A DEFINITE CLAUSE VERSION OF CATEGORIAL GRAMMAR

Remo Pareschi*
Department of Computer and Information Science,
University of Pennsylvania,
200 S. 33rd St., Philadelphia, PA 19104† and
Department of Artificial Intelligence and
Centre for Cognitive Science,
University of Edinburgh,
2 Buccleuch Place,
Edinburgh EH8 9LW, Scotland
remo@linc.cis.upenn.edu

ABSTRACT

We introduce a first-order version of Categorial Grammar, based on the idea of encoding syntactic types as definite clauses. Thus, we drop all explicit requirements of adjacency between combinable constituents, and we capture word-order constraints simply by allowing subformulae of complex types to share variables ranging over string positions. We are in this way able to account for constructions involving discontinuous constituents. Such constructions are difficult to handle in the more traditional version of Categorial Grammar, which is based on propositional types and on the requirement of strict string adjacency between combinable constituents.

We show then how, for this formalism, parsing can be efficiently implemented as theorem proving. Our approach to encoding types as definite clauses presupposes a modification of standard Horn logic syntax to allow internal implications in definite clauses. This modification is needed to account for the types of higher-order functions and, as a consequence, standard Prolog-like Horn logic theorem proving is not powerful enough. We tackle this problem by adopting an intuitionistic treatment of implication, which has already been proposed elsewhere as an extension of Prolog for implementing hypothetical reasoning and modular logic programming.

*I am indebted to Dale Miller for help and advice. I am also grateful to Aravind Joshi, Mark Steedman, David Weir, Bob Frank, Mitch Marcus and Yves Schabes for comments and discussions. Thanks are due to Elsa Gunter and Amy Felty for advice on typesetting. Parts of this research were supported by: a Sloan foundation grant to the Cognitive Science Program, Univ. of Pennsylvania; and NSF grants MCS-8219196-CER, IRI-10413 AO2, ARO grants DAA29-84-K-0061, DAA29-84-9-0027 and DARPA grant NOOO14-85-K0018 to CIS, Univ. of Pennsylvania.

†Address for correspondence

1 Introduction

Classical Categorial Grammar (CG) [1] is an approach to natural language syntax where all linguistic information is encoded in the lexicon, via the assignment of syntactic types to lexical items. Such syntactic types can be viewed as expressions of an implicational calculus of propositions, where atomic propositions correspond to atomic types, and implicational propositions account for complex types. A string is grammatical if and only if its syntactic type can be logically derived from the types of its words, assuming certain inference rules.

In classical CG, a common way of encoding word-order constraints is by having two symmetric forms of "directional" implication, usually indicated with the forward slash / and the backward slash \, constraining the antecedent of a complex type to be, respectively, right- or left-adjacent. A word, or a string of words, associated with a right-(left-) oriented type can then be thought of as a right- (left-) oriented function looking for an argument of the type specified in the antecedent. A convention more or less generally followed by linguists working in CG is to have the antecedent and the consequent of an implication respectively on

the right and on the left of the connective. Thus, the type-assignment (1) says that the ditransitive verb put is a function taking a right-adjacent argument of type *NP*, to return a function taking a right-adjacent argument of type *PP*, to return a function taking a left-adjacent argument of type *NP*, to finally return an expression of the atomic type *S*.

(1) $\mathsf{put}: \ ((S\backslash NP)/PP)/NP$

The Definite Clause Grammar (DCG) framework [14] (see also [13]), where phrase-structure grammars can be encoded as sets of definite clauses (which are themselves a subset of Horn clauses), and the formalization of some aspects of it in [15], suggests a more expressive alternative to encode word-order constraints in CG. Such an alternative eliminates all notions of directionality from the logical connectives, and any explicit requirement of adjacency between functions and arguments, and replaces propositions with first-order formulae. Thus, atomic types are viewed as atomic formulae obtained from two-place predicates over string positions represented as integers, the first and the second argument corresponding, respectively, to the left and right end of a given string. Therefore, the set of all sentences of length j generated from a certain lexicon corresponds to the type $S(0, j)$. Constraints over the order of constituents are enforced by sharing integer indices across subformulae inside complex (functional) types.

