Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) = 2 \overset{\Rightarrow}{\leq} X n .Z(\texttt{phon(S1-S3)},\texttt{\_s},\texttt{\_t},\texttt{\_d}$$

(BLP4)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S2-S3)},\_,\_,\_) \otimes \overset{\Rightarrow}{X}m.Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) = \overset{\Rightarrow}{X}m.Z(\texttt{phon(S1-S3)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) \textbf{ if } \texttt{inverted}(Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}))$$

(BLP5)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S1)},\_,\_,\_) \otimes \overset{\Rightarrow}{X}m.Z(\texttt{phon(S2)},\texttt{\_v},\texttt{\_u},\texttt{\_w}) = \overset{\Rightarrow}{X}m.Z(\texttt{phon(S3)},\texttt{\_v},\texttt{\_u},\texttt{\_w}) \textbf{ if } \texttt{ADVMOD}(\overset{\Rightarrow}{X}m.Z(\texttt{phon(S2)},\texttt{\_v},\texttt{\_u},\texttt{\_w})) \textbf{ and } \texttt{infix(S2,S1)=S3}$$

4.5 Calculus

To summarize, here is the full calculus $\mathcal{D}$-HPSG. LP principles apply as discussed earlier to each inference rule as side-condition; they are suppressed below.

$$\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)] \rightarrow \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)]$$

$$\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\overset{\Rightarrow}{x}.\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)] \rightarrow \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\overset{\Rightarrow}{x}.\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)]$$

$$\mathcal{L}\Rightarrow\frac{\Gamma_1, t(t') : Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, t : X \Rightarrow Y, t' : X, \Gamma_2 \rightarrow \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Gamma_1, t(t') : Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, t' : X, t : X \Rightarrow Y, \Gamma_2 \rightarrow \Delta}$$

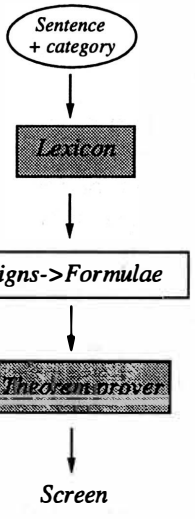
$$\mathcal{L}\forall\frac{\Gamma_1, y : \Lambda[t'/\_x], \Gamma_2 \rightarrow \alpha : X}{\Gamma_1, t : \forall(\_x).\Lambda, \Gamma_2 \rightarrow \alpha[t(t')/y] : X}$$

$$\mathcal{R}\forall\frac{\Gamma \rightarrow t : \Lambda}{\Gamma \rightarrow \lambda\_x.t : \forall(\_x).\Lambda}$$

$$\mathcal{L}\downarrow\frac{\Gamma_2, \Gamma_3 \rightarrow t' : B \quad \Gamma_1, t(t') : A, \Gamma_4 \rightarrow \Lambda}{\Gamma_1, \Gamma_2, t : A \downarrow B, \Gamma_3, \Gamma_4 \rightarrow \Delta}$$

5 Implementation

5.1 Design



The lexicon — in the overall design of the implementation pictured above — consists of lexeme-sign pairs. When a sentence _s is entered to

be analysed, first the lexemes are looked up in the lexicon. The corresponding signs are compiled to a sequence Π of formulae according to the map † of section 4.1. Then, given the result category _c, which the user is prompted to enter, the sequent $\Pi \rightarrow \_c(phon(\_s), \_syn, \_sem, \_dtrs)$ is formed, with *_syn*, *_sem*, *_dtrs* variables for syntactic, semantic and daughter information. The sequent is handed to the theorem prover, which is a meta-interpreter performing sequent deduction. Output is in both Prolog and picture format: the derived type formula for an utterance is printed on screen as a Prolog term; in a separate window the sign equivalent of the formula is drawn in standard HPSG format.