This first-order version of CG can be viewed as a logical reconstruction of some of the ideas behind the recent trend of Categorial Unification Grammars [5, 18, 20][1]. A strongly analogous development characterizes the systems of type-assignment for the formal languages of Combinatory Logic and Lambda Calculus, leading from propositional type systems to the "formulae-as-types" slogan which is behind the current research in type theory [2]. In this paper, we show how syntactic types can be encoded using an extended version of standard Horn logic syntax.

2 Definite Clauses with Internal Implications

Let $\wedge$ and $\rightarrow$ be logical connectives for conjunction and implication, and let $\forall$ and $\exists$ be the universal and existential quantifiers. Let A be a syntactic variable ranging over the set of *atoms*, i. e. the set of atomic first-order formulae, and let D and G be syntactic variables ranging, respectively, over the set of *definite clauses* and the set of *goal clauses*. We introduce the notions of *definite clause* and of *goal clause* via the two following mutually recursive definitions for the corresponding syntactic variables D and G:

- $D := A \mid G \rightarrow A \mid \forall x D \mid D_1 \wedge D_2$
- $G := A \mid G_1 \wedge G_2 \mid \exists x G \mid D \rightarrow G$

We call *ground* a clause not containing variables. We refer to the part of a non-atomic definite clause coming on the left of the implication connective as to the *body* of the clause, and to the one on the right as to the *head*. With respect to standard Horn logic syntax, the main novelty in the definitions above is that we permit implications in goals and in the bodies of definite clauses. Extended Horn logic syntax of this kind has been proposed to implement hypothetical reasoning [3] and modules [7] in logic programming. We shall first make clear the use of this extension for the purpose of linguistic description, and we shall then illustrate its operational meaning.

3 First-order Categorial Grammar

3.1 Definite Clauses as Types

We take *CONN* (for "connects") to be a three-place predicate defined over lexical items and pairs of integers, such that $CONN(item, i, j)$ holds if and only if and only if $i = j - 1$, with the intuitive meaning that *item* lies between the two consecutive string positions i and j. Then, a most direct way to translate in first-order logic the type-assignment (1) is by the type-assignment (2), where, in the formula corresponding to the assigned type, the non-directional implication connective $\rightarrow$ replaces the slashes.

$$\begin{aligned}(2)\ \mathsf{put}: \ \forall x \forall y \forall z \forall w [&CONN(\mathsf{put}, y-1, y) \rightarrow \\ &(NP(y, z) \rightarrow \\ &(PP(z, w) \rightarrow \\ &(NP(x, y-1) \rightarrow \\ &S(x, w))))]\end{aligned}$$

[1] Indeed, Uszkoreit [18] mentions the possibility of encoding order constraints among constituents via variables ranging over string positions in the DCG style.

A definite clause equivalent of the formula in (2) is given by the type-assignment (3)[2].

$$(3)\ \mathsf{put}: \forall x\forall y\forall z\forall w[CONN(\mathsf{put}, y-1, y) \wedge NP(y,z) \wedge PP(z,w) \wedge NP(x, y-1) \rightarrow S(x,w)]$$

Observe that the predicate *CONN*will need also to be part of types assigned to "non-functional" lexical items. For example, we can have for the noun-phrase Mary the type-assignment (4).

$$(4)\quad \mathsf{Mary}: \forall y[CONN(\mathsf{Mary}, y-1, y) \rightarrow NP(y-1, y)]$$

3.2 Higher-order Types and Internal Implications

Propositional CG makes crucial use of functions of higher-order type. For example, the type-assignment (5) makes the relative pronoun which into a function taking a right-oriented function from noun-phrases to sentences and returning a relative clause[3]. This kind of type-assignment has been used by several linguists to provide attractive accounts of certain cases of extraction [16, 17, 10].

$$(5)\qquad \mathsf{which}: REL/(S/NP)$$

In our definite clause version of CG, a similar assignment, exemplified by (6), is possible, since implications are allowed in the body of clauses. Notice that in (6) the noun-phrase needed to fill the extraction site is "virtual", having null length.

$$(6)\quad \mathsf{which}: \forall v\forall y[CONN(\mathsf{which}, v-1, v) \wedge (NP(y,y) \rightarrow S(v,y)) \rightarrow REL(v-1, y)]$$

[2] See [2] for a pleasant formal characterization of first-order definite clauses as type declarations.

[3] For simplicity sake, we treat here relative clauses as constituents of atomic type. But in reality relative clauses are noun modifiers, that is, functions from nouns to nouns. Therefore, the propositional and the first-order atomic type for relative clauses in the examples below should be thought of as shorthands for corresponding complex types.

3.3 Arithmetic Predicates

The fact that we quantify over integers allows us to use arithmetic predicates to determine subsets of indices over which certain variables must range. This use of arithmetic predicates characterizes also Rounds' ILFP notation [15], which appears in many ways interestingly related to the framework proposed here. We show here below how this capability can be exploited to account for a case of extraction which is particularly problematic for bidirectional propositional CG.

3.3.1 Non-peripheral Extraction

Both the propositional type (5) and the first-order type (6) are good enough to describe the kind of constituent needed by a relative pronoun in the following right-oriented case of *peripheral* extraction, where the extraction site is located at one end of the sentence. (We indicate the extraction site with an upward-looking arrow.)

$$\mathsf{which}\ [\ \mathsf{I\ shall\ put\ a\ book\ on}\ \uparrow\ ]$$

However, a case of *non-peripheral* extraction, where the extraction site is in the middle, such as

$$\mathsf{which}\ [\ \mathsf{I\ shall\ put}\ \uparrow\ \mathsf{on\ the\ table}\ ]$$

is difficult to describe in bidirectional propositional CG, where all functions must take left- or right-adjacent arguments. For instance, a solution like the one proposed in [17] involves permuting the arguments of a given function. Such an operation needs to be rather cumbersomely constrained in an explicit way to cases of extraction, lest it should wildly overgenerate. Another solution, proposed in [10], is also cumbersome and counterintuitive, in that involves the assignment of multiple types to wh-expressions, one for each site where extraction can take place.

On the other hand, the greater expressive power of first-order logic allows us to elegantly generalize the type-assignment (6) to the type-assignment (7). In fact, in (7) the variable identifying the extraction site ranges over the set of integers in between the indices corresponding, respectively, to the left and right end of the sentence on which the relative pronoun operates. Therefore, such a sentence can have an extraction site anywhere between its string boundaries.

(7) which : $\forall v \forall y \forall w [CONN(\text{which}, v-1, v) \wedge (NP(y,y) \rightarrow S(v,w)) \wedge v \leq y \wedge y \leq w \rightarrow REL(v-1, w)\ ]$

Non-peripheral extraction is but one example of a class of *discontinuous* constituents, that is, constituents where the function-argument relation is not determined in terms of left- or right-adjacency, since they have two or more parts disconnected by intervening lexical material, or by internal extraction sites. Extraposition phenomena, gapping constructions in coordinate structures, and the distribution of adverbials offer other problematic examples of English discontinuous constructions for which this first-order framework seems to promise well. A much larger batch of similar phenomena is offered by languages with freer word order than English, for which, as pointed out in [5, 18], classical CG suffers from an even clearer lack of expressive power. Indeed, Joshi [4] proposes within the TAG framework an attractive general solution to word-order variations phenomena in terms of linear precedence relations among constituents. Such a solution suggests a similar approach for further work to be pursued within the framework presented here.

4 Theorem Proving

In propositional CG, the problem of determining the type of a string from the types of its words has been addressed either by defining certain "combinatory" rules which then determine a rewrite relation between sequences of types, or by viewing the type of a string as a logical consequence of the types of its words. The first alternative has been explored mainly in Combinatory Grammar [16, 17], where, beside the rewrite rule of *functional application*, which was already in the initial formulation of CG in [1], there are also the rules of *functional composition* and *type raising*, which are used to account for extraction and coordination phenomena. This approach offers a psychologically attractive model of parsing, based on the idea of incremental processing, but causes "spurious ambiguity", that is, an almost exponential proliferation of the possible derivation paths for identical analyses of a given string. In fact, although a rule like functional composition is specifically needed for cases of extraction and coordination, in principle nothing prevents its use to analyze strings not characterized by such phenomena, which would be analyzable in terms of functional application alone. Tentative solutions of this problem have been recently discussed in [12, 19].