5.2 Term unification and uniform signs

Unification is a computationally expensive tool, boiling down to extensive graph inspection and merging. This cost can be saved by adopting one uniform structure for the items to be unified, i.c. signs: if we treat these signs just like rigid term structures, we could directly make use of Prolog's built-in *term unification* mechanism for unifying them. This idea entails that every lexeme in the lexicon has a fixed sign type, with standard slots and values. The structure of this sign is as follows:

```
[[phon,_p]
 [syn,[[loc,[[head,[[maj,_m],
                   [case,_c],
                   [nform,_n],
                   [vform,_v],
                   [aux,_a],
                   [inv,_i],
                   [prd,_j]]],
            [subcat,_s],
            [lex,_l]]],
      [bind,_b]]],
 [sem,_t],
 [dtrs,[[headdtr,_hd],
        [compdtrs,_cd]]]]]
```

Lexemes can be unspecified for certain attributes; these attributes then carry the atomic value **nil**.

In Prolog, variables occurring in literals are implicitly universally quantified, which means that quantifiers are removed from them. This makes it possible to use Prolog's term unification mechanism directly to instantiate values to the variables; see Duffy (1991) for discussion. So, we can simply strip quantifiers from a formula: $\vec{Q}.\mathcal{F} \mapsto \mathcal{F}$.

It is a well-known fact from the proof-theory of predicate logic that once both kinds of quantification (universal and existential) are used, deductions are not invariant under permutation of rule application: the application of quantifier elimination rules becomes order-sensitive (Wallen, 1990). This effect does not take place here, since we only have universal quantification.

5.3 Calculus in Prolog format

In Prolog format, the axioms and rules of the calculus have the following shape:

```
axiom(Name,Antecedent--->Succedent
rule(Name,Antecedent--->Succedent (if C)
```

Given a functor type $A_1 \Rightarrow \ldots A_n \Rightarrow B$, which is right-associative, i.e. $A_1 \Rightarrow (\ldots (A_n \Rightarrow B) \ldots)$, it is necessary to inspect the properties of the ultimate range subtype B when applying the LP principles. This would involve descending into the functor type, until one reaches the ultimate subtype B.

From a practical point of view, we would like to avoid such descent; we therefore notate such a functor as

$$\langle A_1, \ldots, A_n \rangle \Rightarrow B$$

where $\langle A_1, \ldots, A_n \rangle$ is a list of types. For the infix types $A \downarrow B$ we do likewise; they are usually of the form $(A_2 \Rightarrow \ldots A_n \Rightarrow B) \downarrow A_1$. To distinguish the first argument from the rest of the arguments, we write

$$\langle \#A_1, A_2, \ldots, A_n \rangle \Rightarrow B$$

In Prolog, we write `B@(_x,...,_y)` for B(*_x*, ..., *_y*). Sequents are written

$$(L_1, \ldots, L_n) \rightarrow X,$$

where each L_i is a subsequence (list) of types; type sequences of length 1 are denoted as $[T]$, with T a type or type variable. Some examples of the axiom and rule format are then:

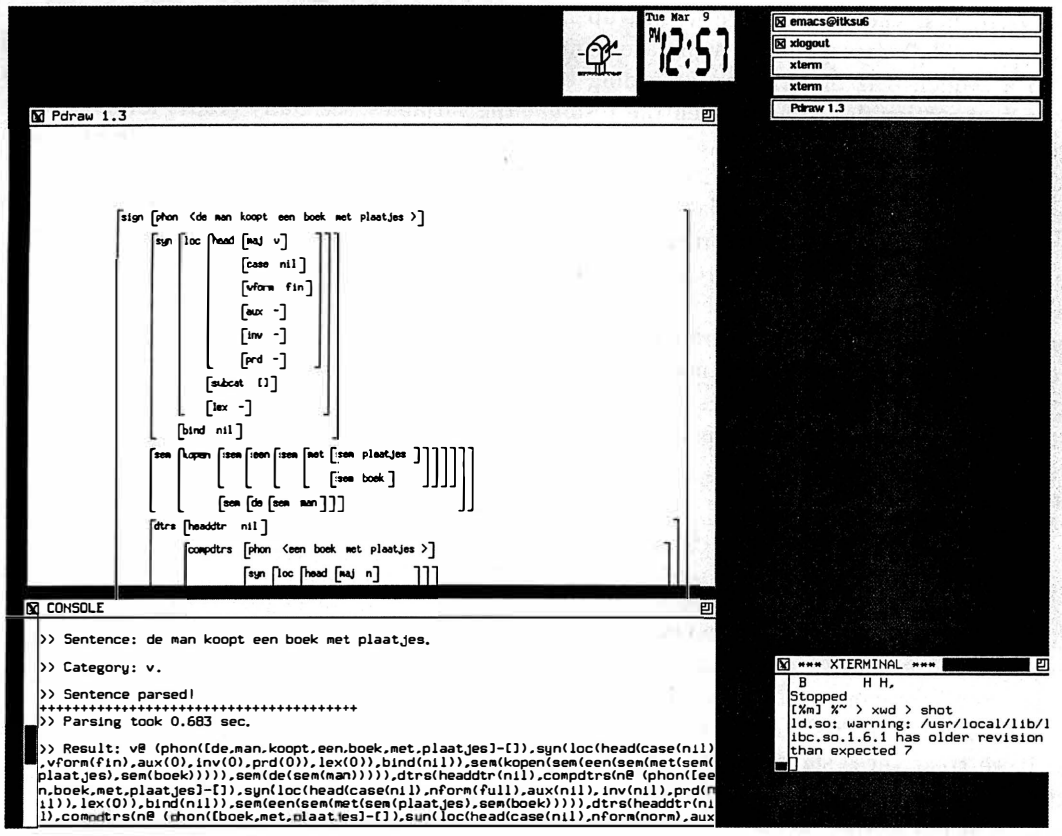


Figure 3: Screendump.

```
axiom(rule1,[C@(P,S,B,D)]--->
            [C@(P,S,B,D)]).
rule(subcat_left,
(T1,T2,[[X|T]=>B@(S1,
                  syn(loc(W1,lex(L)),
                  W2),
                S,D)],V) ---> Z
         if
T2 ---> [X]
         and
{lp(LP_Name,
 X,
 [X|T]=>B@(S1,syn(loc(W1,lex(L)),W2),
             S,D),
 [X|T]=>B@(S3,syn(loc(W1,lex(L)),W2),
             S,D))}
       and
(T1,[T=>B@(S3,syn(loc(W1,lex(0)),W2),
                S,D)],V)
  ---> Z
).
```

Notice how the lexicality value is changed to 0 here, once the functor has combined with its argument. Above we argued that the lexicality feature is a derived feature, arising from the absence or presence of daughters in a tree. Since lexical signs already have (variable) daughters in $\mathcal{D}$-HPSG, checking for lexicality could (and should) be implemented here by inspection of the daughter specifications on the functor type: if the first daughter entry in the `compdtrs` attribute list has, say, a *variable* phon value, the sign as a whole can be concluded to be lexical. For reasons of efficiency, we implement this view on lexicality by switching to non-lexicality the moment a functor combines with an argument. The variable L expresses the irrelevance of the (non-)lexicality of the functor symbol: no matter what value the functor has for `lex`, the range type will have the value - for the attribute `lex`.

5.4 Theorem prover

Various theorem proving techniques can be implemented quite easily in Prolog. As theorem prover, we use a simple sequent-proving device, implemented as follows: the prover is a set of clauses for a predicate `prove(+Goal,-Rules)`, where `Goal` is either a sequence of sequents of length minimally 1, or a structure $\{G_1,\ldots,G_n\}$ with each G_i $(1 \leq i \leq n)$ a non-sequent goal; `Rules` is a list encoding the inference rules and axioms used for proving `Goal`. Initially, `Goal` is the sequent to be proved. The predicate `prove/2` calls a routine matching the sequent against the database of inference rules, i.e. if `Goal` is of the form $X_1,\ldots,X_n \rightarrow Y$, it tries to match (resolve) the sequent against the rules and axioms of the calculus, which take the shape of $A \rightarrow B$ (**if** C). Once the match of $X_1,\ldots,X_n$ against A and Y against B has been made, the eventual premises C are attempted to prove.

```
prove({Goals},[]):- call(Goals).
prove(A--->B,[Rule|Rules]):-
        rule(Rule,(X ---> B if Y)),
        resolve(A,X),
        prove(Y,Rules).
prove(A and B,Rules):-
        prove(A,R1),
        prove(B,R2),
        conc(R1,R2,Rules).
prove(A--->B,[Ax]):-
        axiom(Ax,A1--->B),
        resolve(A,A1).
```

The linear precedence principles are (as illustrated in section 5.3) encoded as goals $\{lp_i,\ldots,lp_n\}$, to be called before entering the eventual premise sequents.