The second alternative has been undertaken in the late fifties by Lambek [6] who defined a decision procedure for bidirectional propositional CG in terms of a Gentzen-style sequent system. Lambek's implicational calculus of syntactic types has recently enjoyed renewed interest in the works of van Benthem, Moortgat and other scholars. This approach can account for a range of syntactic phenomena similar to that of Combinatory Grammar, and in fact many of the rewrite rules of Combinatory Grammar can be derived as theorems in the calculus. However, analyses of cases of extraction and coordination are here obtained via inferences over the internal implications in the types of higher-order functions. Thus, extraction and coordination can be handled in an expectation-driven fashion, and, as a consequence, there is no problem of spuriously ambiguous derivations.

Our approach here is close in spirit to Lambek's enterprise, since we also make use of a Gentzen system capable of handling the internal implications in the types of higher-order functions, but at the same time differs radically from it, since we do not need to have a "specialized" propositional logic, with directional connectives and adjacency requirements. Indeed, the expressive power of standard first-order logic completely eliminates the need for this kind of specialization, and at the same time provides the ability to account for constructions which, as shown in section 3.3.1, are problematic for an (albeit specialized) propositional framework.

4.1 An Intuitionistic Extension of Prolog

The inference system we are going to introduce below has been proposed in [7] as an extension of Prolog suitable for modular logic programming. A similar extension has been proposed in [3] to implement hypotethical reasoning in logic programming. We are thus dealing with what can be considered the specification of a general purpose logic programming language. The encoding of a particular linguistic formalism is but one other application of such a language, which Miller [7] shows to be sound and complete for intuitionistic logic, and to have a well defined semantics in terms of

Kripke models.

4.1.1 Logic Programs

We take a *logic program* or, simply, a program $\mathcal{P}$ to be any set of definite clauses. We formally represent the fact that a goal clause G is logically derivable from a program $\mathcal{P}$ with a *sequent* of the form $\mathcal{P} \Rightarrow G$, where $\mathcal{P}$ and G are, respectively, the *antecedent* and the *succedent* of the sequent. If $\mathcal{P}$ is a program then we take its *substitution closure* $[\mathcal{P}]$ to be the smallest set such that

- $\mathcal{P} \subseteq [\mathcal{P}]$
- if $D_1 \wedge D_2 \in [\mathcal{P}]$ then $D_1 \in [\mathcal{P}]$ and $D_2 \in [\mathcal{P}]$
- if $\forall x D \in [\mathcal{P}]$ then $[x/t]D \in [\mathcal{P}]$ for all terms t, where $[x/t]$ denotes the result of substituting t for free occurrences of t in D

4.1.2 Proof Rules

We introduce now the following proof rules, which define the notion of proof for our logic programming language:

(I) $\mathcal{P} \Rightarrow G$ if $G \in [\mathcal{P}]$

(II) $\dfrac{\mathcal{P} \Rightarrow G}{\mathcal{P} \Rightarrow A}$ if $G \rightarrow A \in [\mathcal{P}]$

(III) $\dfrac{\mathcal{P} \Rightarrow G_1 \qquad \mathcal{P} \Rightarrow G_2}{\mathcal{P} \Rightarrow G_1 \wedge G_2}$

(IV) $\dfrac{\mathcal{P} \Rightarrow [x/t]G}{\mathcal{P} \Rightarrow \exists x G}$

(V) $\dfrac{\mathcal{P} \cup \{D\} \Rightarrow G}{\mathcal{P} \Rightarrow D \rightarrow G}$

In the inference figures for rules **(II)** - **(V)**, the sequent(s) appearing above the horizontal line are the *upper sequent(s)*, while the sequent appearing below is the *lower sequent*. A proof for a sequent $\mathcal{P} \Rightarrow G$ is a tree whose nodes are labeled with sequents such that *(i)* the root node is labeled with $\mathcal{P} \Rightarrow G$, *(ii)* the internal nodes are instances of one of proof rules **(II)** - **(V)** and *(iii)* the leaf nodes are labeled with sequents representing proof rule **(I)**. The *height* of a proof is the length of the longest path from the root to some leaf. The *size* of a proof is the number of nodes in it.

Thus, proof rules **(I)**-**(V)** provide the abstract specification of a first-order theorem prover which can then be implemented in terms of depth-first search, backtracking and unification like a Prolog interpreter. (An example of such an implementation, as a metainterpreter on top of Lambda-Prolog, is given in [9].) Observe however that an important difference of such a theorem prover from a standard Prolog interpreter is in the wider distribution of "logical" variables, which, in the logic programming tradition, stand for existentially quantified variables within goals. Such variables can get instantiated in the course of a Prolog proof, thus providing the procedural ability to return specific values as output of the computation. Logical variables play the same role in the programming language we are considering here; moreover, they can also occur in program clauses, since subformulae of goal clauses can be added to programs via proof rule **(V)**.

4.2 How Strings Define Programs

Let α be a string $a_1 \ldots a_n$ of words from a lexicon $\mathcal{L}$. Then α defines a program $\mathcal{P}_\alpha = \Gamma_\alpha \cup \Delta_\alpha$ such that

- $\Gamma_\alpha = \{CONN(a_i, i-1, i) \mid 1 \leq i \leq n\}$
- $\Delta_\alpha = \{D \mid a_i : D \in \mathcal{L} \text{ and } 1 \leq i \leq n\}$

Thus, Γ_α just contains ground atoms encoding the position of words in α. Δ_α contains instead all the types assigned in the lexicon to words in α. We assume arithmetic operators for addition, subtraction, multiplication and integer division, and we assume that any program $\mathcal{P}_\alpha$ works together with an infinite set of axioms $\mathcal{A}$ defining the comparison predicates over ground arithmetic expressions $<, \leq, >, \geq$. (Prolog's evaluation mechanism treats arithmetic expressions in a similar way.) Then, under this approach a string α is of type G_α if and only if there is a proof for the sequent $\mathcal{P}_\alpha \cup \mathcal{A} \Rightarrow G_\alpha$ according to rules **(I)** - **(V)**.

4.3 An Example

We give here an example of a proof which determines a corresponding type-assignment. Consider the string

whom John loves

Such a sentence determines a program $\mathcal{P}$ with the following set Γ of ground atoms:

$$\{CONN(\text{whom}, 0, 1),\ CONN(\text{John}, 1, 2),\ CONN(\text{loves}, 2, 3)\}$$

We assume lexical type assignments such that the remaining set of clauses Δ is as follows:

$$\{\forall x \forall z[CONN(\mathsf{whom}, x-1, x) \wedge (NP(y,y) \rightarrow S(x,y)) \rightarrow REL(x-1,y)],$$

$$\forall x[CONN(\mathsf{John}, x-1, x) \rightarrow NP(x-1,x)],$$

$$\forall x \forall y \forall z[CONN(\mathsf{loves}, y-1, y) \wedge NP(y,z) \wedge NP(x,y-1) \rightarrow S(x,z)]\}$$

The clause assigned to the relative pronoun whom corresponds to the type of a higher-order function, and contains an implication in its body. Figure 1 shows a proof tree for such a type-assignment. The tree, which is represented as growing up from its root, has size 11, and height 8.

5 Structural Rules

We now briefly examine the interaction of *structural rules* with parsing. In intuitionistic sequent systems, structural rules define ways of subtracting, adding, and reordering hypotheses in sequents during proofs. We have the three following structural rules:

- *Interchange*, which allows to use hypotheses in any order
- *Contraction*, which allows to use a hypothesis more than once
- *Thinning*, which says that not all hypotheses need to be used

5.1 Programs as Unordered Sets of Hypotheses

All of the structural rules above are implicit in proof rules **(I)**-**(V)**, and they are all needed to obtain intuitionistic soundness and completeness as in [7]. By contrast, Lambek's propositional calculus does not have any of the structural rules; for instance, Interchange is not admitted, since the hypotheses deriving the type of a given string must also account for the positions of the words to which they have been assigned as types, and must obey the strict string adjacency requirement between functions and arguments of classical CG. Thus, Lambek's calculus must assume ordered lists of hypotheses, so as to account for word-order constraints. Under our approach, word-order constraints are obtained declaratively, via sharing of string positions, and there is no strict adjacency requirement. In proof-theoretical terms, this directly translates in viewing programs as unordered sets of hypotheses.