5.5 Principles

Here are some linear precedence principles. They are written as

```
lp(Name,Arg,Funct,NewFunct),
```

with `Name` the name of the principle, `Arg`, `Funct`, `NewFunct` types such that `NewFunct` has as its phonology value the ordered phonology values of `Arg` and `Funct`. Uninteresting variables are written as underscores.

```
lp(lp1,
 X@(phon(S2-S3),_,_,_),
 [X@(_,_,_,_)]=>B@(phon(S1-S2),
                   syn(loc(W1,lex(1)),W2),
                   S,D),
 [X@(_,_,_,_)]=>B@(phon(S1-S3),
                syn(loc(W1,lex(1)),W2),
                S,D)).

lp(lp4,
  X@(phon(S2-S3),_,_,_),
  [X@(_,_,_,_)|T]=>B@(phon(S1-S2),P,Q,D),
  [X@(_,_,_,_)|T]=>B@(phon(S1-S3),P,Q,D))
        :-
  inverted(B@(_,P,_,_)).
```

6 Sample lexical entries

Lexical entries are of the form

```
WORD := SIGN (← VAR_CONDITIONS)
```

with `WORD`,`SIGN` resp. a lexeme and its sign representation, and the optional `<- VAR_CONDITIONS` encoding instantiations of variables mentioned in `SIGN` (this is just done to avoid having to type very complex signs).

Some (partially specified) sample lexical entries are:

```
loopt :=
[[phon,[loopt|I]-I],
 [syn,[[loc,[[head,[[maj,v],
                   [case,nil],
                   [nform,nil],
                   [vform,fin],
                   [aux,0],
                   [inv,0],
                   [prd,0]]],
            [subcat,[X]],
              [lex,0]]],
         [bind,nil]]],
  [sem,_],
  [dtrs,[[headdtr,nil],
          [compdtrs,[X]]]]])
       <-

X= [[phon,_],
    [syn,[[loc,[[head,[[maj,n],
                      [case,nil],
                      [nform,_],
                      [vform,nil],
```

```
                        [aux,nil],
                        [inv,nil],
                        [prd,nil]]],
                  [subcat,[]],
                  [lex,0]]],
            [bind,_]]],
    [sem,_],
    [dtrs,_]].

de :=
[[phon,[de|I]-I],
 [syn,[[loc,[[head,[[maj,n],
                    [case,nil],
                    [nform,_],
                    [vform,nil],
                    [aux,nil],
                    [inv,nil],
                    [prd,nil]]],
                [subcat,[X]],
                 [lex,1]]],
       [bind,nil]]],
 [sem,_],
 [dtrs,[[headdtr,nil],
        [compdtrs,[X]]]]])
    <-

X= [[phon,_],
    [syn,[[loc,[[head,[[maj,n],
                       [case,nil],
                       [nform,_],
                       [vform,nil],
                       [aux,nil],
                       [inv,nil],
                       [prd,nil]]],
                  [subcat,[]],
                  [lex,1]]],
            [bind,_]]],
    [sem,_],
    [dtrs,_]].

man :=
[[phon,[man|I]-I],
 [syn,[[loc,[[head,[[maj,n],
                    [case,nil],
                    [nform,norm],
                    [vform,nil],
                    [aux,nil],
                    [inv,nil],
                    [prd,nil]]],
          [subcat,[]],
          [lex,1]]],
       [bind,_]]],
 [sem,_],
 [dtrs,[[headdtr,nil],
        [compdtrs,[]]]]]).
```

7 Performance

The following overview lists real-time parsing results for a small set of Dutch sentences. The results were generated by a compiled Prolog executable version of the program, running under X-windows on a SUN SPARCstation 1.

The sentences and their word-by-word translations are:

1. jan loopt.
   john walks.
2. de man loopt.
   the man walks.
3. de man loopt graag.
   the man walks gladly.
4. jan heeft hard gelopen.
   john has fast walked.
5. jan slaat de man.
   john hits the man.
6. john slaat graag de hond.
   john hits gladly the dog.
7. de man koopt een boek met plaatjes.
   the man buys a book with pictures.
8. jan geeft marie de hond.
   john gives mary the dog.
9. jan geeft marie een boek met plaatjes.
   john gives mary a book with pictures.
10. jan geeft marie graag een boek.
    john gives mary gladly a book.
11. dat jan de hond slaat.
    that john the dog hits.

```
>> Sentence: jan loopt.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.000 sec.

>> Sentence: de man loopt.
```

```
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.000 sec.

>> Sentence: de man loopt graag.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.050 sec.

>> Sentence: jan heeft hard gelopen.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.050 sec.

>> Sentence: jan slaat de man.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.067 sec.

>> Sentence: jan slaat graag de hond.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.450 sec.

>> Sentence: de man koopt een boek met
             plaatjes.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.683 sec.

>> Sentence: jan geeft marie de hond.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.117 sec.

>> Sentence: jan geeft marie een boek
             met plaatjes.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 2.500 sec.

>> Sentence: jan geeft marie graag een
             boek.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 1.316 sec.

>> Sentence: dat jan de hond slaat.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.017 sec.
```

As can be concluded from the output presented in the previous section, performance is relatively good. Sentences taking a lot of time (say, over 1 second), invariably contain at least one adverbial modifier, or involve an NP closure problem. For instance, in

- John gives Mary a book with pictures

the phrase 'John gives Mary a book' can be erroneously analysed as a sentence before the PP 'with pictures' is attached to 'a book'. Once the parser detects the remaining phrase 'with pictures', it will have to backtrack and redo a lot of work. The bad performance is a consequence of the sequent formalism: for any configuration

$$X_1 \Rightarrow X_2, X_1, X_1 \Rightarrow X_1$$

where each X_i is distinct, the analysis

$$((X_1 \Rightarrow X_2, X_1), X_1 \Rightarrow X_1)$$

is tried. One idea would be to employ a well-formed substring table encoding intermediate parsing results, to avoid having to reparse too much once the parser starts backtracking.

Generally speaking, weak performance for long sentences is not surprising, since the various inference rules allow for blind alleys in the left premise deduction, by instantiating wrong subsequences of the antecedent to the factor reducing to an argument type. This is a direct consequence of the non-deterministic nature of the procedure decomposing the antecedent into contexts around a functional type. The problem can be fixed by introducing so-called *proof invariants* (van Benthem, 1991) into the theorem prover. Proof invariants are structural validities for antecedent-succedent pairs, which serve to prune irrelevant options from the search space. The attractive feature of the current setting is that *any* optimalisation coming from proof theory can be used to optimise the parser.

8 Concluding remarks

We have shown that it is possible to give HPSG a deductive basis. The binary version of HPSG we have proposed, has been demonstrated to correspond to a fragment of second-order linear logic. The binarity of this HPSG dialect, which is faithful to classical HPSG in all other respects, is motivated from practical rather than theoretical reasons; in fact, the current approach is open to any version of HPSG. The parser we developed is, although relatively fast, in need of further optimalisation; the use of proof invariants may help to reduce the search space. Also, recently developed low-complexity theorem proving techniques such as *proof nets* (Roorda, 1991), may be of use here. Returning to the five desiderata of section 1, then, the last item, "The parser should have reasonable time/space complexity" has not fully been met yet.

Acknowledgements

I thank my colleagues René Ahn, Miriam Mulders, Gerrit Rentier and Leon Verschuur for fruitful discussion and taking an active interest in the enterprise.