5.2 Trading Contraction against Decidability

The logic defined by rules **(I)**-**(V)** is in general undecidable, but it becomes decidable as soon as Contraction is disallowed. In fact, if a given hypothesis can be used at most once, then clearly the number of internal nodes in a proof tree for a sequent $\mathcal{P} \Rightarrow G$ is at most equal to the total number of occurrences of $\rightarrow$, $\wedge$ and $\exists$ in $\mathcal{P} \Rightarrow G$, since these are the logical constants for which proof rules with corresponding inference figures have been defined. Hence, no proof tree can contain infinite branches and decidability follows.

Now, it seems a plausible conjecture that the programs directly defined by input strings as in Section 4.2 never need Contraction. In fact, each time we use a hypothesis in the proof, either we consume a corresponding word in the input string, or we consume a "virtual" constituent corresponding to a step of hypothesis introduction determined by rule **(V)** for implications. (Constructions like parasitic gaps can be accounted for by associating specific lexical items with clauses which determine the simultaneous introduction of gaps of the same type.) If this conjecture can be formally confirmed, then we could automate our formalism via a metainterpreter based on rules **(I)**-**(V)**, but implemented in such a way that clauses are removed from programs as soon as they are used. Being based on a decidable fragment of logic, such a metainterpreter would not be affected by the kind of infinite loops normally characterizing DCG parsing.

5.3 Thinning and Vacuous Abstraction

Thinning can cause problems of overgeneration, as hypotheses introduced via rule **(V)** may end up as being never used, since other hypotheses can be used instead. For instance, the type assignment

$$(7)\ \mathsf{which}: \quad \forall v \forall y \forall w[CONN(\mathsf{which}, v-1, v) \wedge (NP(y,y) \rightarrow S(v,w)) \wedge v \leq y \wedge y \leq w \rightarrow$$

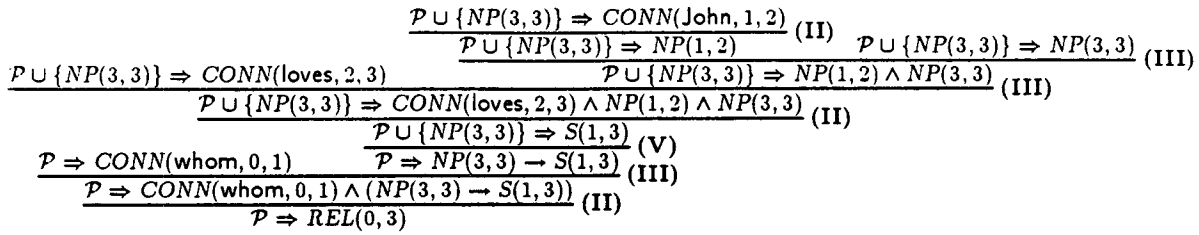

Figure 1: Type derivation for whom John loves

$$REL(v-1, w) \quad]$$

can be used to account for the well-formedness of both

which [I shall put a book on $\uparrow$]

and

which [I shall put $\uparrow$ on the table]

but will also accept the ungrammatical

which [I shall put a book on the table]

In fact, as we do not have to use all the hypotheses, in this last case the virtual noun-phrase corresponding to the extraction site is added to the program but is never used. Notice that our conjecture in section 4.4.2 was that Contraction is not needed to prove the theorems corresponding to the types of grammatical strings; by contrast, Thinning gives us more theorems than we want. As a consequence, eliminating Thinning would compromise the proof-theoretic properties of (I)-(V) with respect to intuitionistic logic, and the corresponding Kripke models semantics of our programming language.

There is however a formally well defined way to account for the ungrammaticality of the example above without changing the logical properties of our inference system. We can encode proofs as terms of Lambda Calculus and then filter certain kinds of proof terms. In particular, a hypothesis introduction, determined by rule (V), corresponds to a step of λ-abstraction, while a hypothesis elimination, determined by one of rules (I)-(II), corresponds to a step of functional application and λ-contraction. Hypotheses which are introduced but never eliminated result in corresponding cases of *vacuous* abstraction. Thus, the three examples above have the three following Lambda encodings of the proof of the sentence for which an extraction site is hypothesized, where the last ungrammatical example corresponds to a case of vacuous abstraction:

- λx put([a book], [on x], I)
- λx put(x, [on the table], I)
- λx put([a book], [on the table], I)

Constraints for filtering proof terms characterized by vacuous abstraction can be defined in a straightforward manner, particularly if we are working with a metainterpreter implemented on top of a language based on Lambda terms, such as Lambda-Prolog [8, 9]. Beside the desire to maintain certain well defined proof-theoretic and semantic properties of our inference system, there are other reasons for using this strategy instead of disallowing Thinning. Indeed, our target here seems specifically to be the elimination of vacuous Lambda abstraction. Absence of vacuous abstraction has been proposed by Steedman [17] as a universal property of human languages. Morrill and Carpenter [11] show that other well-formedness constraints formulated in different grammatical theories such as GPSG, LFG and GB reduce to this same property. Moreover, Thinning gives us a straightforward way to account for situations of lexical ambiguity, where the program defined by a certain input string can in fact contain hypotheses which are not needed to derive the type of the string.

References

[1] Bar-Hillel, Yehoshua. 1953. *A Quasi-arithmetical Notation for Syntactic Description.* Language. 29. pp47-58.

[2] Huet, Gerard 1986. *Formal Structures for Computation and Deduction.* Unpublished lecture notes. Carnegie-Mellon University.

[3] Gabbay, D. M., and U. Reyle. 1984. *N-Prolog: An Extension of Prolog with Hypothetical Implications. I* The Journal of Logic Programming. 1. pp319-355.

[4] Joshi, Aravind. 1987. *Word-order Variation in Natural Language Generation.* In Proceedings of the National Conference on Artificial Intelligence (AAAI 87), Seattle.

[5] Karttunen, Lauri. 1986. *Radical Lexicalism.* Report No. CSLI-86-68. CSLI, Stanford University.

[6] Lambek, Joachim. 1958. *The Mathematics of Sentence Structure.* American Mathematical Monthly. 65. pp363-386.

[7] Miller, Dale. 1987. *A Logical Analysis of Modules in Logic Programming.* To appear in the Journal of Logic Programming.

[8] Miller, Dale and Gopalan Nadathur. 1986. *Some Uses of Higher-order Logic in Computational Linguistics.* In Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, Columbia University.

[9] Miller, Dale and Gopalan Nadathur. 1987. *A Logic Programming Approach to Manipulating Formulas and Programs.* Paper presented at the IEEE Fourth Symposium on Logic Programming, San Francisco.

[10] Moortgat, Michael. 1987. *Lambek Theorem Proving.* Paper presented at the ZWO workshop *Categorial Grammar: Its Current State.* June 4-5 1987, ITLI Amsterdam.

[11] Morrill, Glyn and Bob Carpenter 1987. *Compositionality, Implicational Logic and Theories of Grammar.* Research Paper EUCCS/RP-11, University of Edinburgh, Centre for Cognitive Science.

[12] Pareschi, Remo and Mark J. Steedman. 1987. *A Lazy Way to Chart-parse with Categorial Grammars.* In Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, Stanford University.

[13] Pereira, Fernando C. N. and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis.* CSLI Lectures Notes No. 10. CSLI, Stanford University.

[14] Pereira, Fernando C. N. and David H. D. Warren. 1980. *Definite Clauses for Language Analysis.* Artificial Intelligence. 13. pp231-278.

[15] Rounds, William C. 1987. *LFP: A Logic for Linguistic Descriptions and an Analysis of Its Complexity.* Technical Report No. 9. The University of Michigan. To appear in Computational Linguistics.

[16] Steedman, Mark J. 1985. *Dependency and Coordination in the Grammar of Dutch and English.* Language, 61, pp523-568

[17] Steedman, Mark J. 1987. *Combinatory Grammar and Parasitic Gaps.* To appear in Natural Language and Linguistic Theory.

[18] Uszkoreit, Hans. 1986. *Categorial Unification Grammar.* In Proceedings of the 11th International Conference of Computational Linguistics, Bonn.

[19] Wittenburg, Kent. 1987. *Predictive Combinators for the Efficient Parsing of Combinatory Grammars.* In Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, Stanford University.

[20] Zeevat, H., Klein, E., and J. Calder. 1987. *An Introduction to Unification Categorial Grammar.* In N. Haddock et al. (eds.), Edinburgh Working Papers in Cognitive Science, 1: Categorial Grammar, Unification Grammar, and Parsing.